Answer Key to

Object-Oriented Modeling and Design with UML, Second Edition

Michael Blaha
Modelsoft Consulting Corporation

James Rumbaugh
IBM

Pearson Prentice Hall
Upper Saddle River, New Jersey 07458

Contents

1 Introduction

[There are no right or wrong answers to the first four exercises, which are intended to give you some feedback on the background of the students and to get them thinking about the value of using a software design methodology.]

1.1 The amount of time spent on analysis, design, coding, and testing/debugging/fixing depends on the methodology used. Using an OO approach, we find that our effort is split approximately 20% on analysis, 30% on design, 40% on coding, and 10% on testing/debugging/fixing problems. The exact split depends on factors such as the type of system and the amount of experience with similar systems. We have found that paying extra attention to analysis and design cuts the total development time. It is a lot easier to avoid a problem during analysis and design than it is to find and fix it later on.

One of the most difficult areas of project management is estimating how much effort a project will require. One method that we use is to break the total effort down into several tasks. We think about each task separately and estimate how much effort is required, based on our experience with similar tasks.

One major problem that we have encountered is underestimating the time and effort required to complete a project. Most software projects are finished late and over budget. We try to deal with this by carefully considering our estimates and explicitly allowing for contingencies.

Premature implementation is another problem. Because of an anxiety to complete a project, developers sometimes substitute implementation for design, resulting in systems that are hard to debug. The resulting software suffers from unwarranted assumptions and fuzzy thinking. With such an approach, developers bog down in details and find it difficult to see flaws.

In one of our projects we quickly prototyped a trivial master-slave communications system. We did the initial coding in a week without benefit of a thoughtful design. The resulting system crashed due to occasional data communications errors. We applied many patches to the system over a period of three months. Each time we thought we understood the problem only to find that we really did not. The problem seduced us into investigating many dead ends. We finally scrapped our initial design and started over again. We put more thinking into our second design and successfully completed the communications system.

Another problem we often face is uncertain or changing requirements. For example, another one of our projects changed its hardware platform several times during the course of the project, from a PC to a workstation and then back to a PC. Each time, we ported an uncompleted system, wasting time and effort. This contributed to the failure of the project. What we should have done was complete the system on one platform first and port it to other platforms later.

1.2 [Expect a wide range of answers to this question. Here is an answer based on one of our early applications at GE R&D.]

We created an editor for power system circuit diagrams that served as a graphical front end for capturing parameters needed for simulating their performance. Our main obstacle was integrating several subsystems over which we had no control. We used a structured analysis and design methodology that someone else selected over our objections. We would have preferred a rapid prototyping approach. The focal point of the system was a database, which we designed using OO techniques. Although the project failed, we salvaged several good ideas for other projects.

1.3 [Expect a variety of answers.] Many software systems suffer from these problems, including the system described in the previous answer, which was behind schedule and over budget. Contributing factors were pressures to underestimate the effort and the decision to build the system on top of several subsystems that were being developed separately. Development of one of the subsystems, a graphical interface, ran into problems of its own, and did not become available until after the target completion date of our system.

1.4 [The point of this exercise is for the students to realize that it is easy to create systems that annoy users, and to motivate them to consider the user's position.]

Software systems that truncate names is one of our pet peeves. Truncation can cause many unexpected problems. One check issuing system truncates first names to 5 letters and last names to 7 letters, resulting in checks that some banks refuse to cash. This could have been avoided by a better design.

Another annoying situation is the handling of foreign currency by vending machines. Change dispensed by bill changing machines may contain foreign currency that is refused by adjacent food vending machines.

1.5 a. Addresses can be used to identify mail recipients. The format of an address, which varies with country, often includes name, street, city, state/province, postal code, and country. An address both identifies the recipient of mail and encodes instructions for its delivery.

b. Criminal investigations can use combinations of photographs, fingerprinting, blood-typing, DNA analysis, and dental records to identify people, living and/or deceased, who are involved in, or the subject of, a criminal investigation.

c. Banks can use a variety of schemes to identify safe deposit customers. Usually name plus some other piece of information such as an account number, driver's license, or address is used. Other answers are possible.

d. Telephone numbers are adequate for identifying almost any telephone in the world. In general a telephone number consists of a country code plus a province, city, or area code, plus a local number plus an optional extension number. Businesses may have their own telephone systems with other conventions. Depending on the relative location of the telephone that you are calling, parts of the number may be implied and can be left out, but extra access digits may be required to call outside the local region.

In North America most local calls require 7 digits. Long distance calls in North America use an access digit (0 or 1) + area code (3 digits) + local number (7 digits). Dialing Paris requires an access code (011) + country code (33) + city code (1) + local number (8 digits). The access code is not part of the identifier.

e. Accounts can be used by telephone companies for billing purposes. A single account may be for one or for several telephone numbers. Account information includes account ID, name, and address. The account ID identifies the account. One of the telephone numbers in the account could be used in the construction of the ID. A bill for the service provided to all of the telephone numbers in an account can be sent to the account address.

f. There are logical as well as physical electronic mail addresses used in electronic networks. The formats depend on the particular network. A physical address is a sequence of bytes assigned to a hardware device such as a workstation or a computer at the time of its manufacture, and uniquely identifies the device. A logical address identifies a user on a system. On the Internet there are domain names (such as *edu*, *org*, and *com*). Organization names are unique within a domain and users within an organization. Thus some email addresses are blaha@computer.org and rumbaugh@us.ibm.com.

g. One way that employees are given restricted, after-hours access to a company is through the use of a special, electronically-readable card. Of course, if an employee loses a card and does not report it, someone who finds it could use it for unauthorized entry. Other approaches include a picture ID which requires inspection by a guard, fingerprint readers, and voice recognition.

1.6 [Expect a wide variety of answers. The point of the exercise is for the student to begin to think in terms of classes.]

a. Classes that you would expect in a program for newspaper layout include *Page, Column, Line, Headline,* and *Paragraph.*

b. Classes that you would expect in a program to compute and store bowling scores include *Bowler, Frame, Pin, Score,* and *BallWithinFrame.*

c. Some classes for voice mail include *Telephone, Greeting, Message,* and *Distribution List.*

d. Classes for a controller for a video cassette recorder include *Timer, Channel, Tapedeck,* and *TV.*

e. For a catalog store order entry system, classes include *Customer, Order, Store,* and *Item.*

1.7 For a variable length array the operations behave as follows:

- *Append* adds an object to the end of an array. Duplicates are allowed.
- *Copy* makes a copy of an array. All values in the array elements are copied but the objects referenced are not copied recursively.
- *Count* returns the number of elements in an array.
- *Delete* removes an element from an array. The position of the element to be deleted must be specified. All higher numbered elements are shifted down by one position. If the position is out of range, the operation is ignored. The operation returns the change in size of the array, -1 if deletion occurs, otherwise 0.
- *Index* retrieves an object from an array at a given position. NULL is returned if the index is out of range.
- *Intersect* is not defined for arrays.
- *Insert* places an object into an array at a given position. All elements at the given index or higher are shifted up by one position. An error message is printed if the position is out of range.
- *Update* places an object into an array at a given position, overwriting whatever is there. If the position is out of range, the array is extended with NULLs. The operation returns the change in the size of the array.

For a symbol table (also known as a dictionary) the operations behave as follows:

- *Append* makes sense only for sorted tables, in which case an entry goes at the end unless the table already contains the keyword. In that case the new entry replaces the old entry. Duplicates are not allowed. The operation returns the change in the size of the table.
- *Copy* makes a copy of a table.
- *Count* returns the number of entries in a table.
- *Delete* removes an entry from a table. The entry to be deleted is specified by key word. If the entry does not exist the operation is ignored. The operation returns the change in the size of the table.
- *Index* retrieves an entry from a table that matches a given keyword.
- *Intersect* is not defined for symbol tables.
- *Insert i*s not defined for symbol tables. Use *update* instead.

- *Update* adds an entry to a table. If the keyword is not yet in the table, a new entry is made. If the keyword is already in the table, the entry in the table is updated.The operation returns the change in the size of the table.

Operations on sets behave as follows:

- *Append* is not defined for a set, since elements of a set are not ordered.
- *Copy* makes a copy of a set.
- *Count* returns the number of elements in a set.
- *Delete* removes a given element of a set. If the element does not exist the operation is ignored.The operation returns the change in the size of the set.
- *Index* is not defined for a set.
- *Intersect* performs set intersection of two given sets, creating a new set.
- *Insert i*s not defined for sets. Use *update* instead.
- *Update* adds an element to a set. If the element is not yet in the set, it is added to the set. If it is already in the set, the operation is ignored.The operation returns the change in the size of the set.

1.8 [Adding more classes to each list is optional. We have given a few classes to get the student to think abstractly. We made no attempt to give exhaustive lists in the exercise.]

a. Electron microscopes, eyeglasses, telescopes, bomb sights, and binoculars are all devices that enhance vision in some way. With the exception of the scanning electron microscope, all these devices work by reflecting or refracting light. Eyeglasses and binoculars are designed for use with two eyes; the rest of the objects on the list are designed for use with one eye. Telescopes, bomb sights, and binoculars are used to view things far away. A microscope is used to magnify something that is very small. Eyeglasses may enlarge or reduce, depending on whether the prescription is for a near-sighted or a far-sighted person. Some other classes that could be included in this list are optical microscopes, cameras, and magnifying glasses.

b. Pipes, check valves, faucets, filters, and pressure gauges are all plumbing supplies with certain temperature and pressure ratings. Compatibility with various types of fluids is also a consideration. Check valves and faucets may be used to control flow. With the exception of the pressure gauge, all of the items listed have two ends and have a pressure-flow characteristic for a given fluid. All of the items are passive. Some other classes include pumps, tanks, and connectors.

c. These objects are all means for transportation. Bicycles, cars, trucks, motorcycles, and horses are used on land. Sailboats are used on water. Airplanes and gliders are used in the air. Students may discover other common traits.

d. These are all fasteners. The terms screw and bolt have similar meanings. The term screw is used to refer to wood screws, self-tapping sheet metal screws, and bolts. The term bolt refers to a straight threaded screw. Nails, screws, and bolts are used for carpentry. Bolts and rivets are used in the assembly of machinery.

e. These are all forms of shelter. Any of them afford protection from the rain. People normally live in houses or skyscrapers, although the rest would do in an emergency. Tents, sheds, garages, barns, houses, and skyscrapers are man made. Caves are natural. All except tents are more or less permanent structures. Garages, barns, sheds, and houses are typically made out of wood, brick, or sheet metal. Skyscrapers require special construction techniques. Sheds, garages, and barns are used to store things.

Part 1: Modeling Concepts

2

Modeling as a Design Technique

[The first four exercises emphasize that the content of a model is driven by its relevance to the problem to be solved. The first three exercises are hardware oriented. Exercise 2.4 is software oriented.]

2.1 In purchasing a tire for a car, size is generally constrained to fit the car. In selecting tires, many consumers pay close attention to cost and expected life. Tread design and internal construction are broadly matched to the expected service. For example snow tires provide extra traction in snow. Material and weight are generally not considered.

Size, material, internal construction, tread design, and weight are important physical parameters that would be taken into account in simulating the performance of a computerized anti-skid system for cars. Other physical parameters of the car itself would have to be considered. Cost and expected life would be irrelevant.

In building a tire swing for a child, cost would probably be the main consideration. A discarded tire would be a good candidate. Weight and size would also be relevant. You would probably not use a giant truck tire for a swing, although you might use it in constructing a playground

2.2 Cost, stiffness, availability, and strength would be considered in selecting a wire to unclog a drain. Depending on the urgency of the situation, you might prefer something immediately available. If you went out to buy something you would not want to pay more for the wire than what it would cost to hire a plumber. The wire would have to be stiff enough to push through the clog but would have to be flexible enough to follow the bends of the drain pipe.

We have not had much luck unclogging drains with most common wire such as coat hanger wire or electrical wire. We have had some modest success with a special coiled spring sold in some hardware stores and with chemical caustics. For really tough jobs you should hire a plumber.

2.3 a. For a transatlantic cable, resistance to saltwater is the main consideration. The cable must lie unmaintained at the bottom of the ocean for a long time. Interaction of ocean life with the cable and the effect of pressure and salinity on cable life must be considered. The ratio of strength/weight is important to avoid breakage while the cable is being installed. Cost is an important economic factor. Electrical parameters are important for power consumption and signal distortion.

b. Color, cost, stiffness, and availability are the main considerations. You would probably want an assortment of colors to make the artwork interesting. The wire should be stiff enough to hold its shape after being bent, but flexible enough to be shaped.

c. Weight is very important for wire that is to be used in the electrical system of an airplane, because it affects the total weight of the plane. Toughness of the insulation is important to resist chafing due to vibration. Resistance of the insulation to fire is also important to avoid starting or feeding electrical fires in flight.

d. Cost, stiffness, availability, and strength should be considered in selecting wire to hang a bird feeder. The wire should be flexible enough to work with and strong enough to hold the bird feeder. Another consideration not mentioned in the exercise that is important if bare wire is selected is resistance to corrosion.

e. Cost, stiffness, availability, strength, and resistance to stretching are important considerations in selecting wire for use as piano strings. Because the strings are under a great deal of tension, strength and resistance to stretching are important. Stiffness is important because it affects the way the strings vibrate.

f. Because the filament of a light bulb operates at a high temperature, resistance to high temperatures is important. Tungsten is generally used because of its high melting point, even though tungsten filaments are brittle.

2.4 Electrical noise, buffering and flow control, and character interpretation are relevant in designing a protocol for transferring computer files from one computer to another over telephone lines. Data transmission rate is a secondary factor since it limits the overall speed of the protocol.

Electrical noise determines how much error detection and correction is needed.

Buffering and flow control are techniques that are used in some protocols if the receiving computer cannot keep up with the incoming stream of data.

Control characters in the transmitted data could cause problems if the protocol is not designed with them in mind since they could be interpreted as part of the protocol or the receiving computer could interpret them as commands instead of as data.

2.5 [This exercise illustrates the importance of having multiple models or views of a problem.]

a. The electrical model is the most important model to determine how much electrical power is required to run a motor. It relates voltage, current, and perhaps frequency to the speed and torque requirements of the load. Mechanical and fluid models play secondary roles. A mechanical model is used to compute friction and a fluid model is used to compute the energy needed to drive any fans that keep the motor cool. Waste heat is equal to the difference between the electrical input power and the mechanical output power.

b. The mechanical model is the only one needed to determine motor weight. The weight of each part can be determined from its mechanical dimensions and its density. The total motor weight is equal to the sum of the weights of all of its parts.

c. Electrical, mechanical, thermal, and fluid models must be considered to determine how hot a motor gets. Electrical, mechanical, and fluid models are used to compute waste heat. The amount of cooling air is determined from a fluid model. The temperature rise of the motor is determined from the waste heat, the thermal model, and the cooling air flow rate.

d. Vibration is computed from electrical and mechanical models. An electrical model is used to determine time variations in electrical torque and motor speed. A mechanical model is used to determine the dynamic mechanical behavior of parts of the motor driven by electrical torques, mechanical loads, and unbalance.

e. Bearing wear is computed from electrical, mechanical, thermal, and fluid models. The mechanical model is used to compute the forces on the bearing. Electrical, mechanical, thermal, and fluid models are used to determine bearing temperature. Fluid and mechanical models are used to analyze the lubrication of the bearing.

2.6 a. Class and state models are relevant to the user interface. A class model can be used to represent the pieces being moved. A state model is used to define the protocol of the user interaction. The interaction model would be less important because chess pieces are largely autonomous and mostly interact with the chess board.

b. A class model is used to show a board configuration.

c. The class model is most important because it represents the relationships among the pieces. There would also need to be significant algorithms for exploring the space of possible moves. These algorithms are not explicitly addressed by the three models and would require a supplemental notation.

d. Move validation involves the class model which represents the pieces.

3

Class Modeling

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

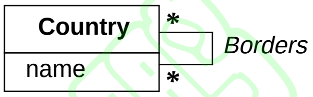


Figure A3.1 Class diagram for international borders

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.

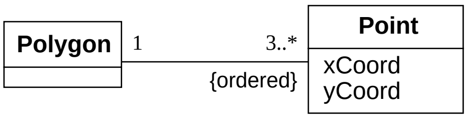


Figure A3.2 Class diagram for polygon and points

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.

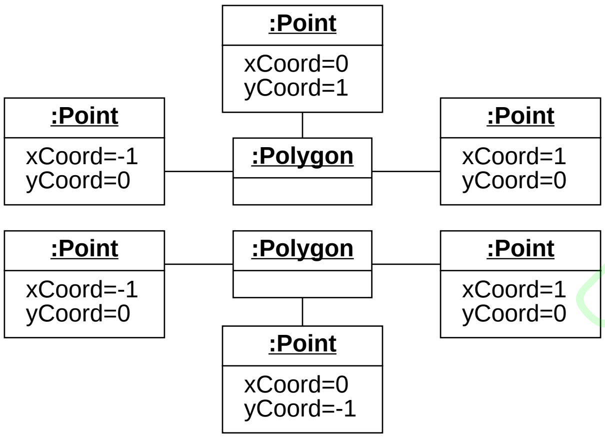


Figure A3.3 Object diagram where each point belongs to exactly one polygon

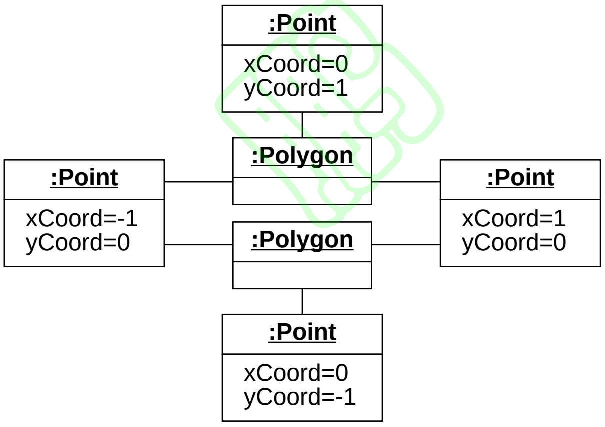


Figure A3.4 Object diagram where each point can belong to multiple polygons

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

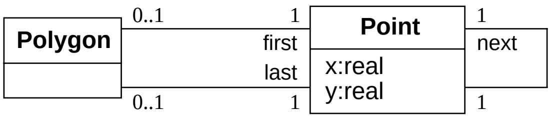


Figure A3.5 Another class diagram for polygon and points

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.

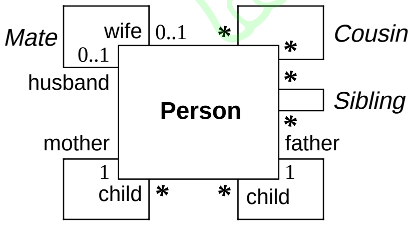


Figure A3.6 Class diagram for family trees

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.

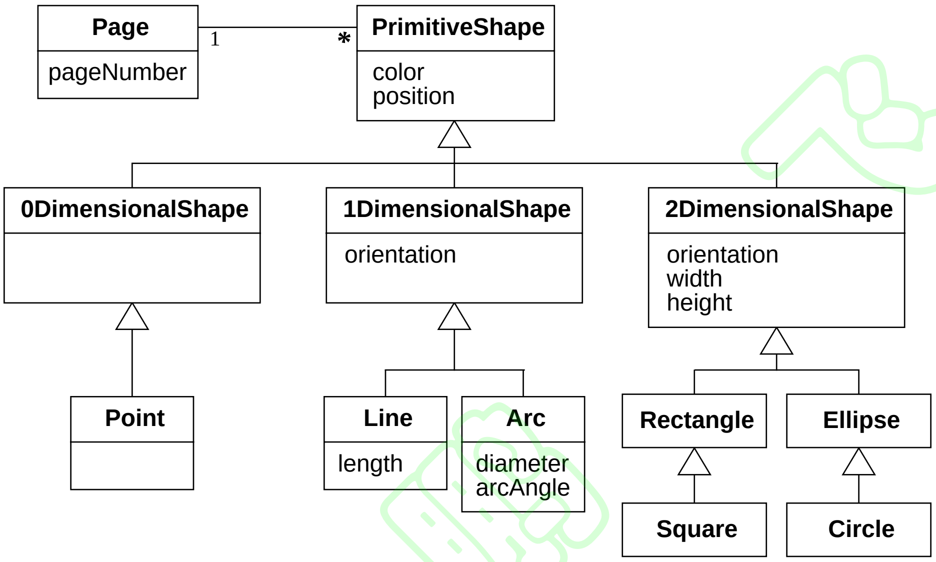


Figure A3.7 Class diagram for geometrical documents

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.

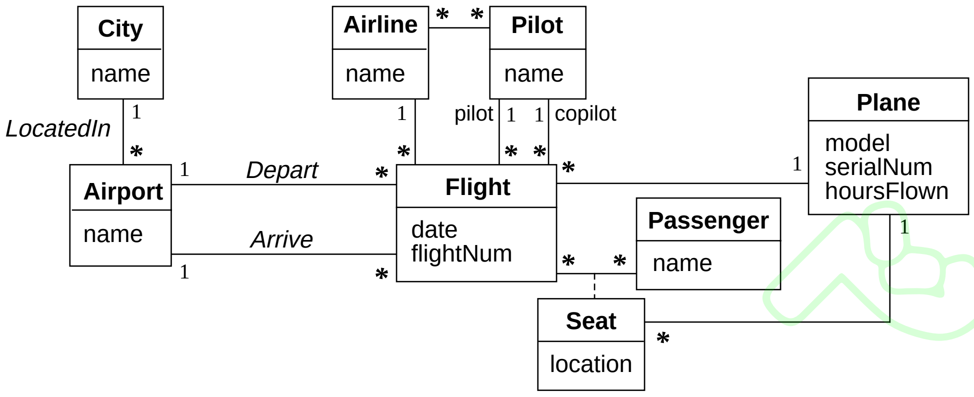


Figure A3.8 Class diagram for an air transportation system

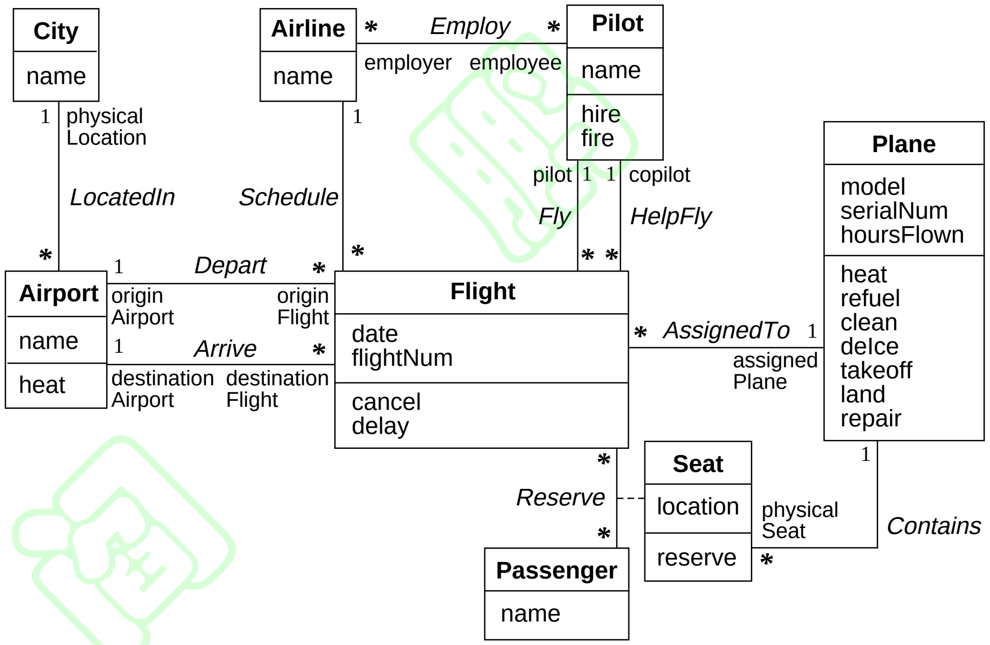


Figure A3.9 Class diagram for an air transportation system with operations, association names, and association end names

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.

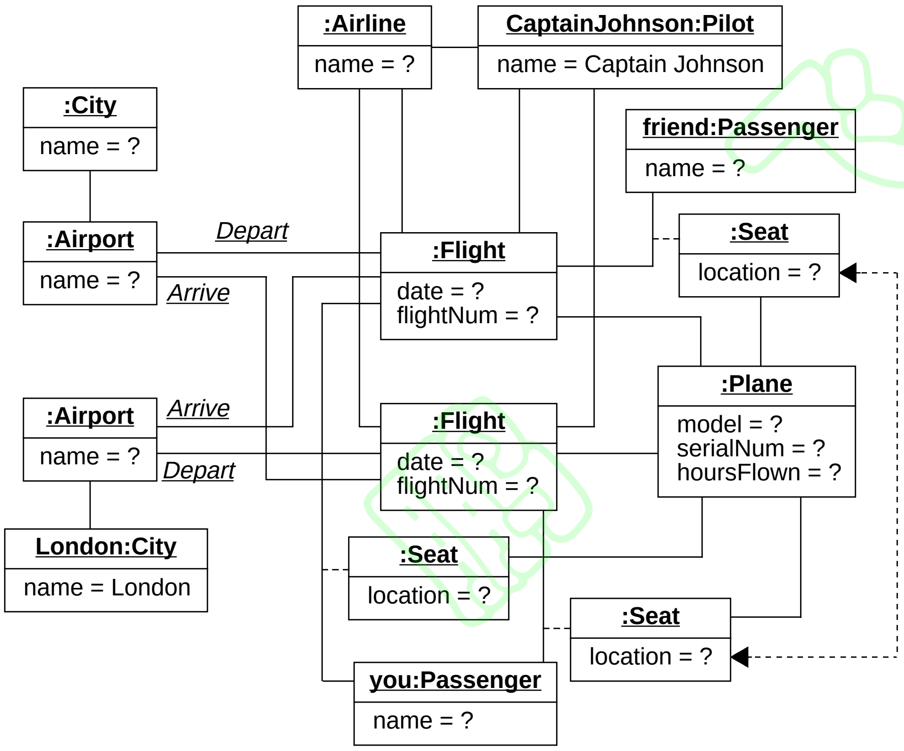


Figure A3.10 Object diagram for an air transportation system

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.

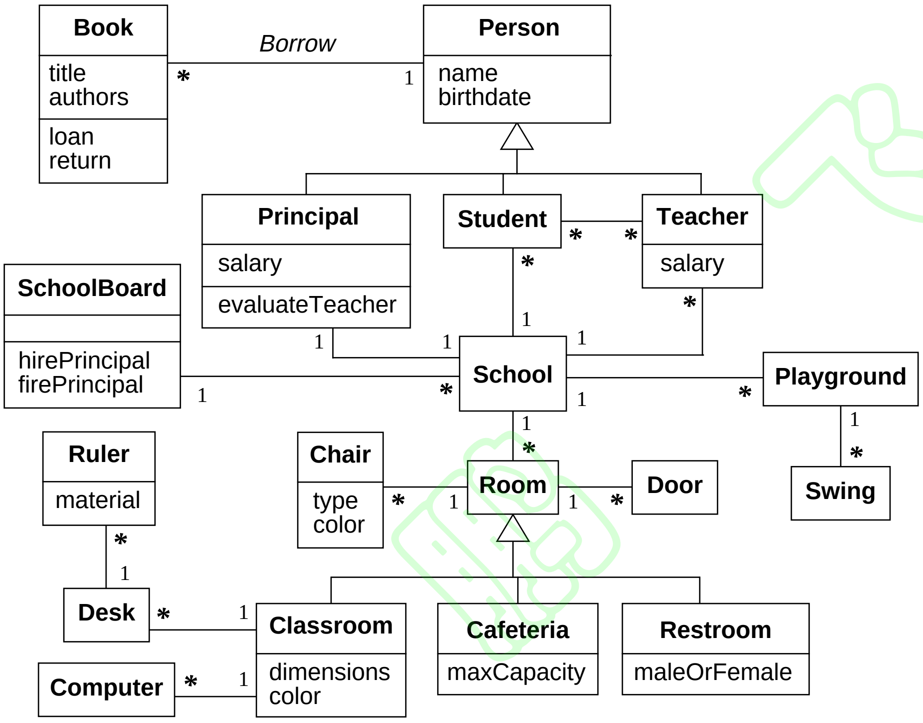


Figure A3.11 Class diagram for a school

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.

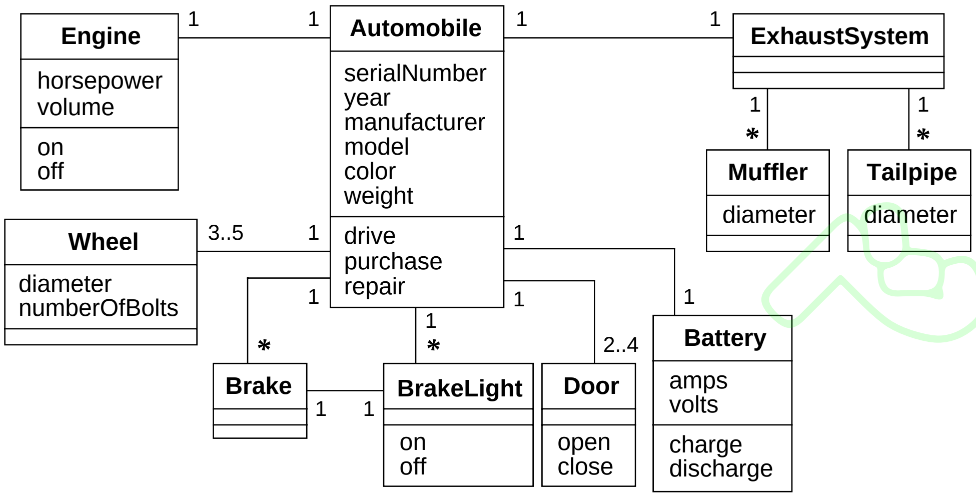


Figure A3.12 Class diagram for an automobile

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-

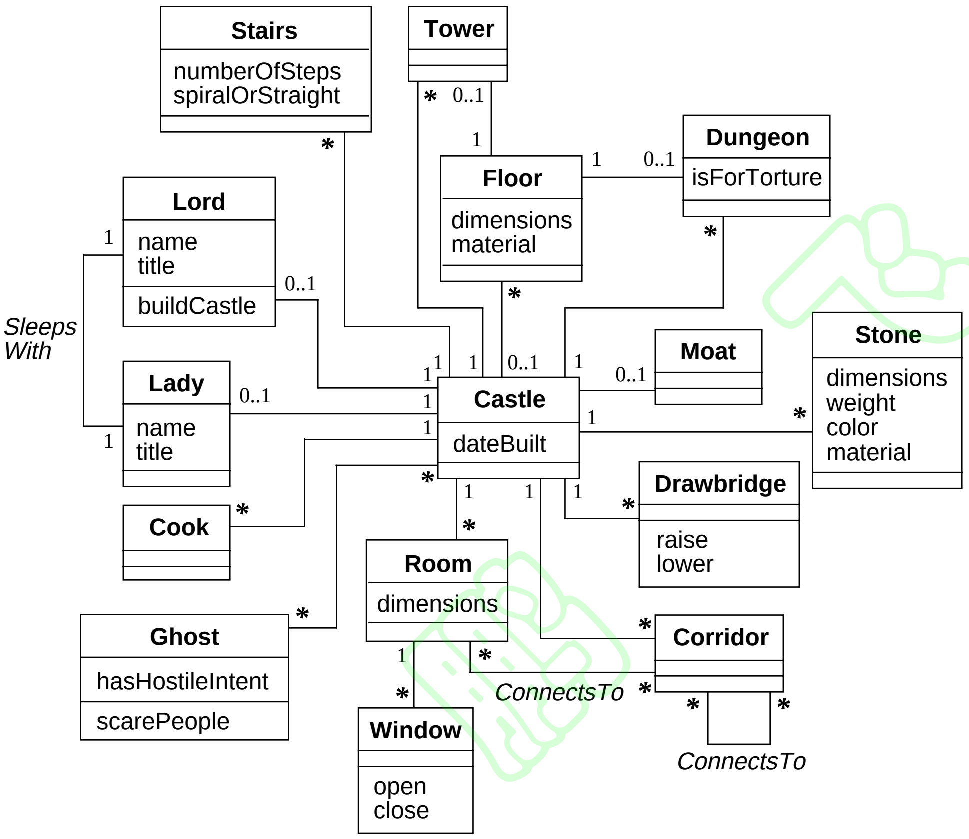


Figure A3.13 Class diagram for a castle

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

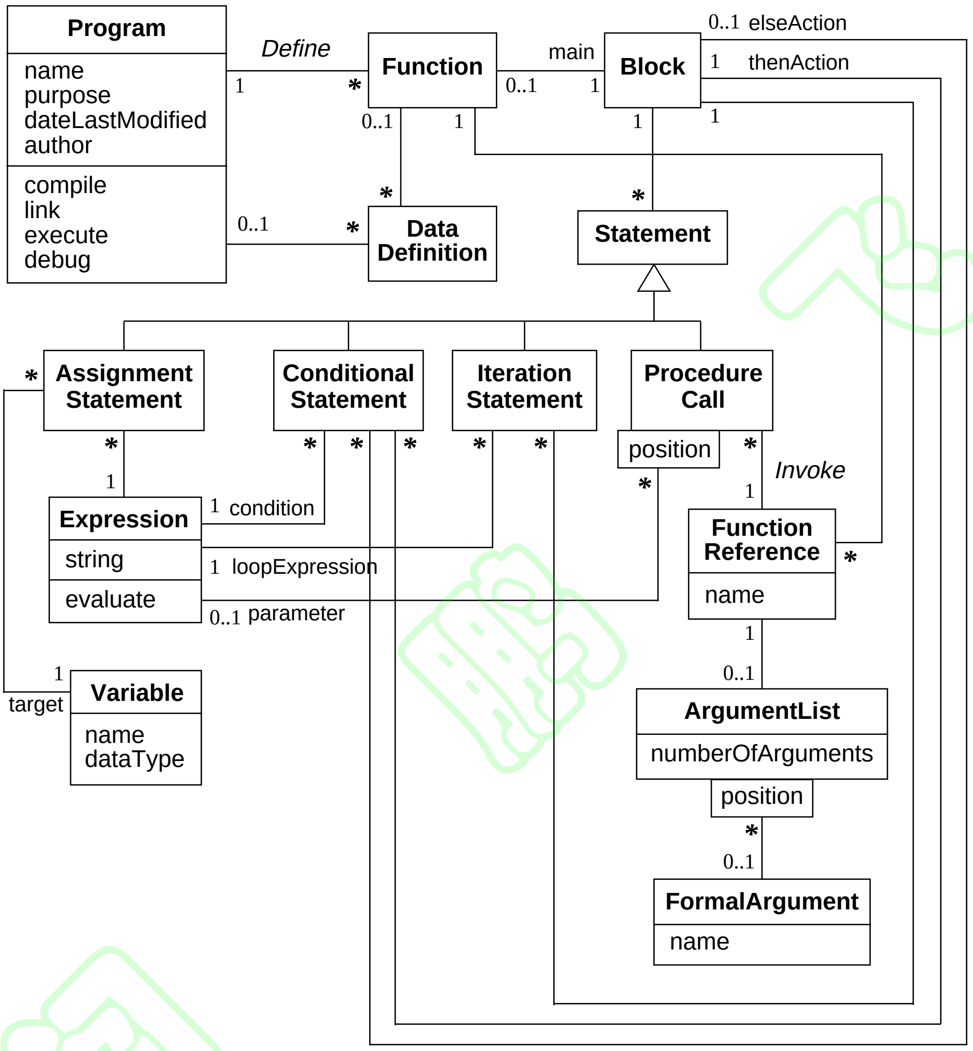


Figure A3.14 Class diagram for a computer program—part 1

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.

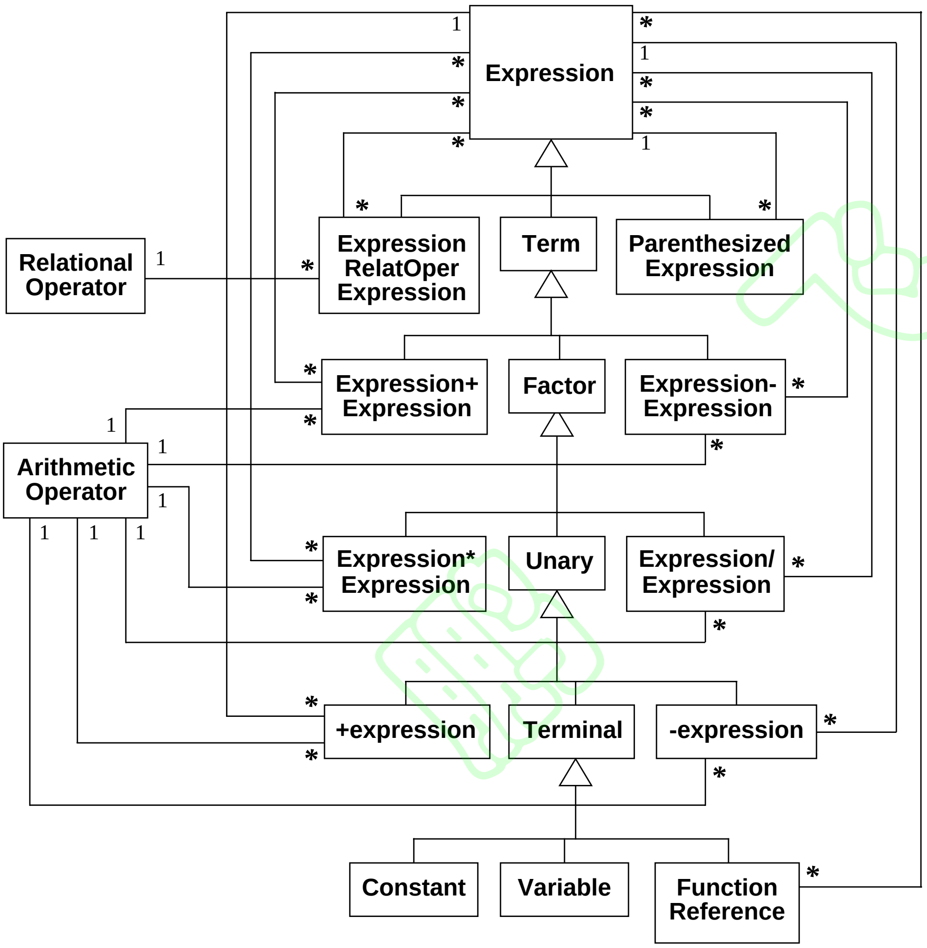


Figure A3.15 Class diagram for a computer program—part 2

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

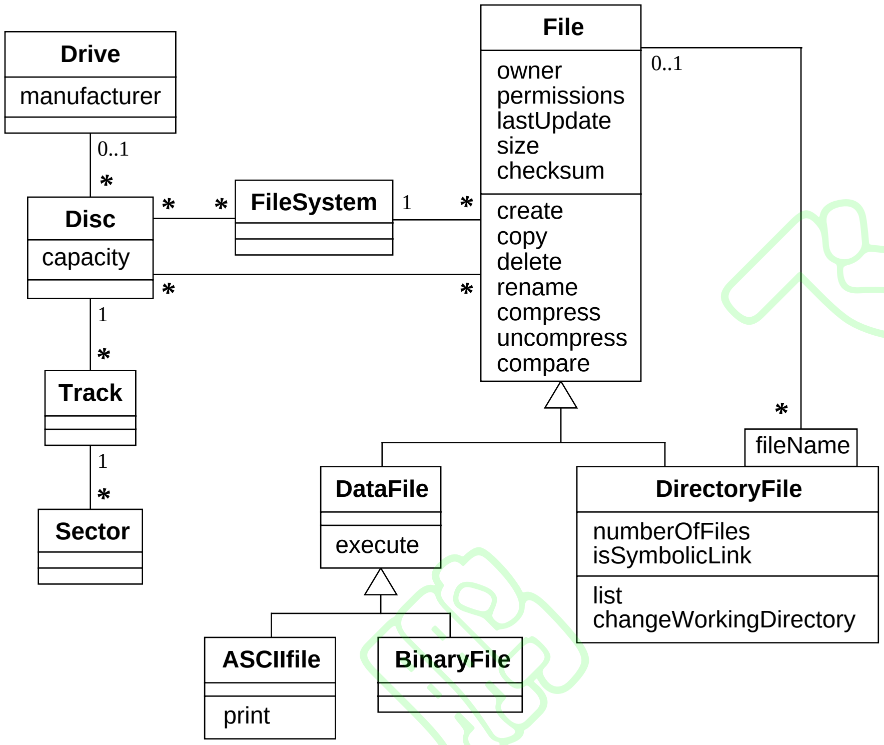


Figure A3.16 Class diagram for a file system

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.

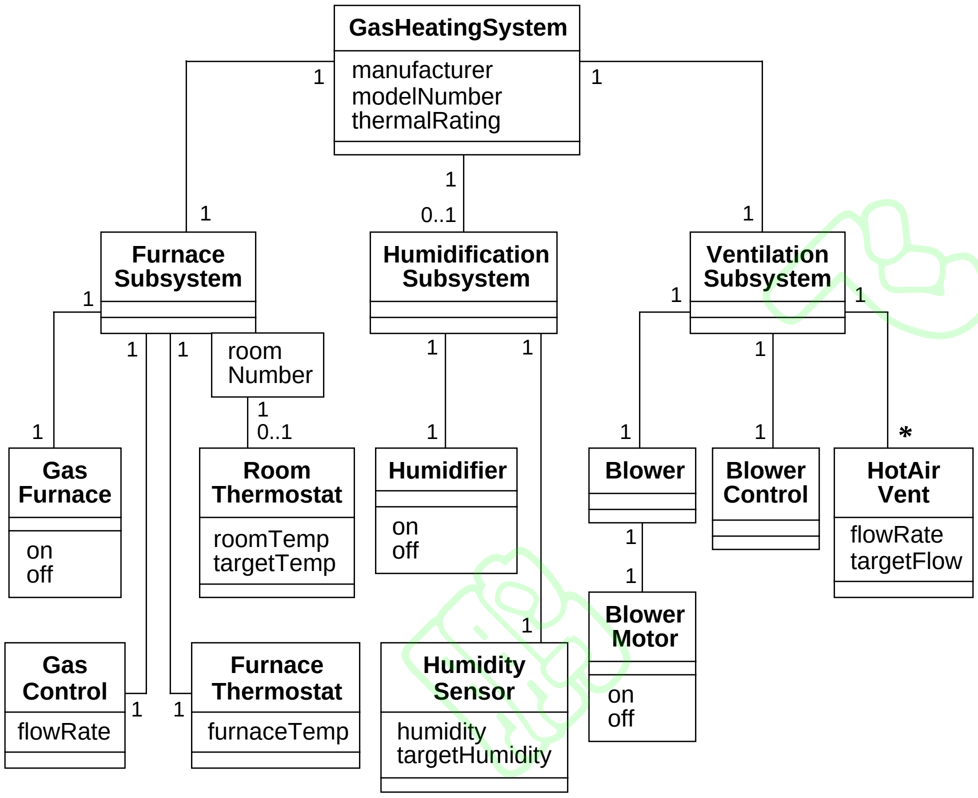


Figure A3.17 Class diagram for a gas-fired, hot-air heating system

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*.

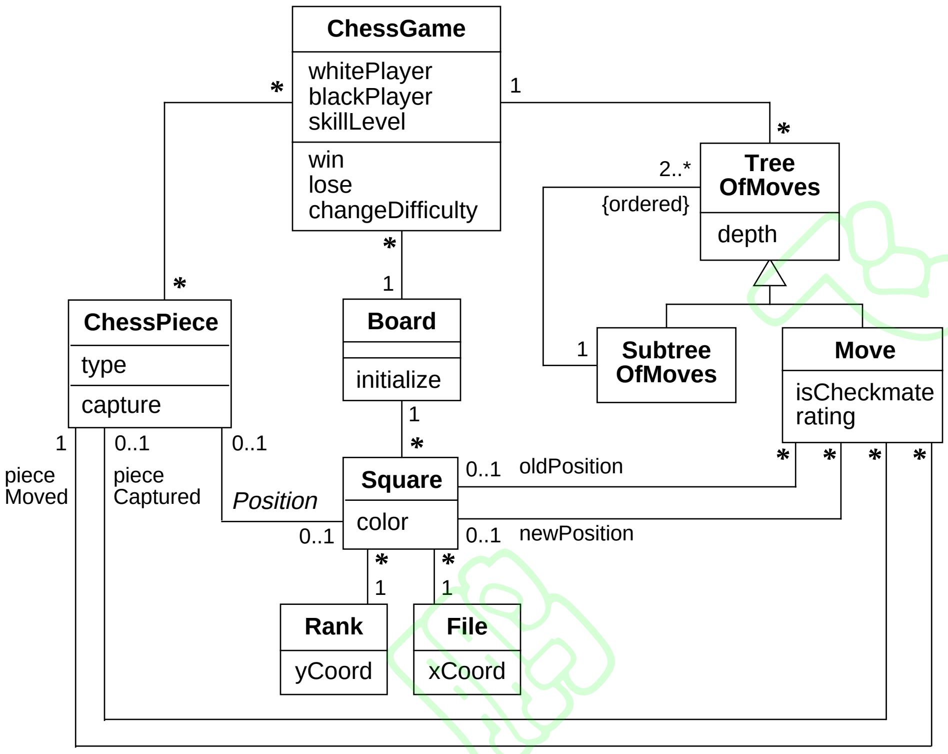


Figure A3.18 Class diagram for a chess game

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-

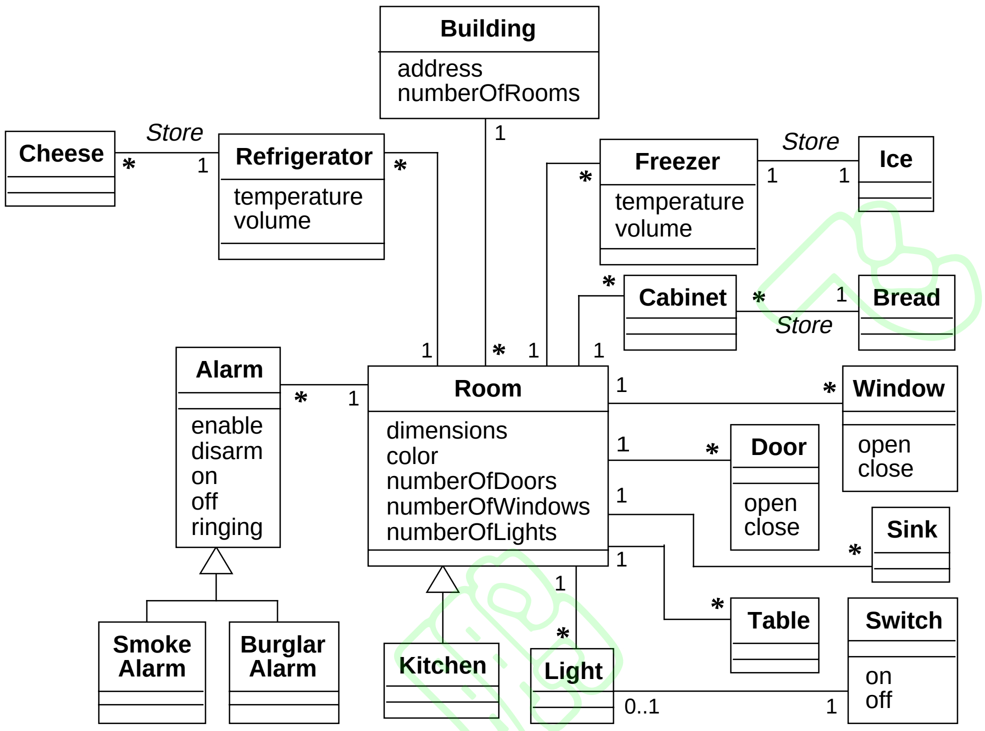


Figure A3.19 Class diagram for a building

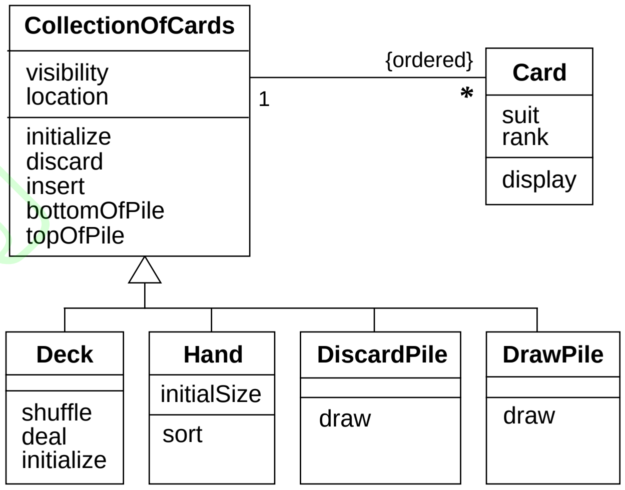


Figure A3.20 Portion of a class diagram for a card playing system

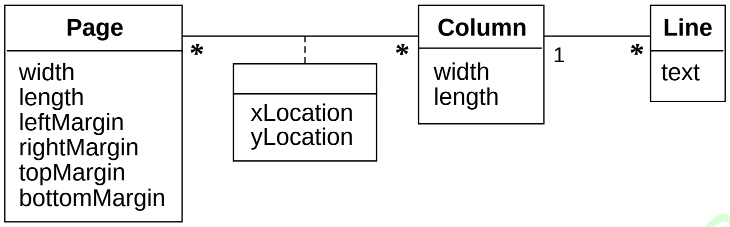


Figure A3.21 Portion of a class diagram for a newspaper publishing system

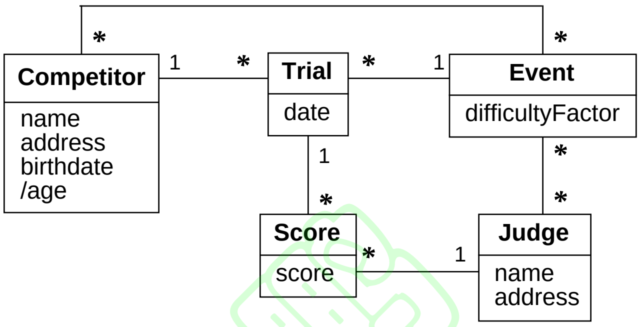


Figure A3.22 Portion of a class diagram for an athletic event scoring system

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.

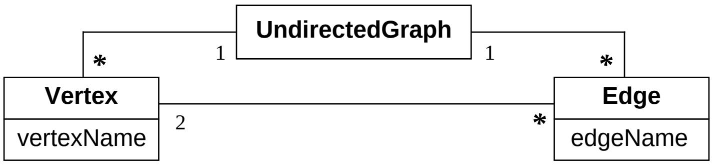


Figure A3.23 Class diagram for undirected graphs

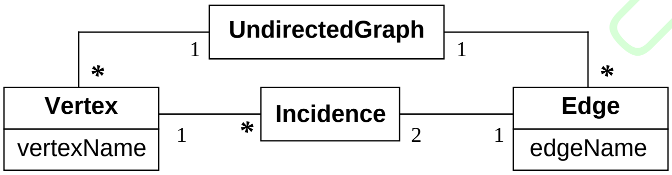


Figure A3.24 Class diagram for undirected graphs in which the incidence between vertices and edges is treated as a class

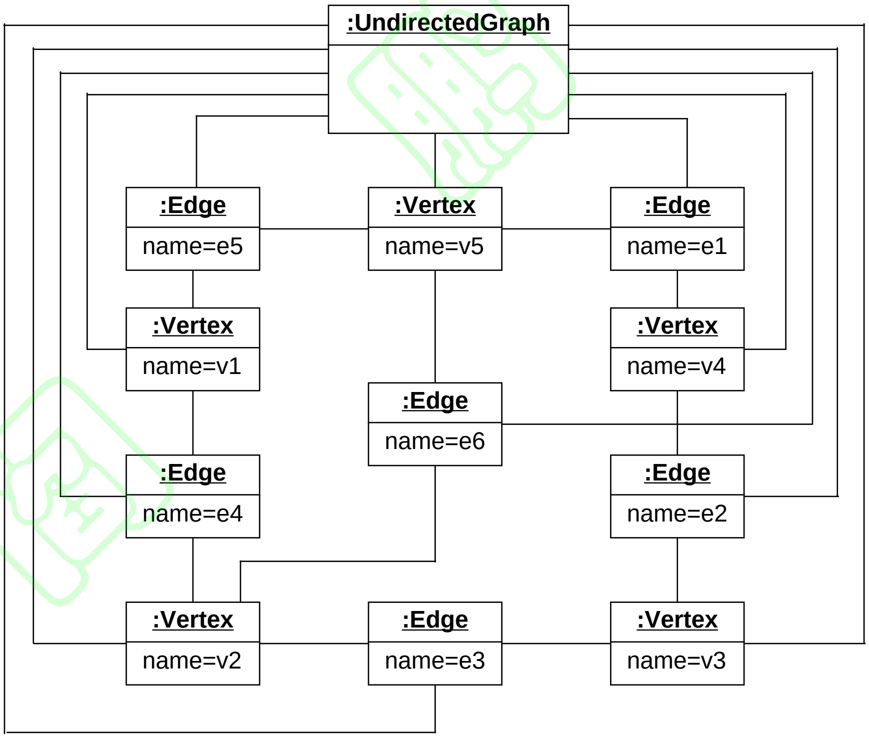


Figure A3.25 Object diagram for the sample undirected graph

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.

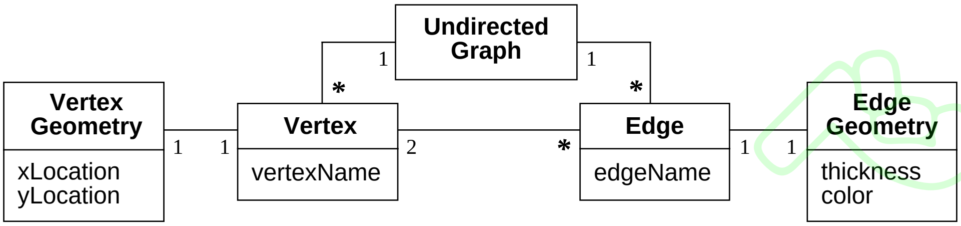


Figure A3.26 Class diagram for undirected graphs with geometrical details

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.*

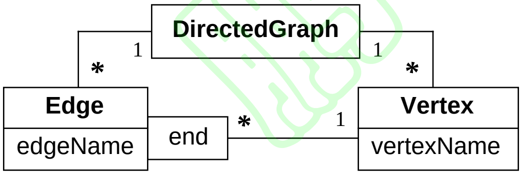


Figure A3.27 Class diagram for directed graphs using a qualified association

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

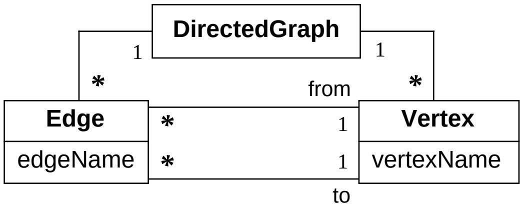


Figure A3.28 Class diagram for directed graphs using two associations

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.

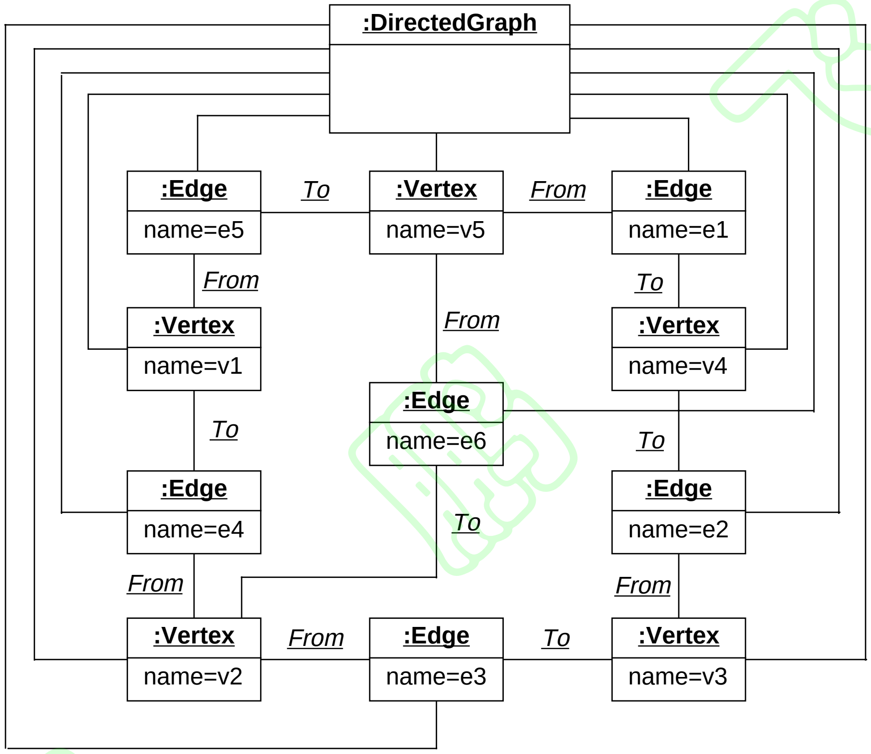


Figure A3.29 Object diagram for the sample directed graph

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

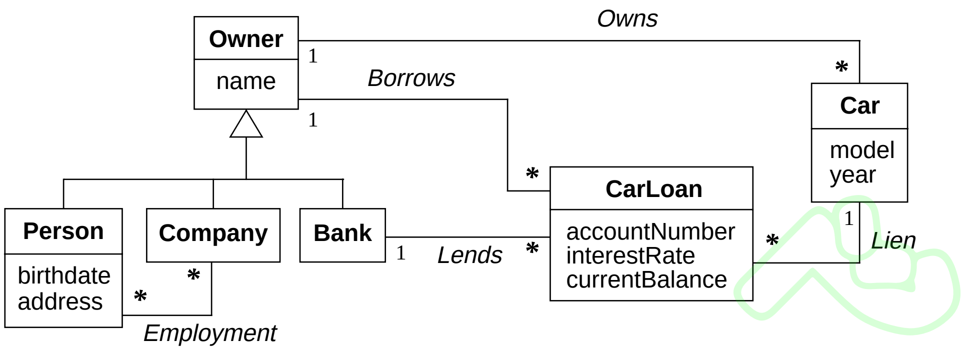


Figure A3.30 Proper class diagram for car loans

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.

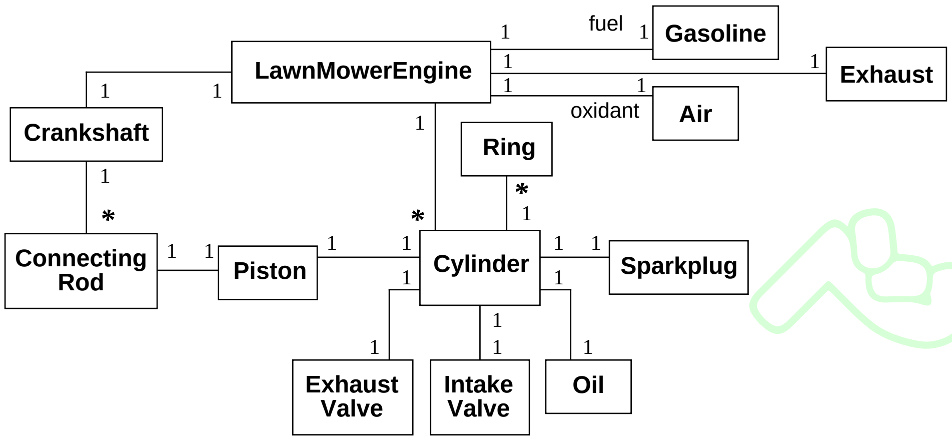


Figure A3.31 Class diagram for 4-cycle lawn mower engine—part 1

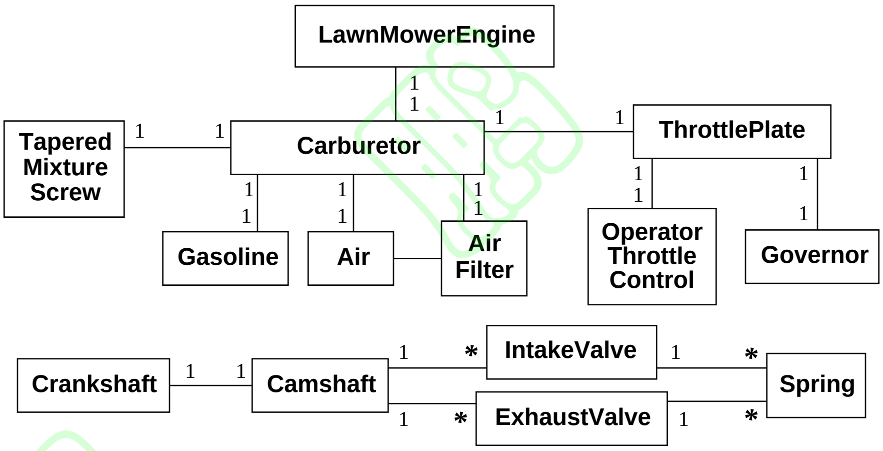


Figure A3.32 Class diagram for 4-cycle lawn mower engine—part 2

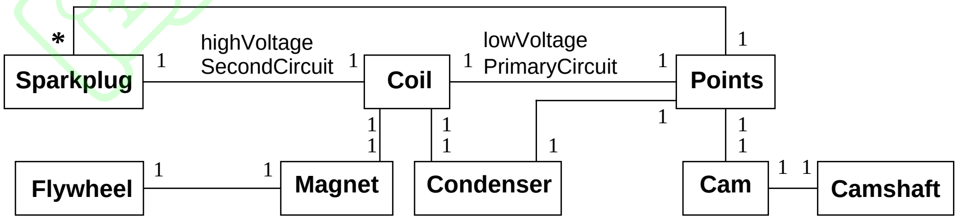


Figure A3.33 Class diagram for 4-cycle lawn mower engine—part 3

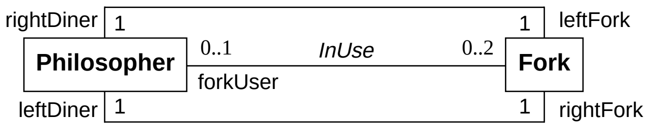


Figure A3.34 Class diagram for the dining philosopher problem

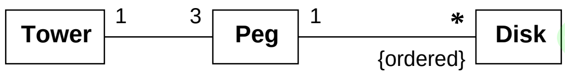


Figure A3.35 Simple class diagram for tower of Hanoi problem

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

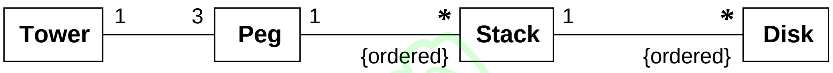


Figure A3.36 Tower of Hanoi class diagram with disks organized into stacks

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.

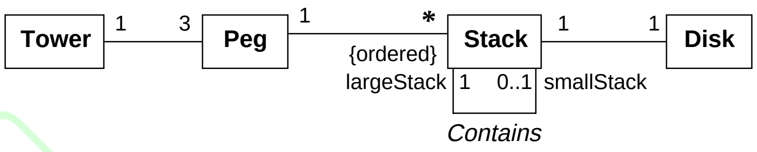


Figure A3.37 Tower of Hanoi class diagram in which the structure of stacks is recursive

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*.

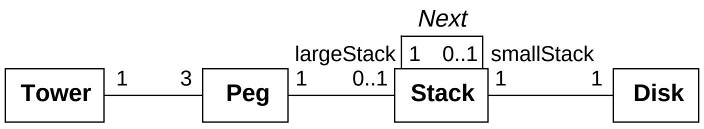


Figure A3.38 Tower of Hanoi class diagram in which stacks are organized into a linked list

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.

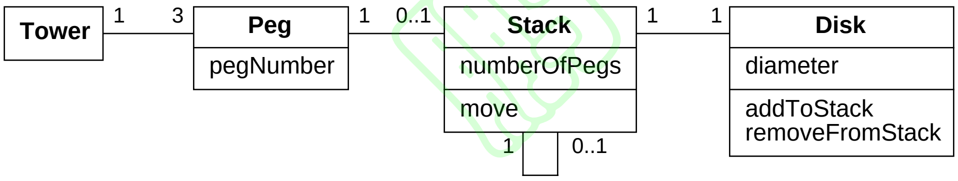


Figure A3.39 Detailed tower of Hanoi class diagram in which the structure of stacks is recursive

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.

4

Advanced Class Modeling

4.1 Figure A4.1 improves the class diagram in the exercise by changing some associations to aggregations. For a bill-of-material parts hierarchy, all parts except the root must belong to something.

Note that none of the aggregations are composition—the parts do not have coincident lifetime with the assemblies.

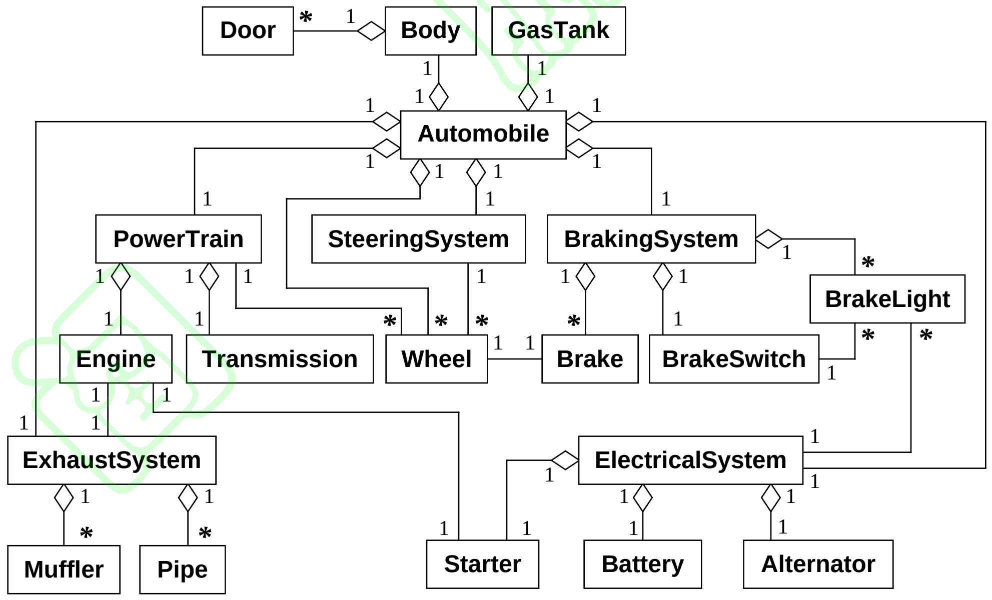


Figure A4.1 Portion of a class diagram of the assembly hierarchy of an automobile

4.2 The class diagram in Figure A4.2 generalizes the classes *Selection, Buffer,* and *Sheet* into the superclass *Collection*. This is a desirable revision. The generalization promotes code reuse because many operations apply equally well to the subclasses. Six aggregation relationships in the original diagram, which shared similar characteristics, have been reduced to two. Finally, the structure of the diagram now captures the constraint that each *Box* and *Line* should belong to exactly one *Buffer, Selection,* or *Sheet*.

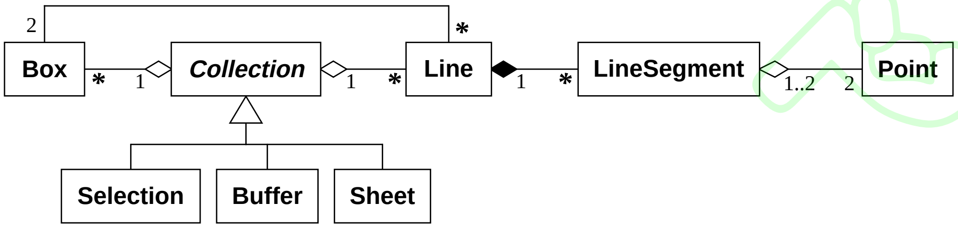


Figure A4.2 Generalization of the classes *Selection*, *Buffer*, and *Sheet* into the class *Collection*

4.3 a. *A country has a capital city.* Association. A capital city and a country are distinct things so generalization certainly does not apply. You could argue that a capital city is a part of a country and thus they are related by aggregation.

b. *A dining philosopher uses a fork.* Association. Dining philosophers and forks are completely distinct things and are therefore not in a generalization relationship. Similarly, neither object is a part of the other and the relationship is not aggregation.

c. *A file is an ordinary file or a directory file.* Generalization. The word “or” often is an indicator of generalization. *File* is the superclass and *OrdinaryFile* and *DirectoryFile* are subclasses.

d. *Files contain records.* Aggregation. The word “contain” is a clue that the relationship may be aggregation. A record is a part of a file. Some attributes and operations on files propagate to their constituent records.

e. *A polygon is composed of an ordered set of points.* Aggregation. The phrase “is composed of” should immediately make you suspicious that there is an aggregation. An ordered set of points is a part of a polygon. Some attributes and operations on a polygon propagate to the corresponding set of points.

f. *A drawing object is text, a geometrical object, or a group.* Generalization. Once again, the word “or” should prompt you to think of generalization. *DrawingObject* is the superclass. *Text*, *GeometricalObject*, and *Group* are subclasses.

g. *A person uses a computer language on a project.* Ternary association. *Person, ComputerLanguage*, and *Project* are all classes of equal stature. The association cannot be reduced to binary associations. None of these classes are a-kind-of or a-part-of another class. Thus generalization and aggregation do not apply.

h. *Modems and keyboards are input / output devices.* Generalization. The keyword "are" is the clue. *Modem* and *Keyboard* are the subclasses; *InputOutputDevice* is the superclass.

i. *Classes may have several attributes.* Association or aggregation. It depends on your perspective and the purpose of the model whether aggregation applies.

j. *A person plays for a team in a certain year.* Ternary association. *Person*, *Team*, and *Year* are distinct classes of equal stature.

k. *A route connects two cities.* Association. Either *Route* is a class associated with the *City* class, or *Route* is the name of the association from *City* to *City*.

l. *A student takes a course from a professor.* Ternary association. *Student*, *Course*, and *Professor* are distinct classes of equal stature.

4.4 Figure A4.3 shows a class diagram for a graphical document editor. The requirement that a *Group* contain 2 or more *DrawingObject*s is expressed as a multiplicity of 2..* on *DrawingObject* in its aggregation with *Group*. The fact that a *DrawingObject* need not be in a *Group* is expressed by the zero-one multiplicity.

It is possible to revise this diagram to make a *Circle* a special case of an *Ellipse* and to make a *Square* a special case of a *Rectangle*.

We presume that a *DrawingObject* belongs to a *Sheet* and has a coincident lifetime with it. Similarly, we presume that a *Sheet* belongs to one *Document* for its lifetime. Hence both are composition relationships.

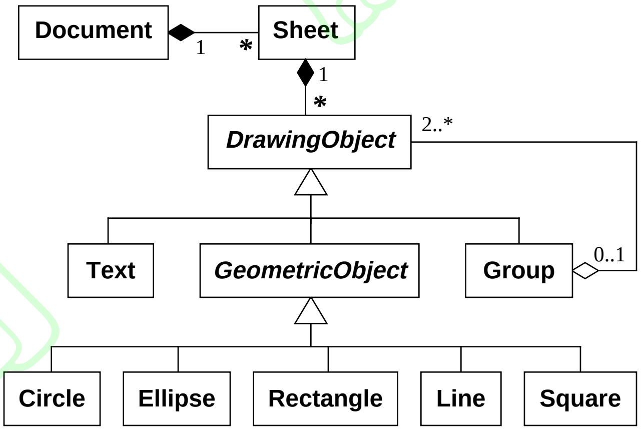


Figure A4.3 Class diagram for a graphical document editor that supports grouping

4.5 Figure A4.4 shows a class diagram with several classes of electrical machines. We have included attributes that were not requested.

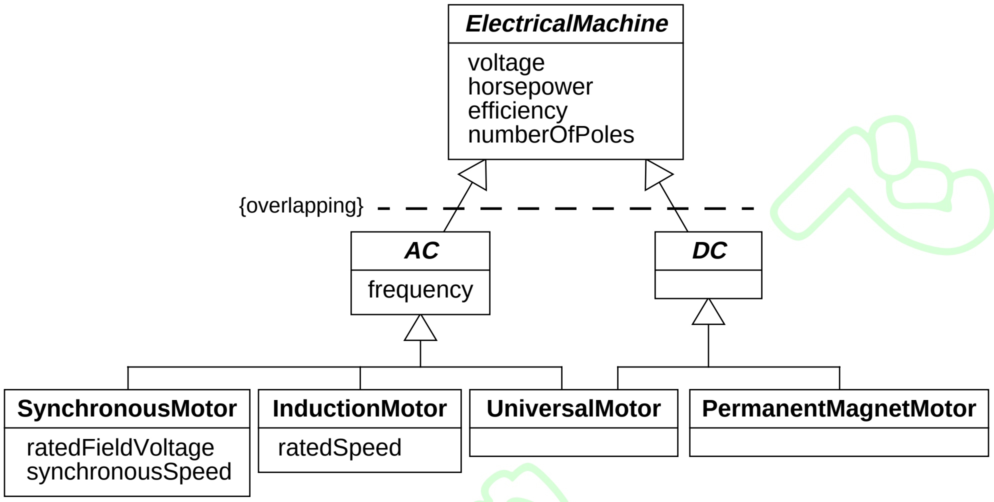


Figure A4.4 Partial taxonomy for electrical machines

4.6 Figure A4.5 converts the overlapping combination of classes into a class of its own to eliminate multiple inheritance. (Instructor's note: you may want to give the students a copy of our answer to the previous exercise.)

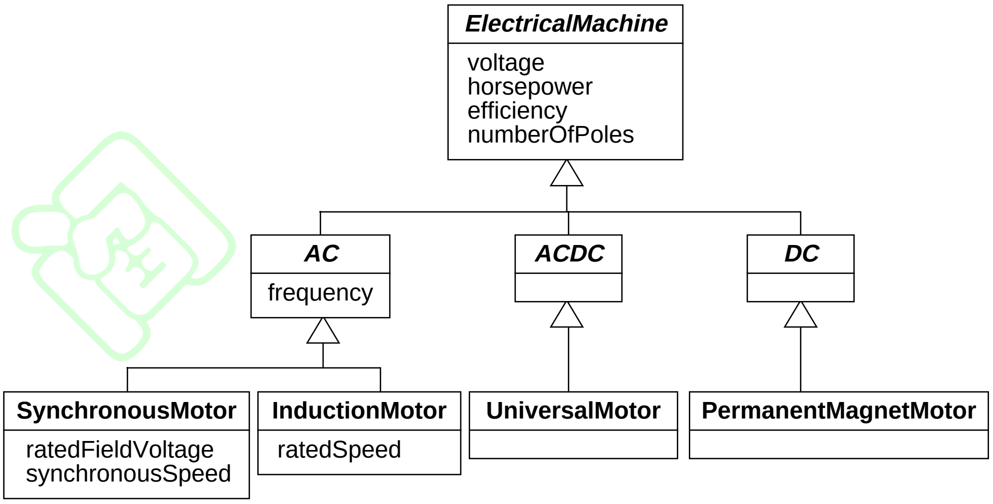


Figure A4.5 Elimination of multiple inheritance

4.7 Figure A4.6 is a metamodel of the following UML concepts: class, attribute, association, association end, multiplicity, class name, and attribute name.

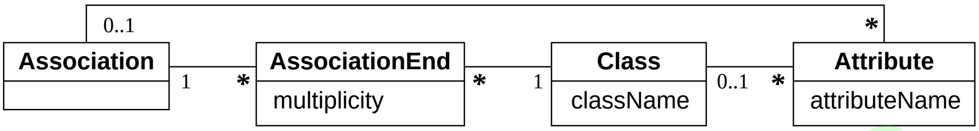


Figure A4.6 Metamodel for some UML concepts

4.8 Figure A4.7 treats the metamodel as a class diagram that can be described by itself. The metamodel in Figure A4.6 is self-descriptive. In general some metamodels are self-descriptive and some are not. (Instructor's note: you may want to give the students a copy of our answer to the previous exercise.)

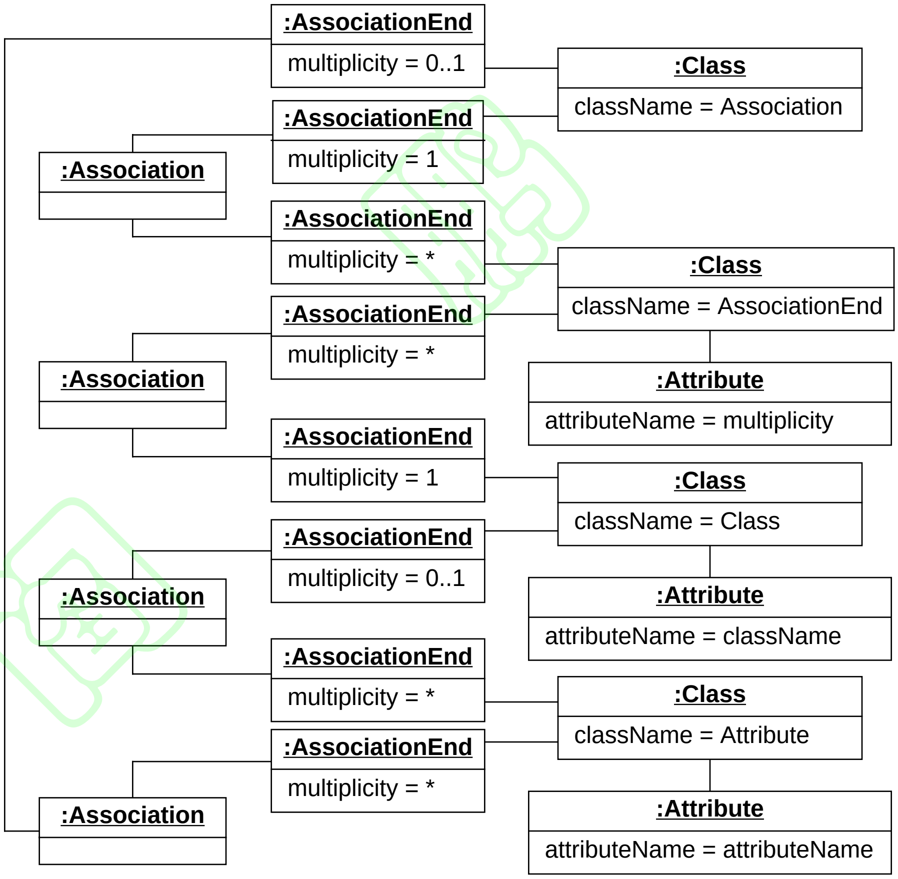


Figure A4.7 Instance diagram for the metamodel

4.9 Figure A4.8 revises the metamodel so that an attribute belongs to exactly one class or association. (Instructor's note: you may want to give the students a copy of our answer to Exercise 4.7.)

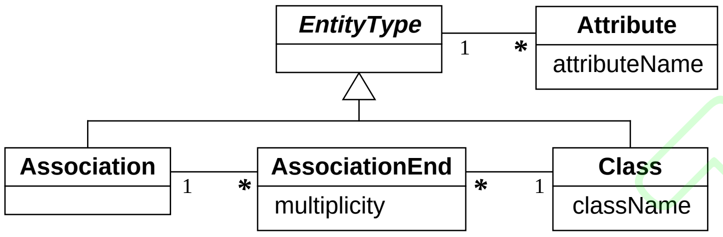


Figure A4.8 UML metamodel where an attribute belongs to exactly one class or association

4.10 The class diagram in Figure E4.3 does support multiple inheritance. A class may have multiple generalization roles of subclass participating in a variety of generalizations.

4.11 To find the superclass of a generalization using Figure E4.3, first query the association between *Generalization* and *GeneralizationRole* to get a set of all roles of the given instance of *Generalization*. Then sequentially search this set of instances of *GeneralizationRole* to find the one with *roleType* equal to *superclass*. (Hopefully only one instance will be found with *roleType* equal to *superclass*, which is a constraint that the model does not enforce.) Finally, scan the association between *GeneralizationRole* and *Class* to get the superclass.

Figure A4.9 shows one possible revision which simplifies superclass lookup. To find the superclass of a generalization, first query the association between *Generalization* and *SuperclassRole*. Then query the association between *SuperclassRole* and *Class* to find the corresponding instance of *Class*.

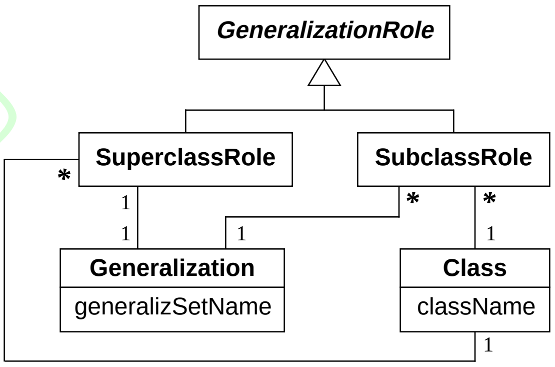


Figure A4.9 Metamodel of generalizations with separate subclass and superclass roles

Figure A4.10 shows another metamodel of generalization that supports multiple inheritance. To find the superclass of a generalization using this metamodel, simply query the *Superclass* association.

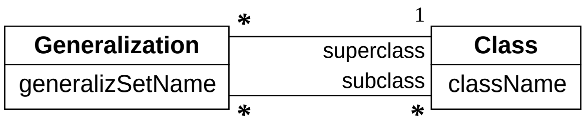


Figure A4.10 Simplified metamodel of generalization relationships

We do not imply that the metamodel in Figure A4.10 is the best model of generalization, only that it simplifies the query given in the exercise. The choice of which model is best depends on the purpose of the metamodel.

The following query finds the superclass given a generalization for Figure E4.3.

- aGeneralization.GeneralizationRole->SELECT(roleType='superclass').Class

The following query finds the superclass given a generalization for Figure A4.9.

- aGeneralization.SuperclassRole.Class

The following query finds the superclass given a generalization for Figure A4.10.

- aGeneralization.superclass

4.12 The metamodel in Figure E4.3 does not enforce the constraint that every generalization has exactly one superclass. In this figure, a *Generalization* has many *Generalization-Roles*; each *GeneralizationRole* may have a *roleType* of *superclass* or *subclass*. Figure A4.9 and Figure A4.10 are improved metamodels and enforce the constraint that every generalization has exactly one superclass.

4.13 Figure A4.11 is the instance diagram that corresponds to Figure E4.3 and Figure E4.4.

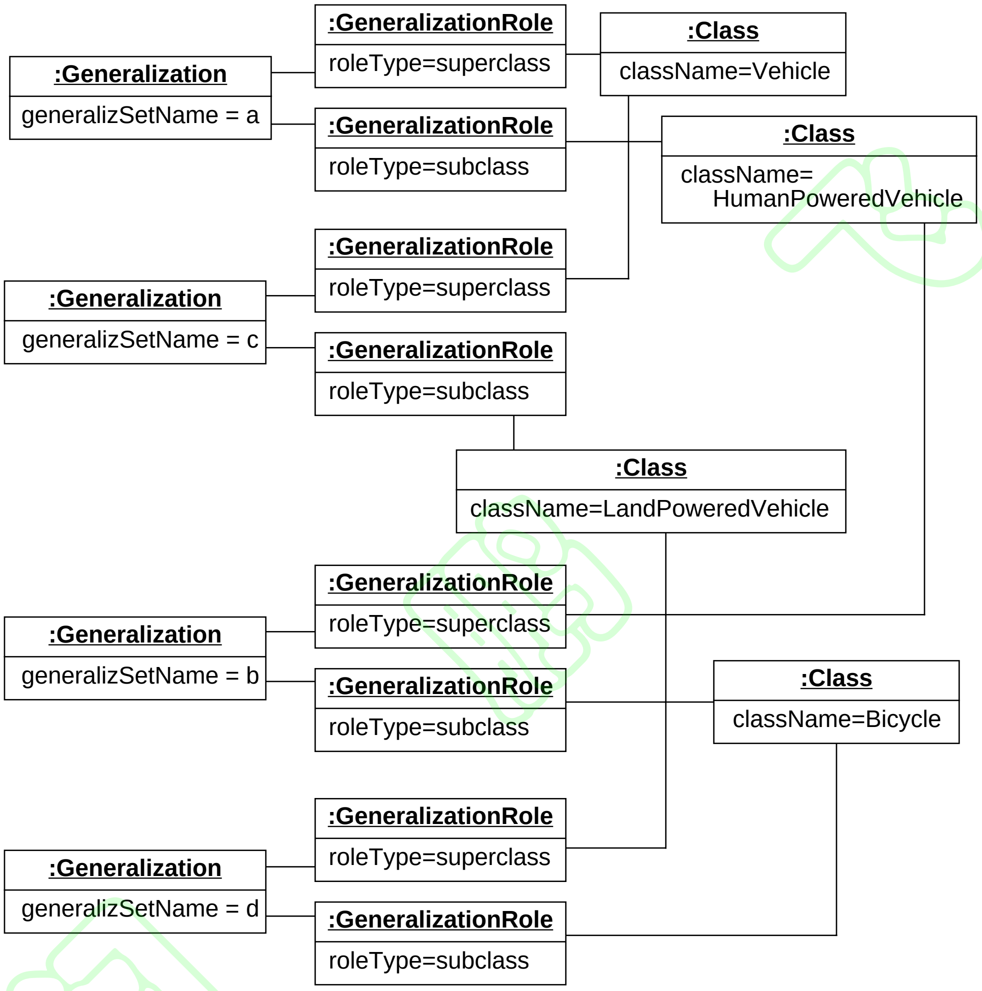
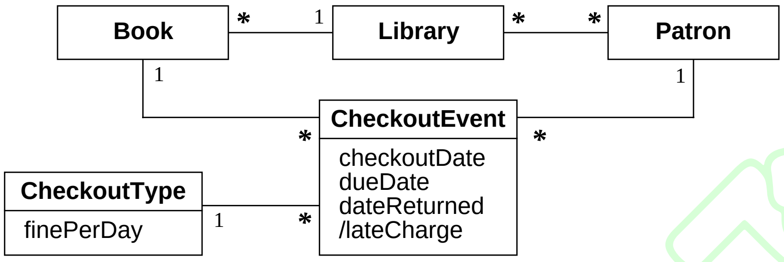


Figure A4.11 Instance diagram for multiple inheritance

4.14 Figure A4.14 shows our answer to the exercise.



{lateCharge = (dateReturned - dueDate) * CheckoutType.finePerDay}

Figure A4.12 Class diagram for library book checkout system

4.15 Figure A4.13 presents one possible metamodel for BNF grammars. A BNF grammar consists of many production rules. Each production rule has many or-clauses which in turn reference many identifiers. An identifier may be a terminal such as a character string or a number or may be a non-terminal in which case it is defined by another production rule.

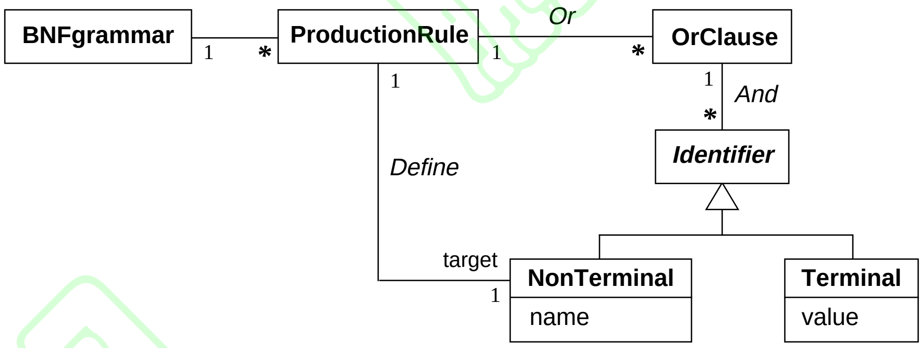


Figure A4.13 Metamodel for BNF grammars

In general, there are at least three ways to express BNF grammars: "railroad" diagrams as shown in the exercise, textual production rules, and state machine diagrams. The metamodel in Figure A4.13 describes the underlying meaning of a BNF grammar and is independent of the manner used to express it. Thus the metamodel applies equally well to each of these three ways of expressing a BNF grammar.

If you were developing software to automate the drawing of BNF "railroad" diagrams, you would need a model of the BNF logic as Figure A4.13 shows and another model to store the corresponding graphic representation. The graphical model would describe the size of rectangles and circles, the placement of geometric shapes, and the stopping and starting position for lines and arrows.

4.16 The simple class model in Figure A4.14 is sufficient for describing the given recipe data.

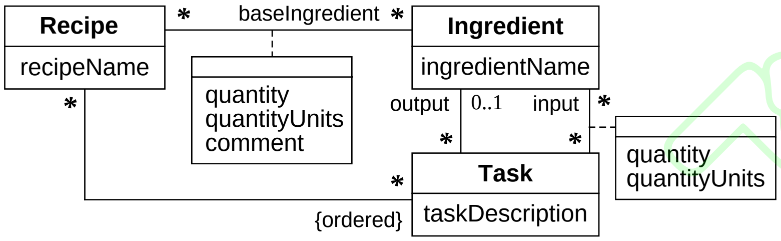


Figure A4.14 A simple class model for recipes

4.17 Figure A4.15 shows our initial solution to the exercise—merely adding an association that binds original ingredients to substitute ingredients. This model has two flaws.

The first problem is that the model awkwardly handles interchangeable ingredients. For example, in some recipes you can freely substitute butter, margarine, and shortening for each other. Figure A4.15 would require that we store each possible pair of ingredients. Thus we would have the following combinations of original and substitute ingredients—(butter, margarine), (butter, shortening), (margarine, butter), (margarine, shortening), (shortening, butter), and (shortening, margarine).

The second problem is that the substitutability of ingredients does not always hold, but can depend on the particular recipe.

Figure A4.16 shows a better class model that remedies both flaws.

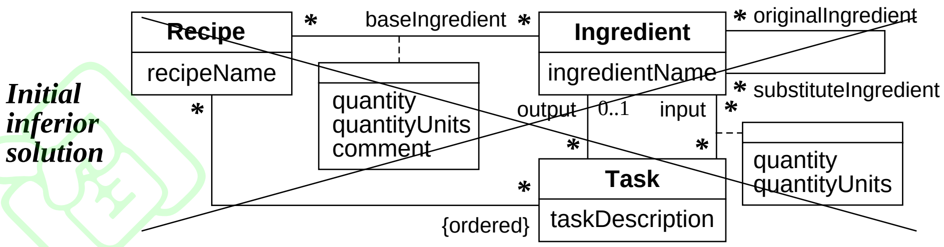


Figure A4.15 Initial class model for recipes with alternate ingredients

4.18 Figure A4.17 shows a class model for the NASAA form. We could have made *Recommendation* an association class, but we promoted it to a class to handle the unusual situation where a broker has multiple suggestions for a security in the same call. Your model could vary a bit from ours, depending on the precise interpretation of the form.

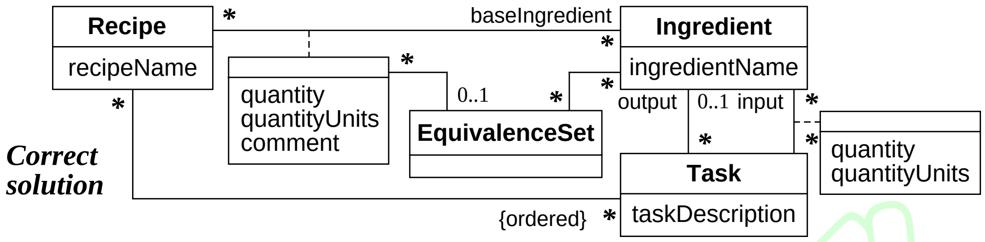


Figure A4.16 Correct class model for recipes with alternate ingredients

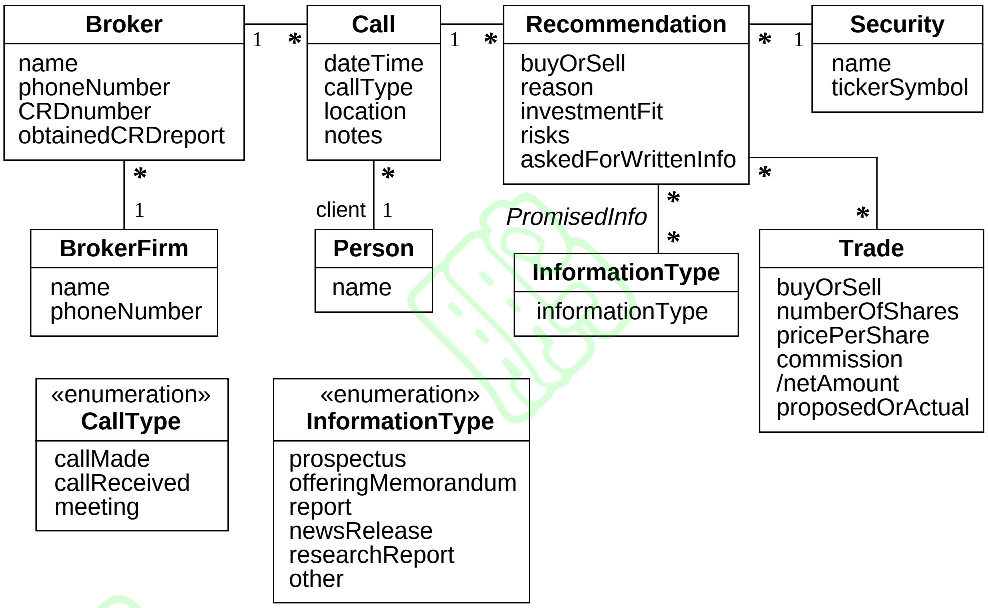


Figure A4.17 Class diagram for NASAA form

4.19 Figure A4.18 shows an inferior answer.

- *meaningType* has values *normal*, *colloquial*, and *slang*.
- *grammarType* has values *noun*, *verb*, *adjective*, *adverb*, and *preposition*.
- The words in a dictionary are alphabetically ordered. The meanings for a word are ordered by frequency of usage.
- The meaning of a word depends on the word and the grammar type.

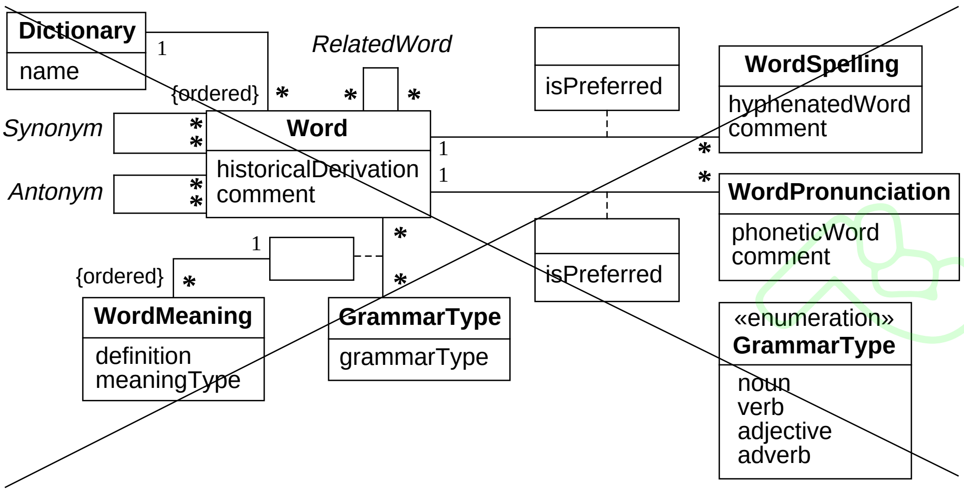


Figure A4.18 An inferior model for a dictionary

The problem with the *RelatedWord*, *Synonym*, and *Antonym* associations is their symmetry. We must store each possible pair of combinations. Figure A4.19 shows a better model.

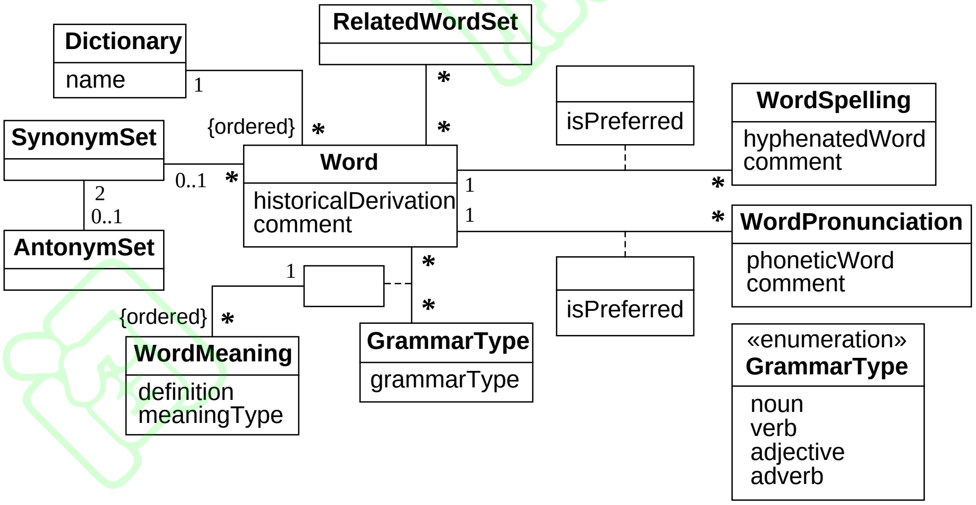


Figure A4.19 A better model for a dictionary

5

State Modeling

5.1 Figure A5.1 shows a state diagram for an extension ladder. You could also include states for the ladder fully extended and fully contracted.

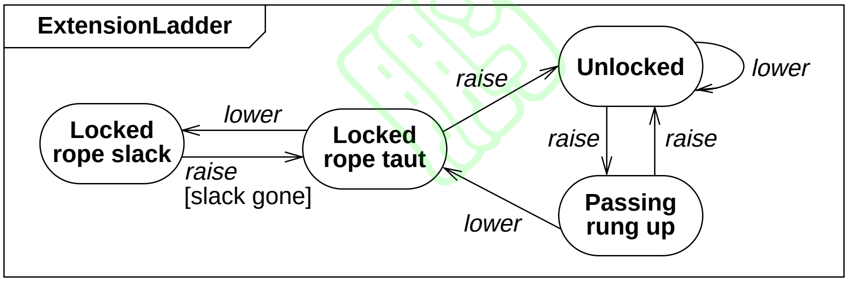


Figure A5.1 State diagram for an extension ladder.

5.2 In Figure A5.2 the event *A* refers to pressing the A button. In this diagram, releasing the button is unimportant and is not shown (although you must obviously release the button before you can press it again). Note that a new button event cannot be generated while any button is pressed. You can consider this a constraint on the input events themselves and need not show it in the state diagram (although it would not be wrong to do so).

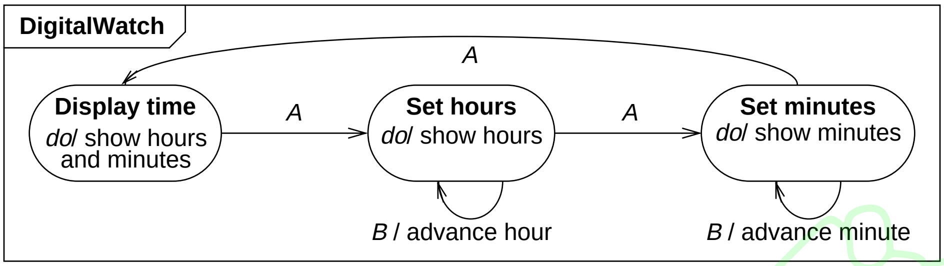


Figure A5.2 State diagram for a simple digital watch

5.3 Figure A5.3 elaborates the state diagram in the exercise.

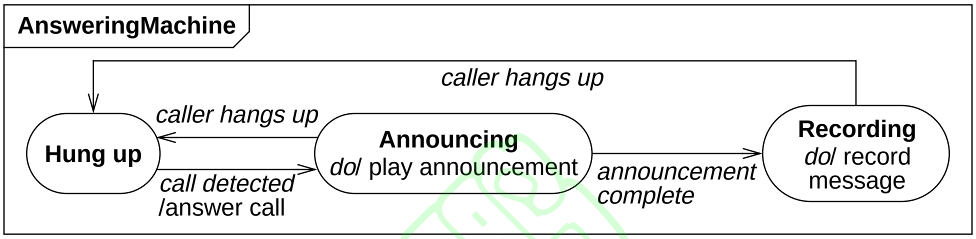


Figure A5.3 State diagram for a telephone answering machine

5.4 Figure A5.4 extends the state diagram from the previous answer to answer after five rings. The number of rings are kept in an internal counter that is reset on each new call and incremented on every ring.

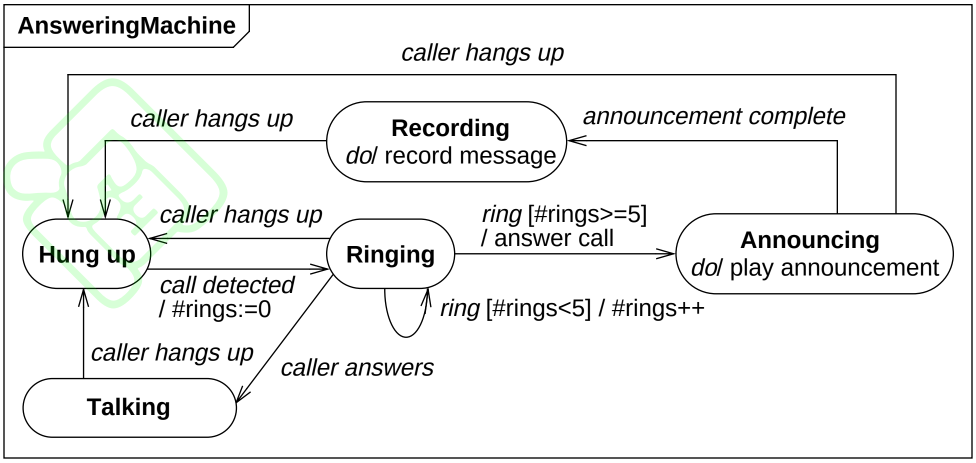


Figure A5.4 State diagram for a machine that answers after five rings

5.5 Figure A5.5 elaborates the state diagram in the exercise.

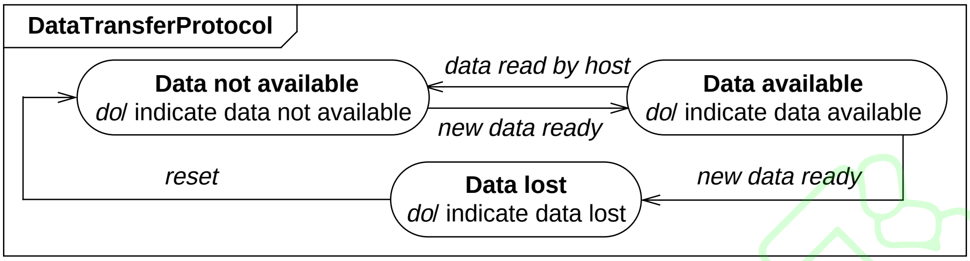


Figure A5.5 State diagram of a data transfer protocol

5.6 Figure A5.6 shows the completed state diagram for the motor control.

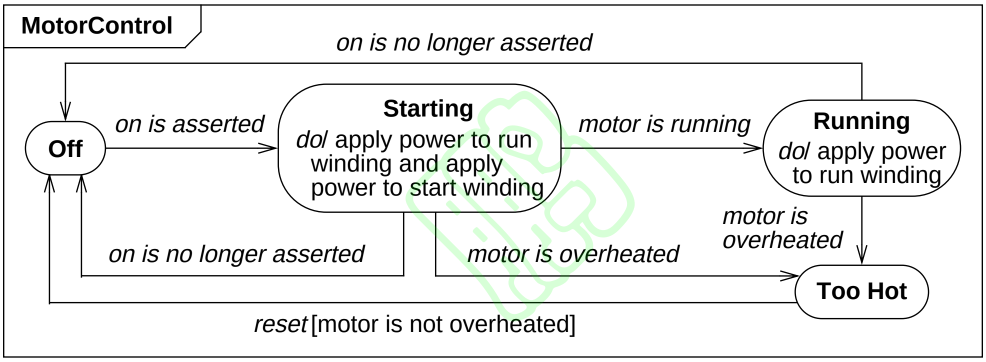


Figure A5.6 State diagram for a motor control

5.7 Figure A5.7 shows another approach for motor control.

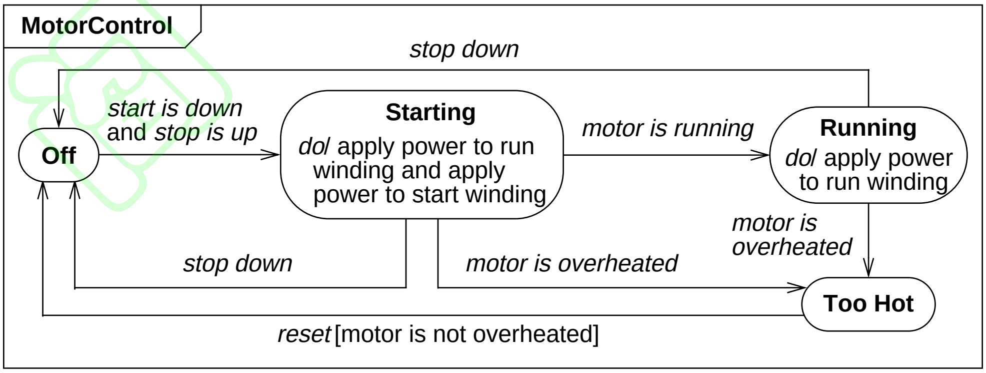

MotorControl
stop down
Off
start is down and stop is up
Starting
do/ apply power to run winding and apply power to start winding
motor is running
Running
do/ apply power to run winding
motor is overheated
stop down
motor is overheated
Too Hot
reset [motor is not overheated]

Figure A5.7 An alternate approach to motor control

5.8 Figure A5.8 shows a state diagram for a diagram editor. This is only a small part of a complete editor. In the real editor there are more ways to pick items and you can pick more than one item (next exercise).

Some events can be ignored. In state *Nothing selected* event left down with cursor on no object has no effect and therefore is not shown. Similarly, left drag (that is moving the cursor while the left button is down) has no effect in the *Nothing selected* state.

In the real editor, the right button is used to pop up a menu dependent on the selection state. There would be many selection substates, one for each kind of item (or combination of items) that can be selected. Similarly, there would be transitions to create new items.

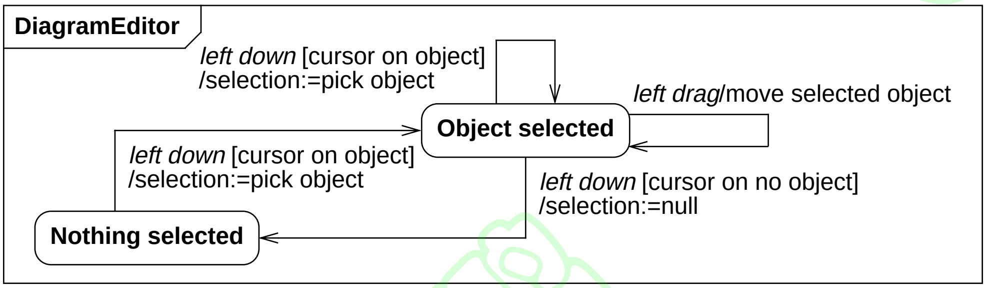

DiagramEditor
left down [cursor on object] /selection:=pick object
Object selected
left drag/move selected object
left down [cursor on object] /selection:=pick object
left down [cursor on no object] /selection:=null
Nothing selected

Figure A5.8 State diagram for a diagram editor

5.9 Figure A5.9 extends the diagram editor for selecting multiple objects.

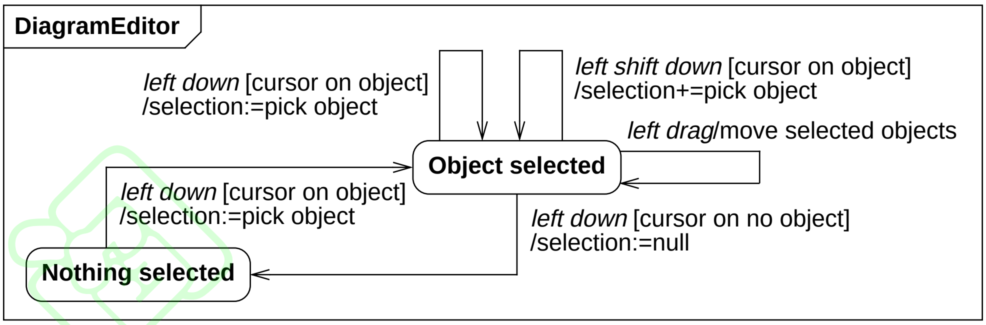

DiagramEditor
left down [cursor on object] /selection:=pick object
left shift down [cursor on object] /selection+=pick object
left drag/move selected objects
Object selected
left down [cursor on object] /selection:=pick object
left down [cursor on no object] /selection:=null
Nothing selected

Figure A5.9 State diagram for a diagram editor with selection of multiple objects

5.10 Figure A5.10 extends Figure E5.4 to represent the observed jamming behavior properly.

5.11 Figure A5.11 shows a state diagram. Note that even simple state diagrams can lead to complex behavior. A *change* event occurs whenever the candle is taken out of its holder

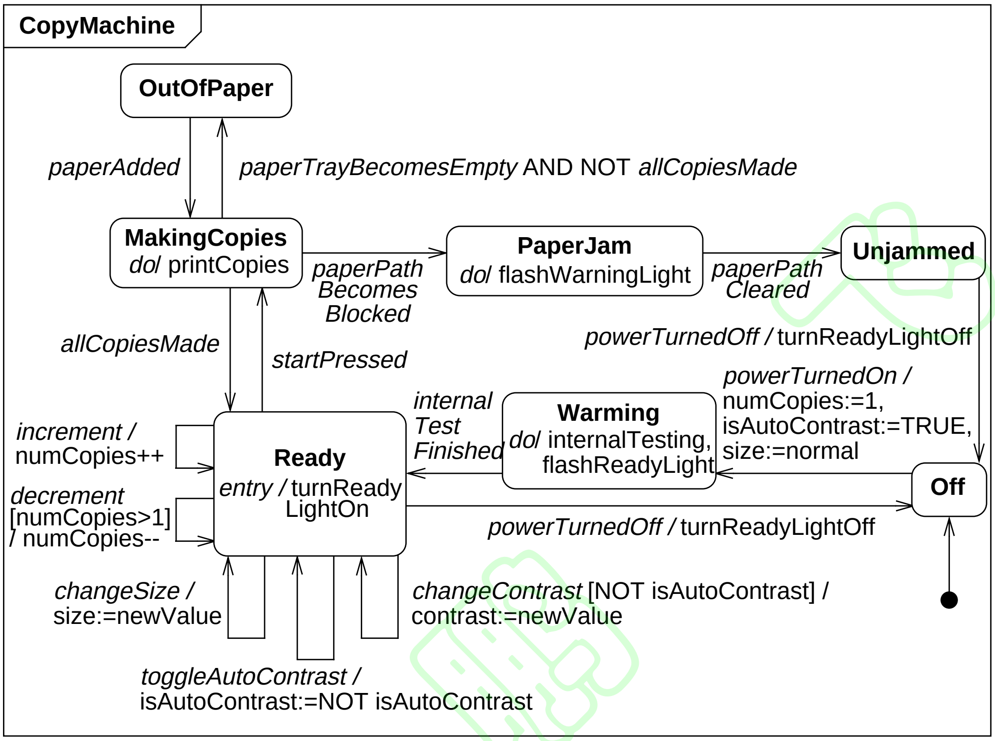

CopyMachine
OutOfPaper
paperAdded
paperTrayBecomesEmpty AND NOT allCopiesMade
MakingCopies
do/ printCopies
paperPath Becomes Blocked
PaperJam
do/ flashWarningLight
paperPath Cleared
Unjammed
powerTurnedOff / turnReadyLightOff
allCopiesMade
startPressed
powerTurnedOn / numCopies:=1, isAutoContrast:=TRUE, size:=normal
internal Test Finished
Warming
do/ internalTesting, flashReadyLight
increment / numCopies++
Ready
entry / turnReady LightOn
decrement [numCopies>1] / numCopies--
Off
powerTurnedOff / turnReadyLightOff
changeSize / size:=newValue
changeContrast [NOT isAutoContrast] / contrast:=newValue
toggleAutoContrast / isAutoContrast:=NOT isAutoContrast

Figure A5.10 State diagram for copy machine with observed behavior for jamming

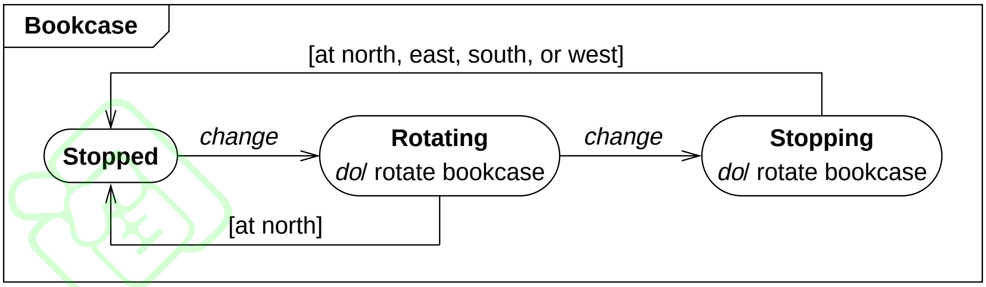


Figure A5.11 State diagram for bookcase control

or whenever it is put back. The condition *at north* is satisfied whenever the bookcase is behind the wall. The condition *at north, east, south, or west* is satisfied whenever the bookcase is facing front, back, or to the side.

When you first discovered the bookcase, it was in the *Stopped* state pointing south. When your friend removed the candle, a *change* event drove the bookcase into the *Ro-*

tating state. When the bookcase was pointing north, the condition *at north* put the bookcase back into the *Stopped* state. When your friend reinserted the candle, another *change* event put the bookcase into the *Rotating* state until it again pointed north. Pulling the candle out generated another *change* event and would have caused the bookcase to rotate a full turn if you had not blocked it with your body. Forcing the bookcase back is outside the scope of the control and does not have to be explained.

When you put the candle back again another *change* event was generated, putting the bookcase into the *Rotating* state once again. Taking the candle back out resulted in yet another *change* event, putting the bookcase into the *Stopping* state. After 1/4 turn, the condition *at north, east, south or west* was satisfied, putting the bookcase into the *Stopped* state.

What you should have done at first to gain entry was to take the candle out and quickly put it back before the bookcase completed 1/4 turn.

6

Advanced State Modeling

6.1 The headlight (Figure A6.1) and wheels (Figure A6.2) each have their own state diagram. Note that the stationary state for a wheel includes several substates.

We have shown default initial states for the headlight and wheels. The actual initial state of the wheels may be arbitrary and could be any one of the power off states. The system operates in a loop and does not depend on the initial state, so you need not specify it. Many hardware systems have indeterminate initial states.

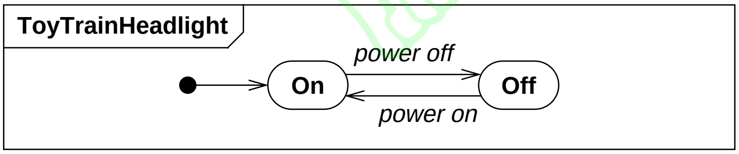


Figure A6.1 State diagram for a toy train headlight

6.2 Figure A6.3 extends Figure A5.2 for rapid setting of time. *B* means press button B while $\overline{B}$ means to release it. $B\overline{B}$ means to press and release it. (Instructor's note: you may want to give the students a copy of our answer to Exercise 5.2.)

6.3 Figure A6.4 adds *Motor On* to capture the commonality of the starting and running state. We have shown a transition from the *Off* state to the *Starting* state. We could instead have shown a transition from *Off* to *Motor On* and made *Starting* the initial state of *Motor On*. Note that the activity *apply power to run winding* has been factored out of both starting and running states. (Instructor's note: you may want to give the students a copy of our answer to Exercise 5.6.)

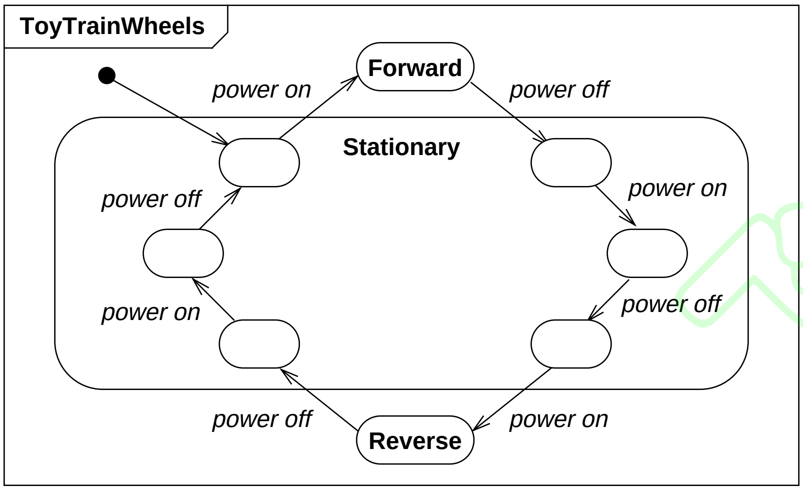


Figure A6.2 State diagram for the wheels of a toy train

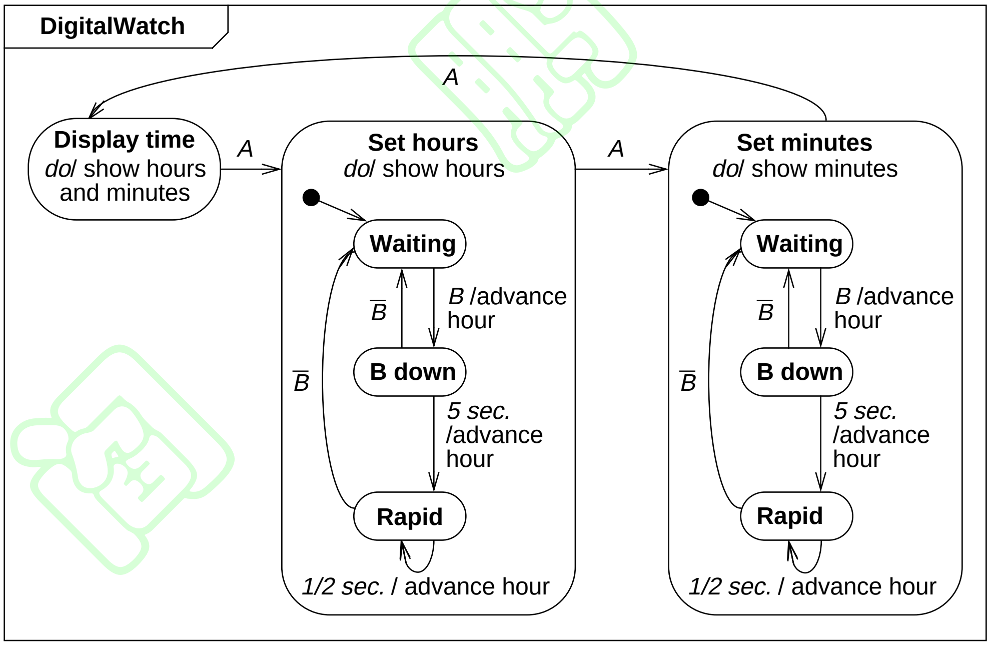


Figure A6.3 Extended state diagram for a digital watch

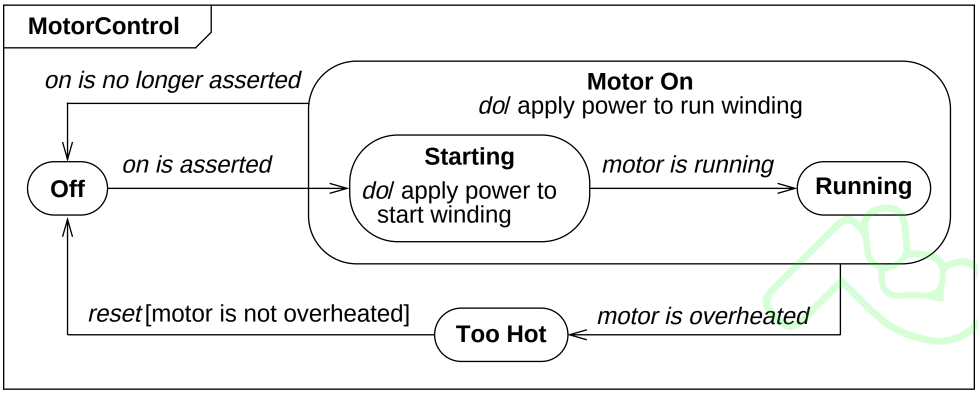


Figure A6.4 State diagram for a motor control using nested states

6.4 Figure A6.5 revises the motor state diagram. Note that a transition from *Off* to either *Forward* or *Reverse* also causes an implicit transition to *Starting*, the default initial state of the lower concurrent subdiagram. An off request causes a transition out of both concurrent subdiagrams back to state *Off*.

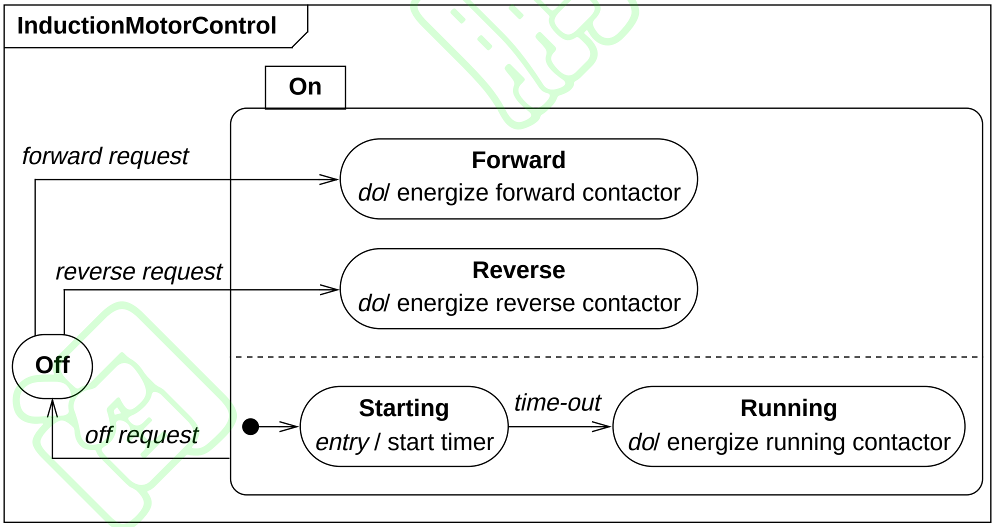


Figure A6.5 Revised state diagram for an induction motor control

6.5 a. Add an arrow labeled "[overheating detected]" from each non-*Off* state to state *Off*.

b. Add an arrow labeled "[overheating detected]" from state *On* to state *Off*. Because state *On* has nested states, the transition applies to them.

6.6 Figure A6.6 places the signal classes into a generalization hierarchy along with a few sample signal attributes. Do not confuse *textPick* with *characterInput*. Often *character-Input* will follow a *textPick* event that indicates which input object receives the input character.

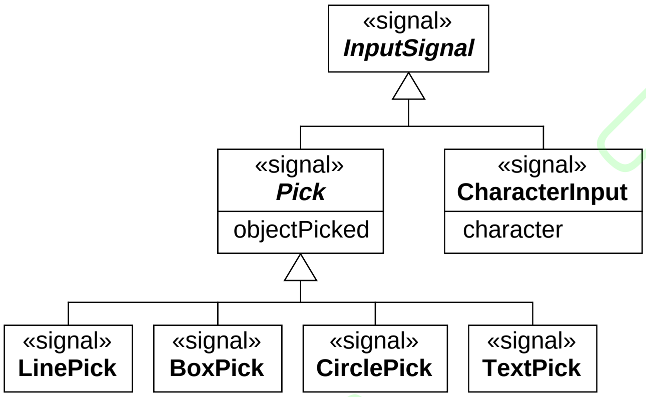


Figure A6.6 A signal hierarchy

6.7 There would be one concurrent state machine for each room, plus one each for the furnace control, blower control, and humidity control. They would have the following responsibilities:

Room control. Measures temperature in the room. Requests heat if temperature is too low. Cancels heat request if temperature is high enough. Controls flappers to the room based on room temperature. Allows user to set target temperature.

Furnace control. Turns on furnace if any room requests heat. Turns off furnace if no room requests heat. Turns off furnace if furnace temperature is too high.

Blower control. Turns on blower when furnace goes on. Turns off blower if furnace is off and furnace is cool enough.

Humidity control. Measures humidity and outside temperature. Allows user to set target humidity. Turns on humidifier if humidity is too low and blower is on.

6.8 a. Events are:

selectpush select button
on-offpush on-off button
timedpush timed button
autopush auto button
setpush set button
vcrpush vcr button
time-outpreset recording time has expired

Activities are:

display timedisplay current time of day in time display
display start time display start recording time in time display
display stop time display stop recording time in time display
flash dayflash day segment of time display
flash hourflash hour segment of time display
flash minutesflash minutes segment of time display
flash channelflash channel display
initialize start time set time setting display for start time to be current time
initialize stop time set time setting display for stop time to be start time
next dayadvance time setting display to next day of the week
next houradvance time setting display to next hour
next minuteadvance time setting display to next minute
next channeladvance channel setting display to next preset channel
more timeadd fixed time increment to preset recording time
recordrecord the preset channel on the vcr tape
update timechange the time display as the current time changes
display “auto” light the “auto” indicator on the display panel
vcr outputenable the vcr as the source of output (and disable the tv)
tv outputenable the tv as the source of output (and disable the vcr)

b. A synopsis of the user manual is as follows:

To set the clock, push SELECT successively to set the day of the week, the hour, and the minutes in turn. Within each setting, push SET to advance the respective setting by 1 unit (15 minutes for minutes).

To set the record timer, push ON/OFF. The time to start recording is displayed. It is initially set to the current time. The start time may be set using the SELECT and SET buttons as described for setting the clock to set the day, hour, minutes, and channel to begin recording.

When the recording start time and channel have been selected, press ON/OFF again to set the recording stop time. The stop time is displayed. It is initially set to be the same as the start time. Press SELECT and SET to set the hours and minutes of the stop time as described for setting the clock.

When the stop time has been set, press ON/OFF a final time to return to the time-of-day display. To enable the automatic recording mode, press AUTO. The “auto” indicator lights up. When the preset recording start time is reached, the vcr automatically switches to the preset channel and begins recording. When the preset recording stop time is reached, the vcr automatically stops recording. The current time is displayed continuously in automatic mode. To disable automatic recording mode, press AUTO again; if the vcr is recording, then recording will cease.

To record for 15 minutes starting immediately, press TIMED. The vcr begins recording on the current channel. To extend the recording time by an additional 15 minutes, press TIMED one or more times. [Note that there appears to be no way to cancel timed recording before time expires. This is a bug in the state diagram and would be extremely annoying to vcr owners.]

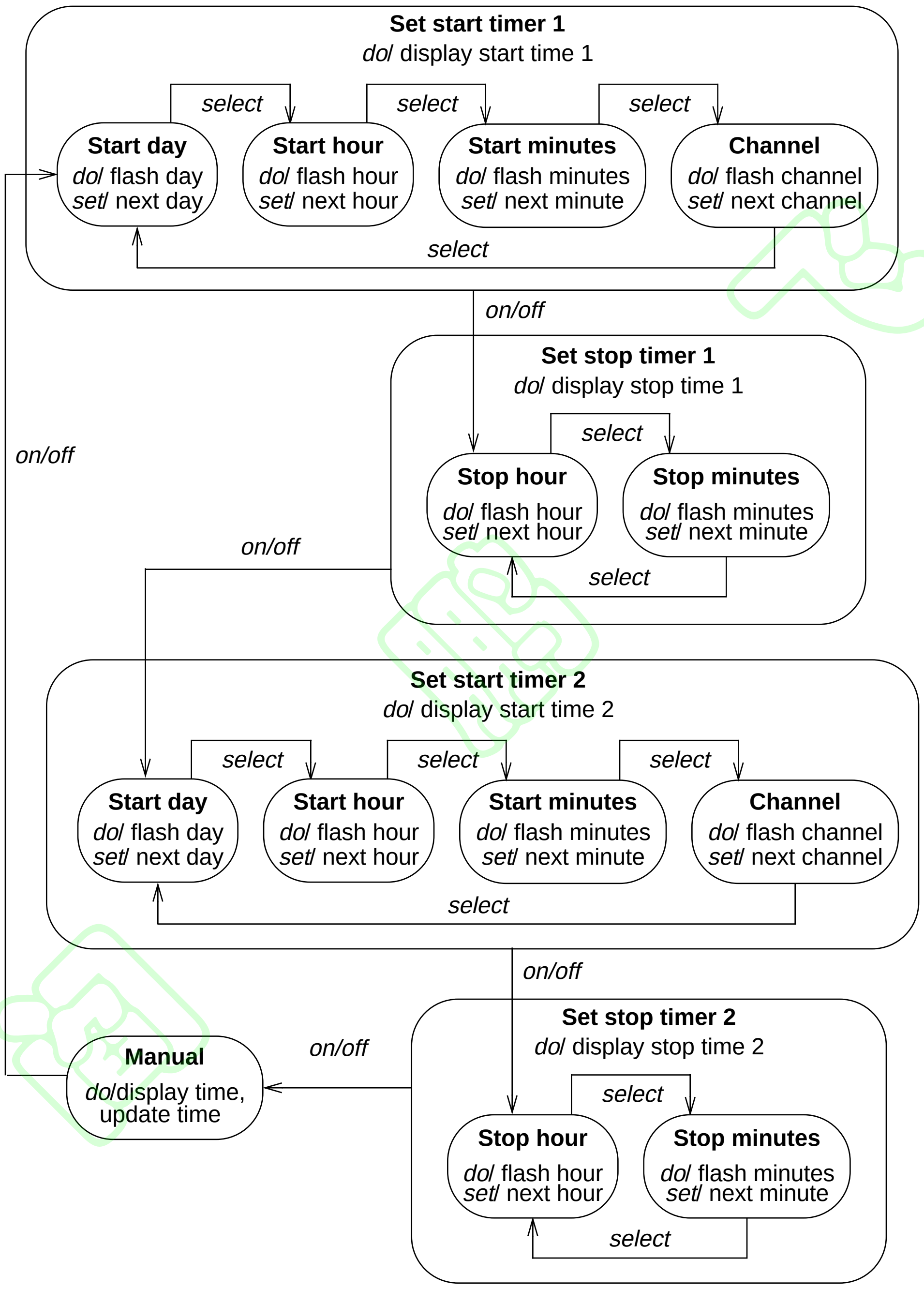


Figure A6.7 Adding a second start-stop timer for a vcr

Press the VCR button to toggle the output of the vcr between straight antenna input (to use the tuner on the TV set) and the output of the vcr (TV must be set on channel 3 or 4).

[The user's manual would also include descriptions of the manual control buttons omitted from the state diagram. The state diagram also omits controls for setting the current channel.]

c. We added to the state diagram (Figure A6.7) by duplicating states *Set start timer* and *Set stop timer* for the second time and hooking the new states into the ON/OFF sequence. We have deleted the entry activity to initialize the start time and stop time, because the user would not want to lose a previously set time in using the second setting.

d. We could parameterize the states to avoid duplicating information in several places (Figure A6.8). As Section 6.1 explains, we would use event parameters and define submachines for states *Set start timer* and *Set stop timer*.

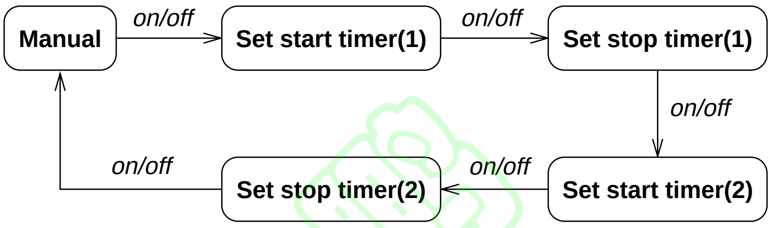


Figure A6.8 Using submachines to reduce the size of the vcr state diagram

6.9 Figure A6.9 uses nested states to repair the flaw in the state diagram for a copy machine. The power can be turned off at any time for the copy machine causing a transition to the off state.

6.10 Figure A6.10 shows the state diagram for *TableTennisGame*. This is the only class with important temporal behavior. Consequently this one state diagram constitutes the entire state model.

6.11 Figure A6.11 reifies class modeling concepts. An event within a state (entry event, exit event, or some other event) may or may not have an associated activity.

6.12 Figure A6.12 flattens the transmission state diagram. There are a few more transitions than for the nested state diagram. The increase in size would be more dramatic for a complex nested state diagram.

Note that we omit a self-transition from *First* to *First* upon *stop*. Such a behavior is included in Figure 6.5 and is a minor flaw of the diagram. The distinction between remaining in *First* and making a self-transition to *First* would be meaningful if activities were bound to the *stop* transition.

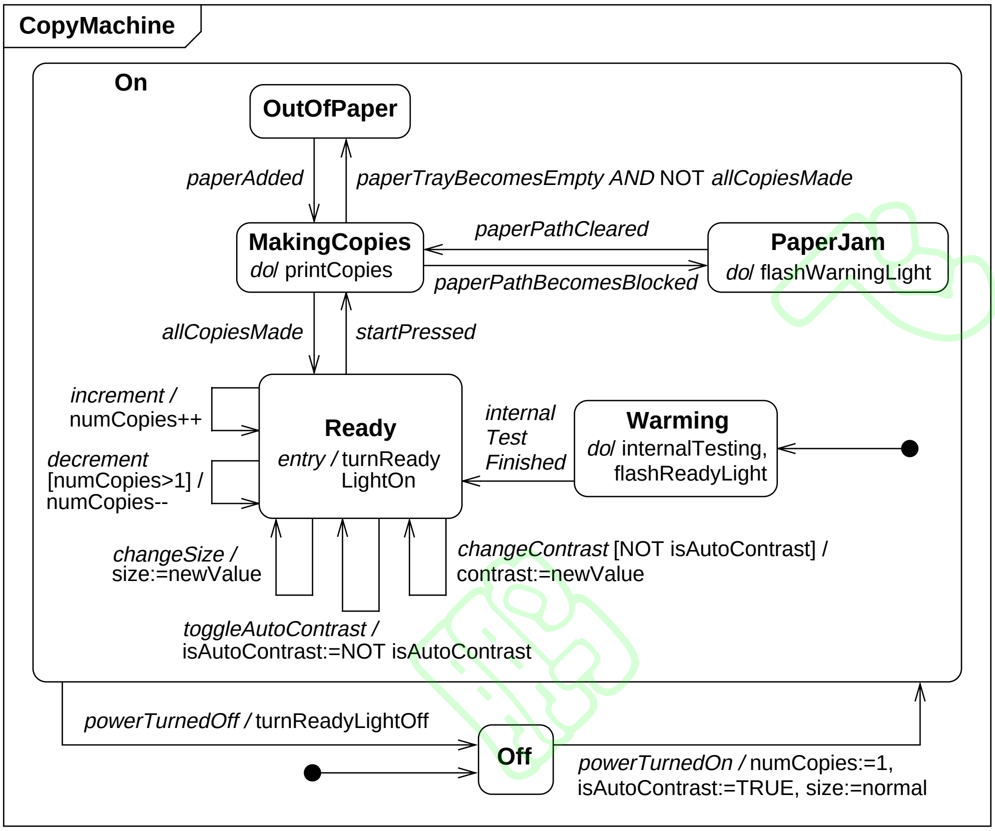

CopyMachine
On
OutOfPaper
paperAdded
paperTrayBecomesEmpty AND NOT allCopiesMade
MakingCopies
do/ printCopies
paperPathCleared
paperPathBecomesBlocked
PaperJam
do/ flashWarningLight
allCopiesMade
startPressed
increment / numCopies++
decrement [numCopies>1] / numCopies--
Ready
entry / turnReady LightOn
internal Test Finished
Warming
do/ internalTesting, flashReadyLight
changeSize / size:=newValue
changeContrast [NOT isAutoContrast] / contrast:=newValue
toggleAutoContrast / isAutoContrast:=NOT isAutoContrast
powerTurnedOff / turnReadyLightOff
Off
powerTurnedOn / numCopies:=1, isAutoContrast:=TRUE, size:=normal

Figure A6.9 State diagram for copy machine with nested states

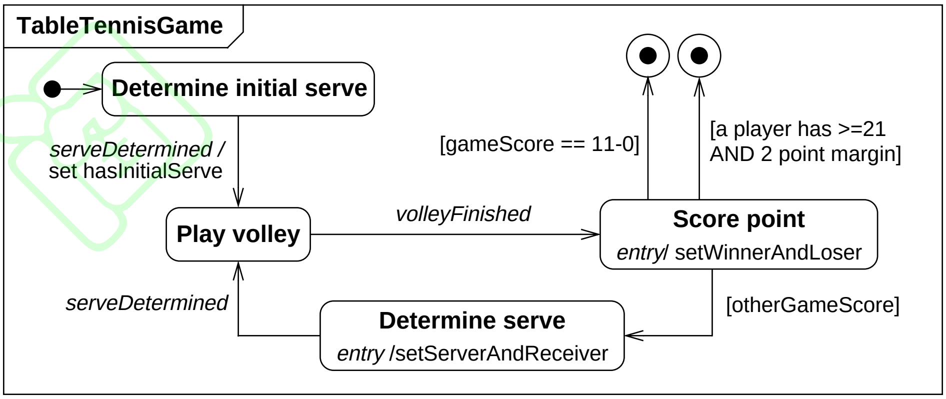


Figure A6.10 State diagram for *TableTennisGame*

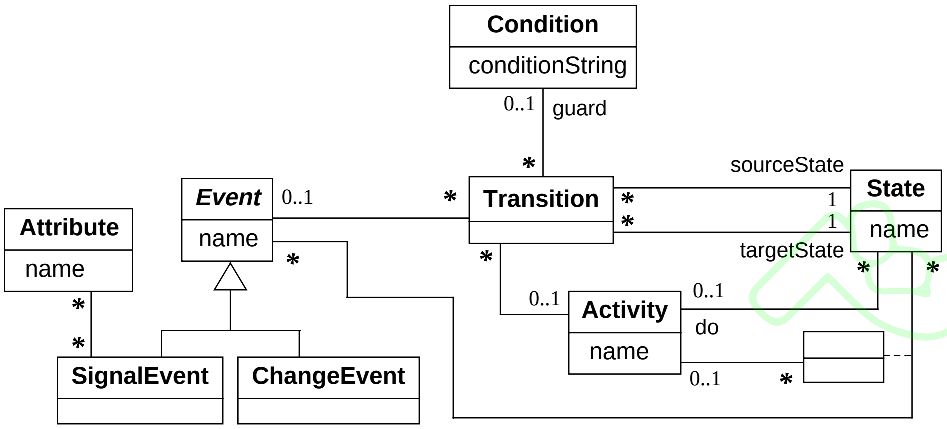


Figure A6.11 Class diagram that reifies state modeling concepts

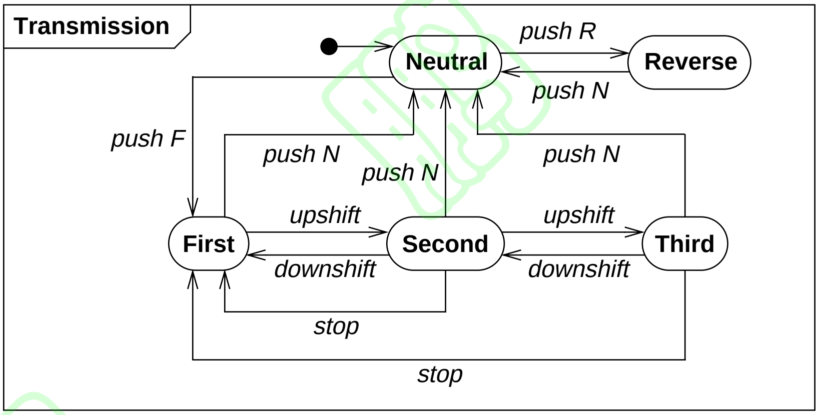


Figure A6.12 Flat state diagram for a car transmission

7

Interaction Modeling

7.1 Here are answers for a physical bookstore.

a. Some actors are:

- **Customer**. A person who initiates the purchase of an item.
- **Cashier**. An employee who is authorized to check out purchases at a cash register.
- **Payment verifier**. The remote system that approves use of a credit or debit card.

b. Some use cases are:

- **Purchase items**. A customer brings one or more items to the checkout register and pays for the items.
- **Return items**. The customer brings back items that were previously purchased and gets a refund.

c. Figure A7.1 shows a use case diagram.

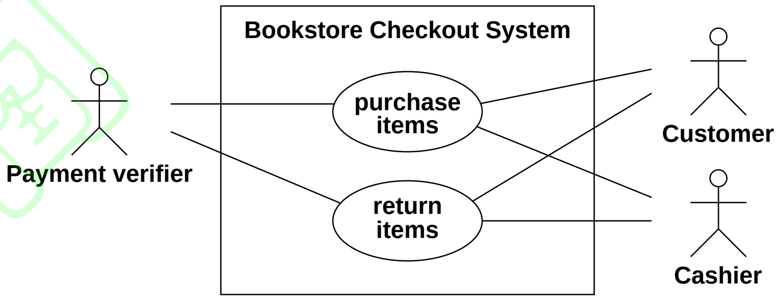


Figure A7.1 Use case diagram for a physical bookstore checkout system

d. Here is a normal scenario for each use case. There are many possible answers. [Instructor's note: You may wish to give the students the answers to Exercise 7.1 parts a-c.]

- **Purchase items**.
  Customer brings items to the counter.
  Cashier scans each customer item.
  Cashier totals order, including tax.
  Cashier requests form of payment.
  Customer gives a credit card.
  Cashier scans card.
  Checkout system communicates scan data to verifier.
  Verifier reports that credit card payment is acceptable.
  Customer signs credit card slip.
- **Return items**.
  Customer brings purchased item to the counter.
  Customer has receipt from earlier purchase.
  Cashier notes that payment was in cash.
  Cashier accepts items and gives customer a cash refund.

e. Here is an exception scenario for each use case. There are many possible answers. [Instructor's note: You may wish to give the students the answers to Exercise 7.1 parts a-c.]

- **Purchase items**.
  Customer brings items to the counter.
  Cashier scans each customer item.
  An item misscans and cashier goes to item display to get the item price.
- **Return items**.
  Customer brings purchased item to the counter.
  Customer has no receipt from earlier purchase.
  Customer is given a credit slip, but no refund.

f. Figure A7.2 shows a sequence diagram for the first scenario in (d). Figure A7.3 shows a sequence diagram for the second scenario in (d). [Instructor's note: You may wish to give the students the answers to Exercise 7.1 parts a-e.]

7.2 Here are answers for a computer email system.

a. Some actors are:

- **User**. A person who is the focus of services.
- **Server**. A computer that communicates with the email system and is the intermediate source and destination of email messages.
- **Virus checker**. Software that protects against damage by a malicious user.

b. Some use cases are:

- **Get email**. Retrieve email from the server and display it for the user.

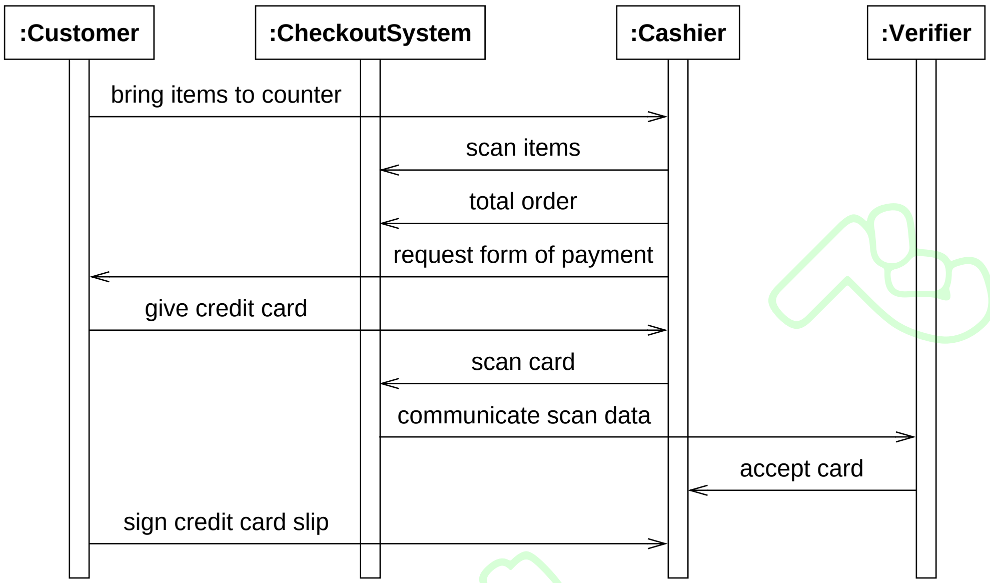


Figure A7.2 Sequence diagram for a purchase of items

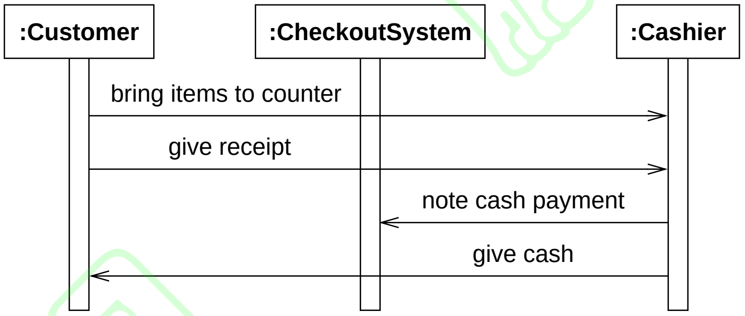


Figure A7.3 Sequence diagram for a return of items

- **Send email**. Take a message that was composed by the user and send it to the server.
- **Manage email**. Perform tasks such as deleting email, managing folders, and moving email between folders.
- **Set options**. Do miscellaneous things to tailor preferences to the user's liking.
- **Maintain address list**. Keep a list of email addresses that are important to the user.

c. Figure A7.4 shows a use case diagram. The diagram assumes that mail is moved from the server to the email system as the user reads it. If mail is kept on the server, then the server would also be involved in the *manage email* and *maintain address list* use cases.

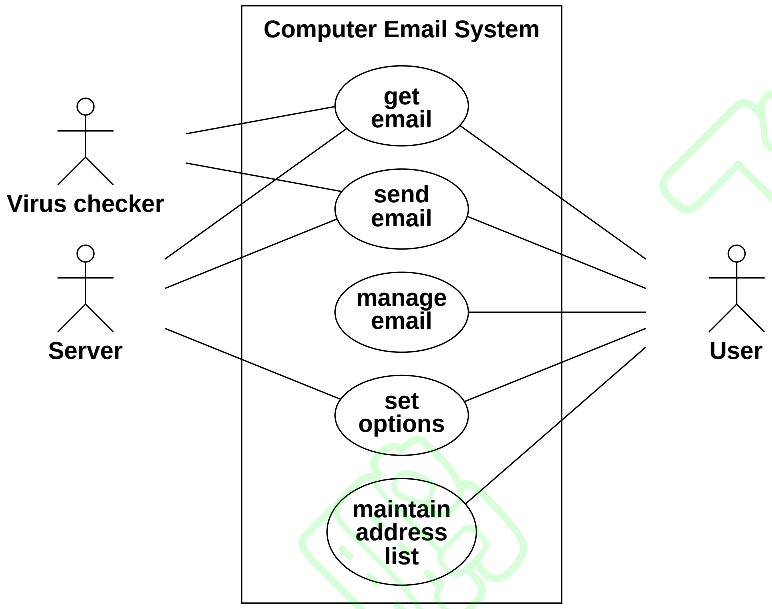


Figure A7.4 Use case diagram for a computer email system

d. Here is a normal scenario for each use case. There are many possible answers. [Instructor's note: You may wish to give the students the answers to Exercise 7.2 parts a-c.]

- **Get email**.
  User logs in to email system.
  System displays the mail in the *Inbox* folder.
  User requests that system get new email.
  System requests new email from server.
  Server returns new email to system.
  System displays the mail in the *Inbox* folder and highlights unread messages.
- **Send email**.
  User composes an email message and requests that system sends it.
  Local computer sends email to server.
  Server acknowledges receipt of email to send.
- **Manage email**.
  User logs in to email system.
  User clicks on *Sent* folder.
  User deletes all email in *Sent* folder.

- **Set options**.
  User logs in to email system.
  User requests change of password.
  Local computer displays password change form.
  User enters new password.
  User reenters new password.
  Local computer sends new password to server.
  Server accepts new password.
- **Maintain address list**.
  User clicks on *Inbox* folder.
  User selects a message.
  User adds sender to address list.

e. Here is an exception scenario for each use case. There are many possible answers. [Instructor's note: You may wish to give the students the answers to Exercise 7.2 parts a-c.]

- **Get email**.
  User logs in to email system.
  Local computer sends password to server.
  Server rejects password as incorrect.
- **Send email**.
  User composes an email message.
  User requests that system sends email.
  System detects loss of Internet connection.
- **Manage email**.
  User logs in to email system.
  User clicks on *Sent* folder.
  System reports that data in folder is corrupted.
- **Set options**.
  User logs in to email system.
  User requests change of password.
  Local computer displays password change form.
  User enters new password.
  Password length is illegal and password is not accepted.
- **Maintain address list**.
  User clicks on *Inbox* folder.
  User selects a message.
  User tries to add sender to address list.
  System reports that sender is already in address list.

f. Figure A7.5 through Figure A7.9 show the sequence diagrams for the scenarios in (d). [Instructor's note: You may wish to give the students the answers to Exercise 7.2 parts a-e.]

7.3 Here are answers for an online airline reservation system.

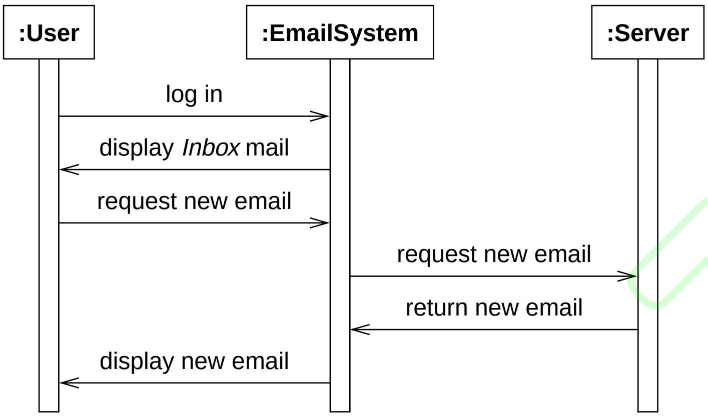


Figure A7.5 Sequence diagram for getting email

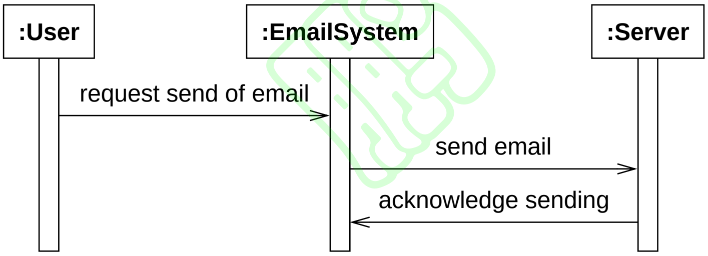


Figure A7.6 Sequence diagram for sending email

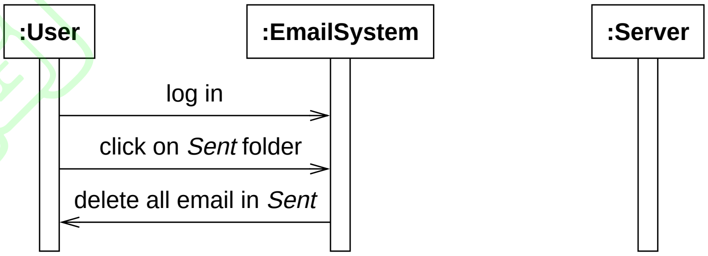


Figure A7.7 Sequence diagram for managing email

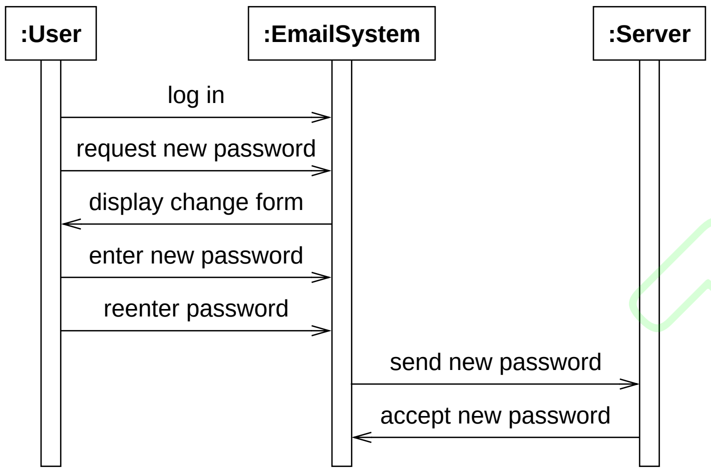


Figure A7.8 Sequence diagram for setting options

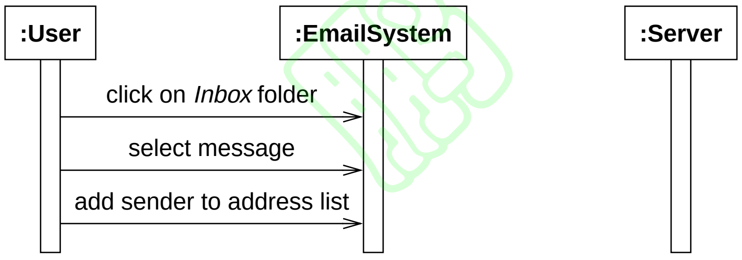


Figure A7.9 Sequence diagram for maintaining address list

a. Some actors are:

- **User**. The person who is seeking service from the airline.
- **Airline**. A company that offers scheduled flights available for purchase.

b. Some use cases are:

- **Make reservation**. Book and pay for an airline flight and possibly choose a seat.
- **Check availability**. Given data (such as origin, destination, date, airline preference), find the available flights and prices.
- **Check flight status**. Report the status of a flight that is underway (on-time, delayed, missing information, ...).
- **Handle benefits**. Display the current frequent traveler status for a person, register a new frequent traveler, book frequent traveler flights, show available flights, ...

- **Send an inquiry**. Let a user submit email to report flight dissatisfaction, frequent traveler benefit problem, or communicate some miscellaneous message to an airline.

c. Figure A7.10 shows a use case diagram.

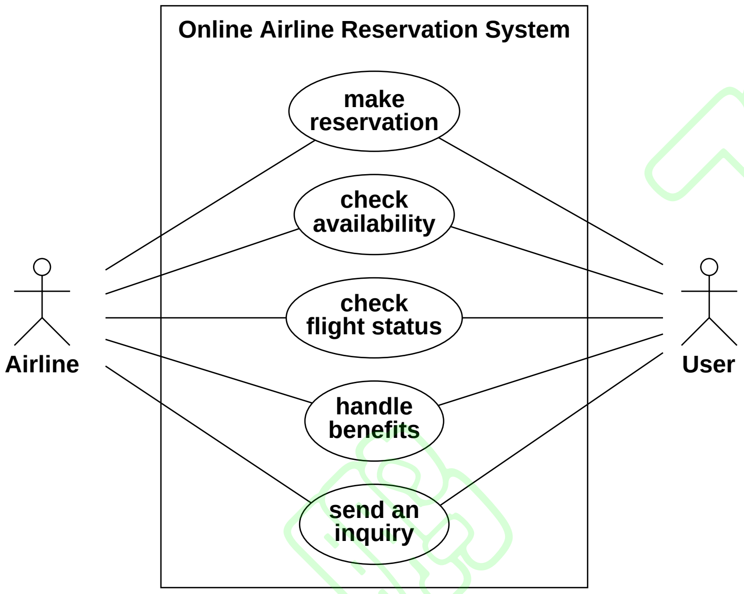


Figure A7.10 Use case diagram for an online airline reservation system

7.4 Here are answers for a library checkout system.

a. Some actors are:

- **Patron**. Someone who is the beneficiary of library services.
- **Librarian**. A library employee who handles most interaction with patrons.
- **Library**. A facility that mediates the borrowing of copies of library items. We assume that the library checkout system spans multiple libraries and that items can be exchanged between libraries.
- **Library item**. A book, magazine, movie, compact disc, or audio tape that a library offers for borrowing.

b. Some use cases are:

- **Borrow library item**. Remove an item from the library premises and promise to return it within the specified time.
- **Return library item**. Bring an item back to the library after borrowing it, and possibly pay a fine if it is late.

- **Inquire about library item**. Get data about library items that are of interest.
- **Reserve library item**. Secure placement on a waiting list for a library item that is currently checked out by another user.

c. Figure A7.11 shows a use case diagram. We assume inquiries and reservations can be handled without the intervention of a librarian.

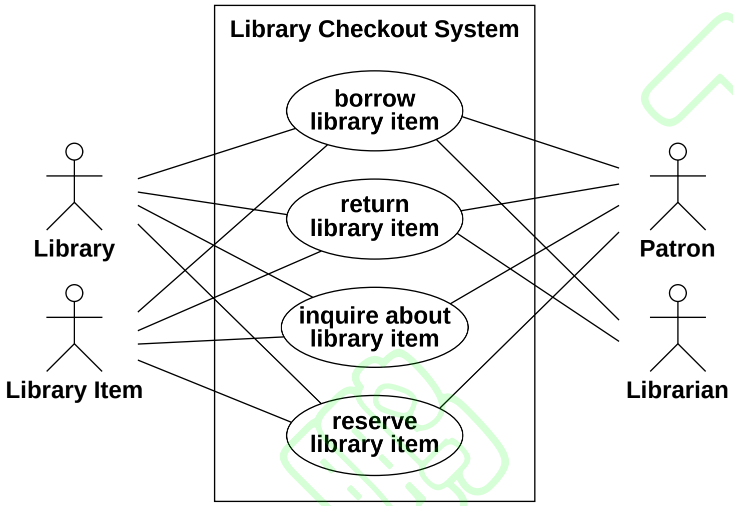


Figure A7.11 Use case diagram for a library checkout system

7.5 Here are some use cases for the Windows Explorer.

- **Create new folder**. Create a new folder as a child of the current folder.
- **Create new shortcut**. Create a new shortcut and place it in the current folder.
- **Select file**. Choose a file for subsequent activity and highlight the file icon.
- **Deselect file**. Deactivate a file for subsequent activity and unhighlight the file icon.
- **Copy file**. Copy the file to the file buffer.
- **Cut file**. Move the file to the file buffer. Shade the file icon to indicate cutting.
- **Paste file**. Move the file from the file buffer to the new folder. If the file was being cut, remove the file icon from the original folder.
- **Delete file**. Remove the file from file storage.
- **Open file**. Launch the application associated with the file.
- **View file properties**. Show the file name, creator, last update, and other incidental data.

- **Expand folder**. Show the files in the folder and make the folder the focus of subsequent actions.
- **Go up to parent folder**. Show the files in the parent folder and make the parent folder the focus of subsequent actions.

7.6 Here are the scenarios.

a. Assume that everything starts out on the east side and is to be moved to the west side. A scenario in which nothing gets eaten:

(Farmer, fox, goose, corn all on W.)
Farmer takes goose to E.
Farmer returns alone to W.
Farmer takes fox to E.
Farmer takes goose to W.
Farmer takes corn to E.
Farmer returns alone to W.
Farmer takes goose to E.
(Farmer, fox, goose, corn all on E.)

A scenario in which something gets eaten:

(Farmer, fox, goose, corn all on W.)
Farmer takes goose to E.
Farmer returns alone to W.
Farmer takes corn to E.
Farmer returns alone to W.
Goose eats corn.
Farmer takes fox to E.
(Farmer, fox, goose on E. Corn is gone.)

b. There is ambiguity about what to include and how much detail to show. This is common for specifications. One possible scenario:

Open door.
Get in car and sit down.
Close door.
Put key in ignition.
Put on seat belt.
Check that transmission is in park.
Depress and release accelerator pedal.
Turn key.
Engine starts.
Release key.
Depress brake pedal.
Release emergency brake.
Move transmission lever to drive.
Check for traffic in rear view mirror.

Turn on left directional light.
Move foot to accelerator pedal.
Depress pedal slowly and begin to drive.

c. Again there is considerable ambiguity about what to include.
Press "up" button.
Button lights up.
Bell sounds and "up" indicator lights.
Elevator door opens.
Get in elevator.
Press button for floor 6.
Button lights up.
Elevator door closes.
Elevator moves up.
Elevator stops at floor 3.
Elevator door opens.
Two passengers get on.
Elevator door closes.
Elevator moves up.
Elevator stops at floor 6.
Elevator door opens.
You and the other two passengers get out.
Note that specific incidents are properly part of a scenario.

d. This is similar to the previous part. Conditions are shown in parentheses:
(Traveling on the highway in drive.)
Accelerate to 90 km/hr.
Press "set" button on cruise control.
Cruise control engages control of accelerator.
Take foot off the accelerator pedal.
(Car operates accelerator under cruise control.)
Encounter slow moving car that you want to pass.
Depress accelerator and pass car.
Remove foot from accelerator.
(Cruise control continues to maintain speed.)
Encounter slow moving traffic that you cannot pass.
Depress brake pedal.
Cruise control disengages control of accelerator.
Maneuver past slow moving traffic.
Press "resume" button on cruise control.
Cruise control re-engages control of accelerator.
Take foot off accelerator.
Cruise control accelerates to preset speed.
(Car is operating under cruise control)

7.7 One of many possible scenarios:

Get into tub.
Turn on cold and hot water.
Water flows from faucet.
Adjust temperature.
Pull shower diverter.
Warm water flows from shower head.
Phone rings.
Turn off hot and cold water.
Water stops flowing.
Get out of tub.
Answer phone.
Talk.
Hang up phone.
Get into tub.
Turn on cold and hot water.
Push shower diverter.
Water flows from faucet.
Adjust temperature.
Pull shower diverter.
Warm water flows from shower head.
Wash yourself.
Turn off hot and cold water.
Water stops flowing.
Get out of tub.

7.8 Figure A7.12 shows an activity diagram for computing a restaurant bill.

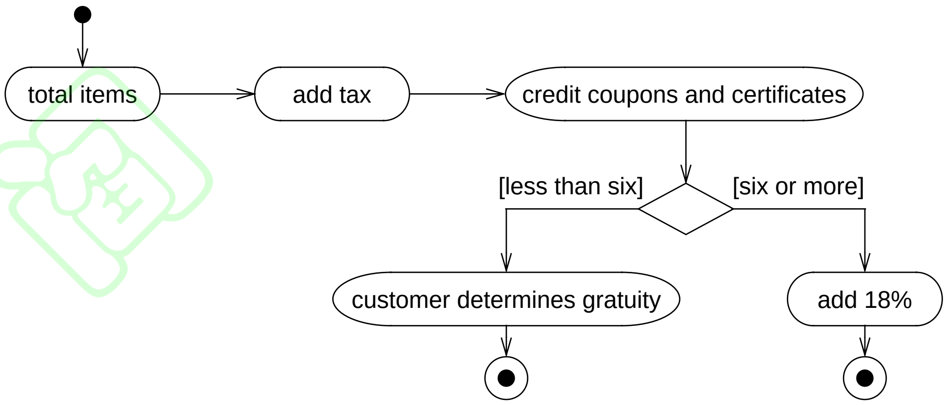


Figure A7.12 Activity diagram for computing a restaurant bill

7.9 Figure A7.13 shows an activity diagram for awarding frequent flyer credits.

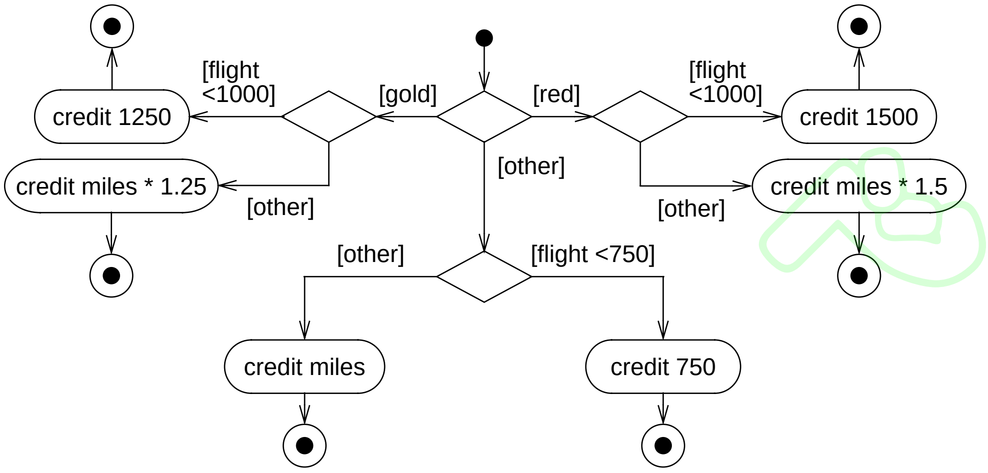


Figure A7.13 Activity diagram for awarding frequent flyer credits

7.10 Figure A7.14 shows an activity diagram that elaborates the details of logging into an email system.

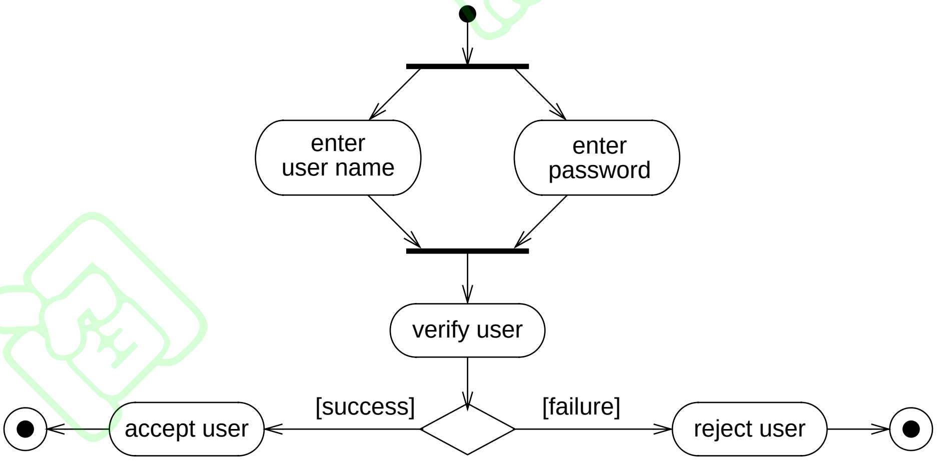


Figure A7.14 Activity diagram for logging into an email system

8

Advanced Interaction Modeling

8.1 Here are answers for an electronic gasoline pump.

a. Figure A8.1 shows a use case diagram.

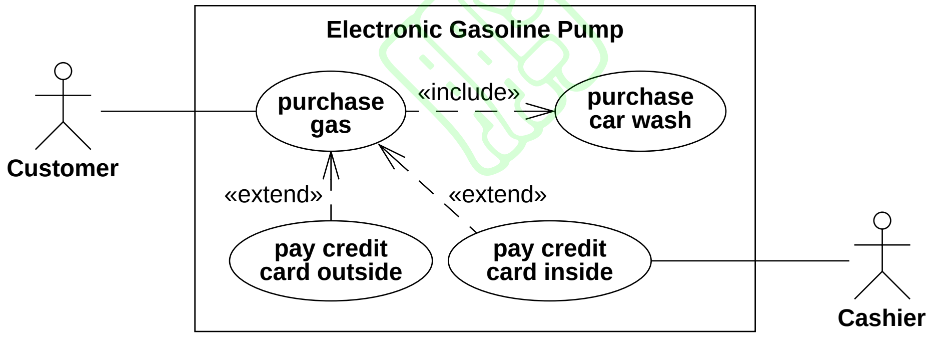


Figure A8.1 Use case diagram for an electronic gasoline pump

b. There are two actors:

- **Customer**. A person who initiates the purchase of gas.
- **Cashier**. A person who handles manual credit card payments and monitors the sale of gas.

c. There are four use cases:

- **Purchase gas**. Obtain gas from the electronic gas pump and pay for it with cash.
- **Purchase car wash**. A customer also decides to purchase a car wash and pays for it with cash.
- **Pay credit card outside**. Instead of cash, pay for the gas and optional car wash with a credit card that is directly handled by the gas system.
- **Pay credit card inside**. Instead of cash, pay for the gas and optional car wash with a credit card that is manually handled by the cashier.

8.2 Figure A8.2 shows a use case diagram for an online travel agent.

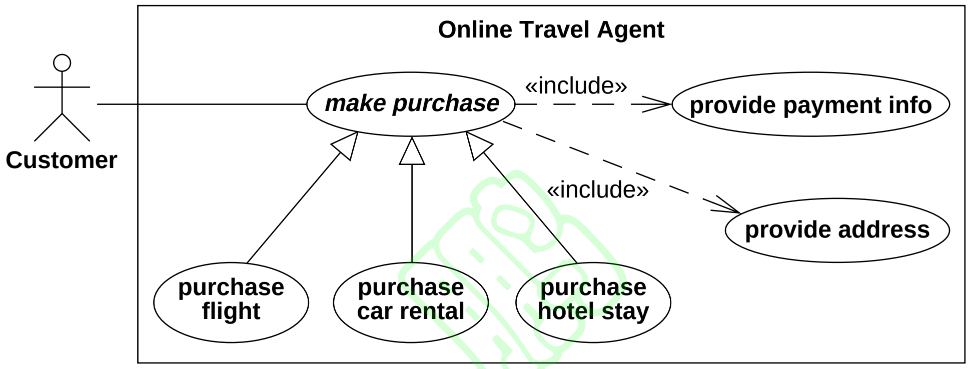


Figure A8.2 Use case diagram for an online travel agent

8.3 Figure A8.3 shows a use case diagram for the online frequent flyer program. Finding a free flight is the same as finding a paid flight, except seats are limited for free flights (hence the *extend* relationship). When submitting a claim for missing credits, a user must first view their existing credits (hence the *include* relationship).

8.4 Figure A8.4 shows a use case diagram for the electronic music management software. We chose to make the CD an actor, because it is an external entity apart from the electronic music software.

8.5 Figure A8.5 shows a use case diagram for a simple payroll system.

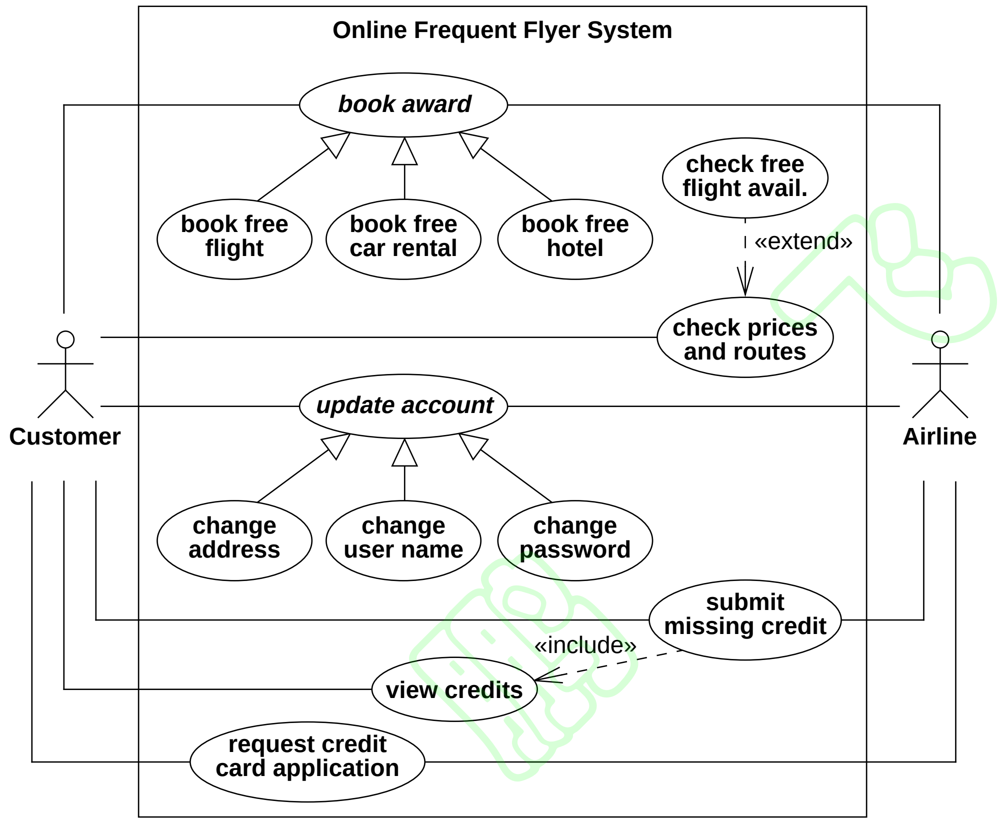


Figure A8.3 Use case diagram for an online frequent flyer program

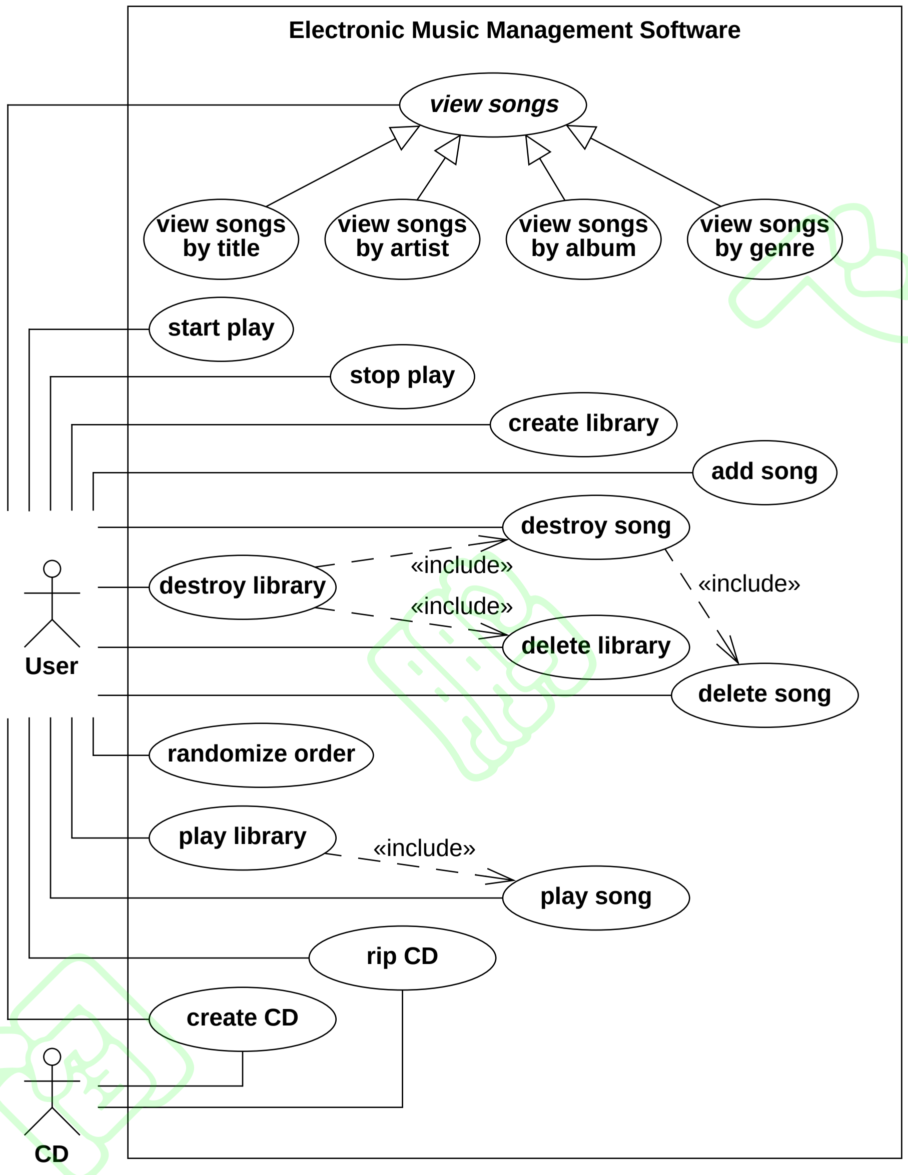


Figure A8.4 Use case diagram for electronic music management software

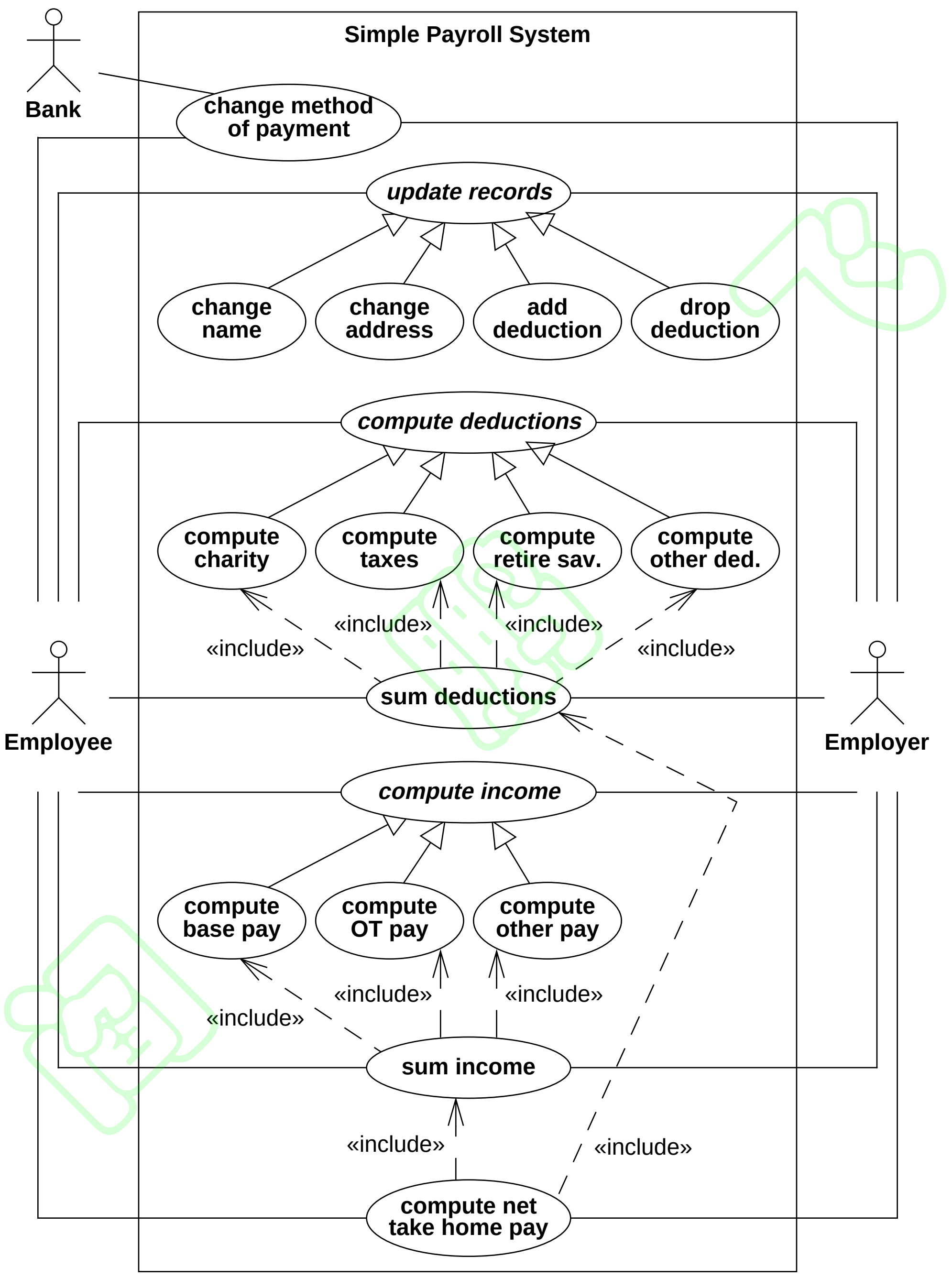


Figure A8.5 Use case diagram for a simple payroll system

8.6 Figure A8.6 computes the contents of a portfolio of stocks.

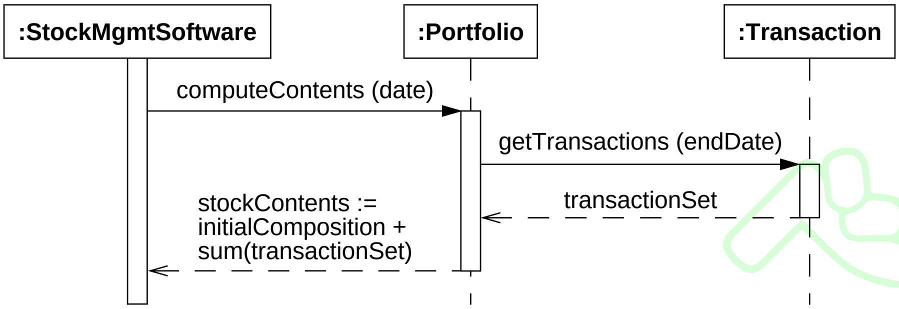


Figure A8.6 Sequence diagram for computing the contents of a portfolio of stocks

8.7 Figure A8.7 computes the value of a stock portfolio on a specified date.

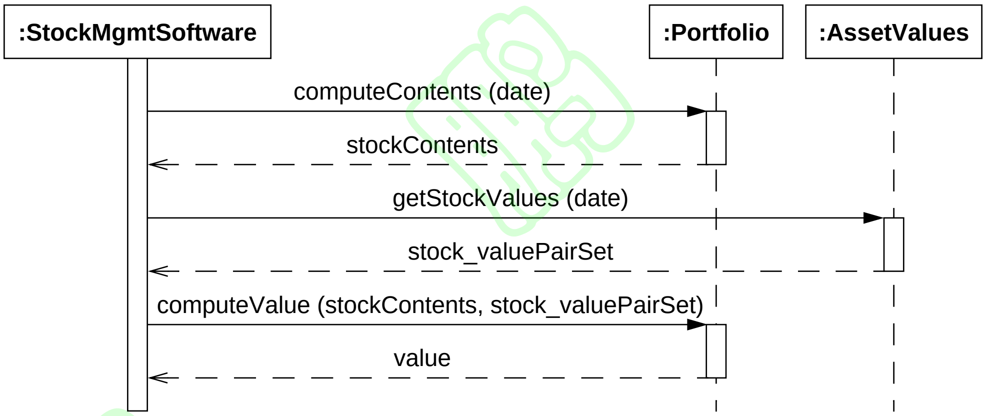


Figure A8.7 Sequence diagram for computing the value of a portfolio of stocks

8.8 Figure A8.8 compute the values of a recursive stock portfolio that is limited to three levels deep.

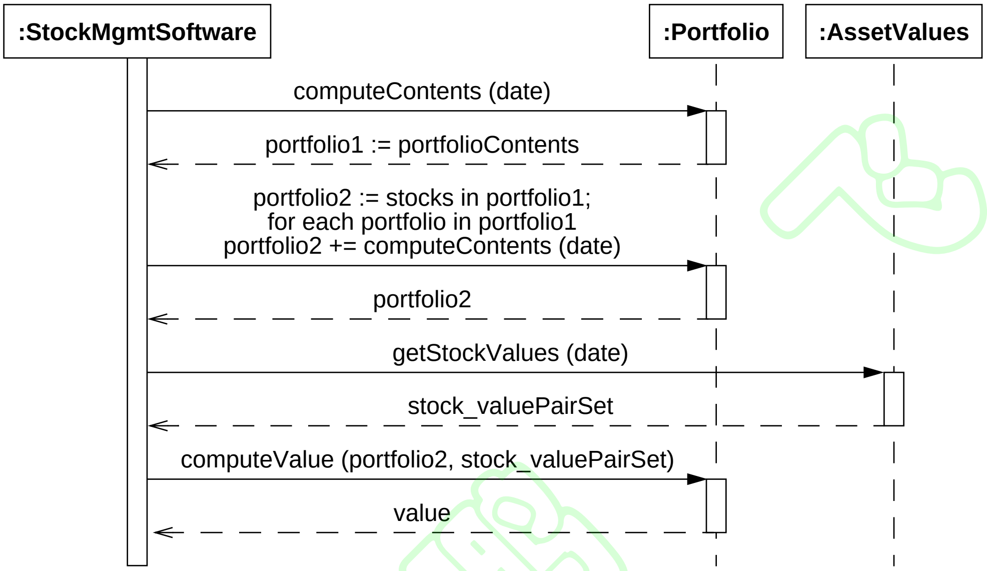

:StockMgmtSoftware
:Portfolio
:AssetValues
computeContents (date)
portfolio1 := portfolioContents
portfolio2 := stocks in portfolio1;
for each portfolio in portfolio1
portfolio2 += computeContents (date)
portfolio2
getStockValues (date)
stock_valuePairSet
computeValue (portfolio2, stock_valuePairSet)
value

Figure A8.8 Sequence diagram for computing the value of a recursive portfolio

8.9 Figure A8.9 shows the various activities and interactions for a DVD purchase.

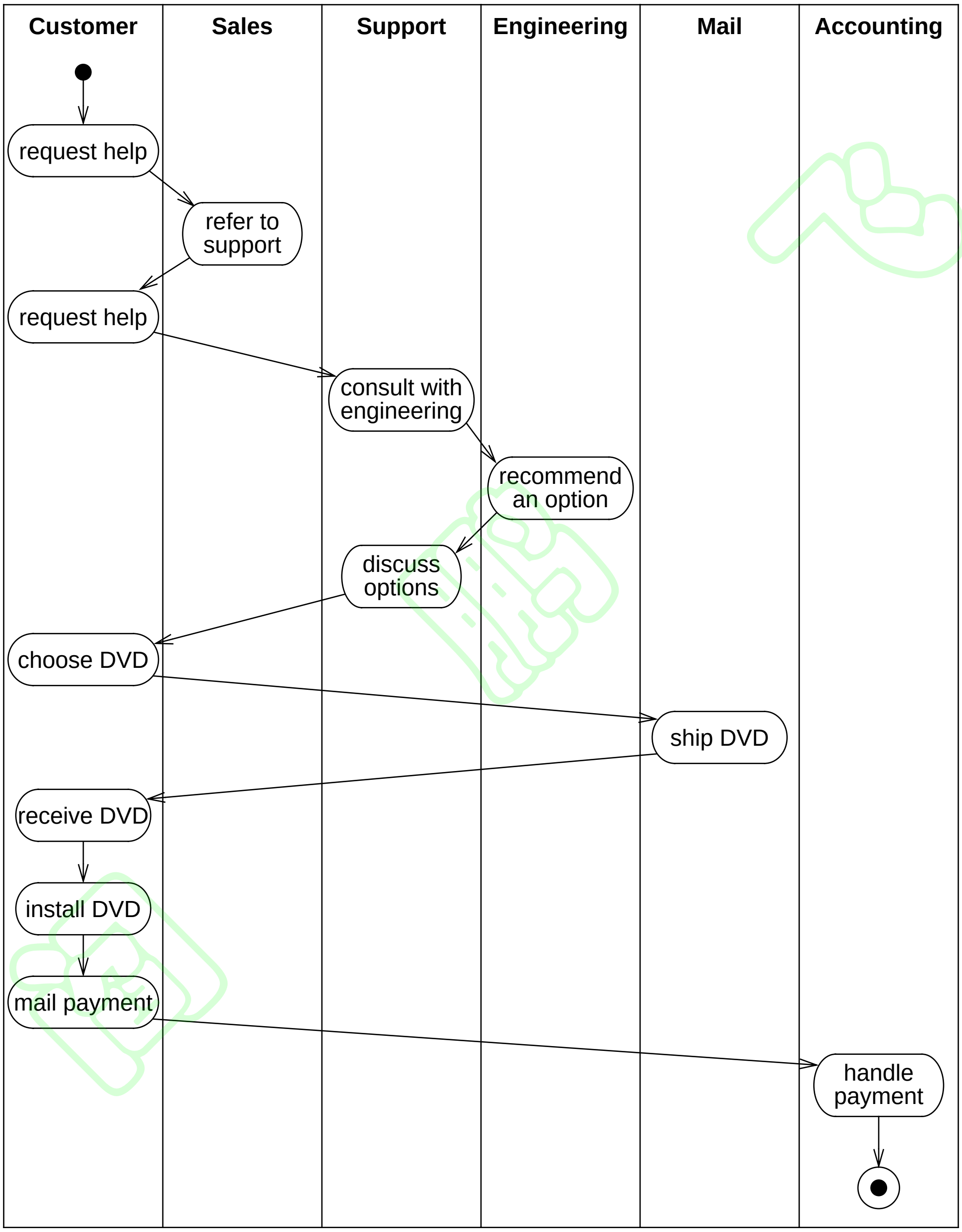


Figure A8.9 Activity diagram for a DVD purchase

8.10 Figure A8.10 shows an activity diagram for the creation of a product.

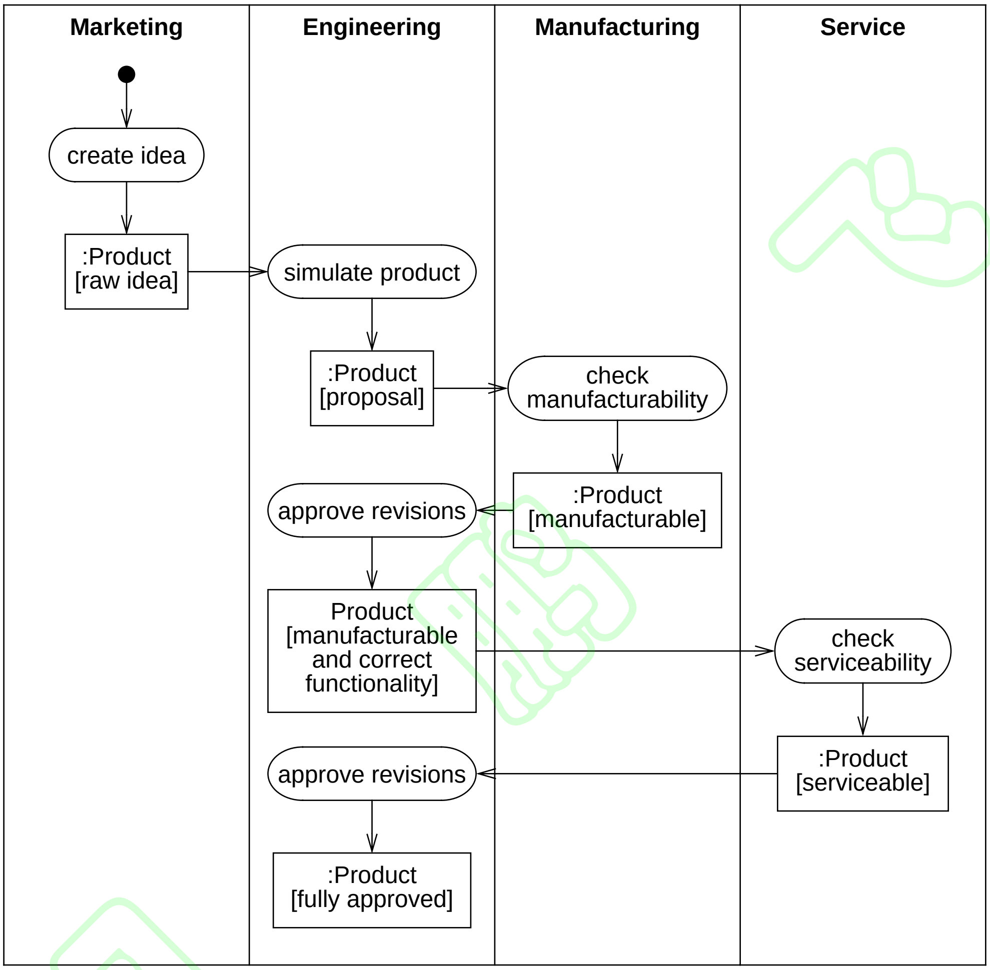


Figure A8.10 Activity diagram for the creation of a product

9

Concepts Summary

[There are no exercises in Chapter 9.]

Part 2: Analysis and Design

10 Process Overview

10.1 We have learned this lesson more times than we would care to admit. Carpenters have a similar maxim: “Measure twice, cut once.” This exercise is intended to get the student to think about the value of software engineering in general. There is no single correct answer. It is probably too early in the book for the student to answer in detail about how software engineering will help. Look for indications that the student appreciates the pitfalls of bypassing careful design.

The effort needed to detect and correct errors in the implementation phase of a software system is an order of magnitude greater than that required to prevent errors through careful design in the first place. Many programmers like to design as they code, probably because it gives them a sense of immediate progress. This leads to conceptual errors which are difficult to distinguish from simple coding mistakes. For example, it is easy to make conceptual errors in algorithms that are designed as they are coded. During testing, the algorithm may produce values that are difficult to understand. Analysis of the symptoms often produces misleading conclusions. It is difficult for the programmer to recognize a conceptual error, because the focus is at a low level. The programmer “cannot see the forest for the trees”.

10.2 The intent of this exercise is get the student to think creatively about object-oriented techniques. There is no single correct answer.

In language design, for example, a variation of class diagrams could be used to describe production rules. (See Exercise 4.15.) Production rules define how non-terminals are constructed from terminals and other non-terminals. There are two basic operators in the rules: *and* and *or*. There is a loose analogy between the and/or structure of production rules and the and/or nature of aggregation and generalization relationships. As an added benefit, the notion of multiplicity could be used to express collections and options in a more elegant fashion than the paradigms used to express them via production rules.

11

System Conception

11.1 Here is elaboration for an antilock braking system for an automobile.

a. An antilock braking system could target the mass market. If the antilock system was inexpensive and safer than current technology, it could be government mandated and installed on all cars. (Further study would be needed to determine what price is "inexpensive" and what would be a "significant" safety improvement.)

There would be several stakeholders. Auto customers would expect improved safety and minimal detriment to drivability. Auto manufacturers would want to minimize the cost and quantify the benefit so they could tout the technology in their advertising. The government would be looking for a statistical safety improvement without compromising fuel efficiency.

If the new system was inexpensive, worked well, and did not hurt drivability, all car owners could be potential customers. An expensive antilock system could be a premium option on high-end cars.

b. Desirable features would include: effective prevention of brake locking, ability to detect excessive brake wear, and acquisition of data to facilitate auto maintenance. Some undesirable features would be: reduced fuel efficiency, reduced drivability, and greater maintenance complexity.

c. An antilock system must work with the brakes, steering, and automotive electronics.

d. There would be a risk that an antilock braking system could fail leading to an accident and a lawsuit. Also it might be difficult to understand fully how the antilock system would interact with the brakes.

11.2 Here is elaboration for Internet bookselling software.

a. An Internet bookseller would most likely seek a large market. There is a modest profit per book and quantity would be needed to realize significant revenue.

Both individuals and businesses would be potential customers. Individuals would tend to buy in small quantities, while businesses would occasionally order large quantities. Another important stakeholder would be bookselling companies; use of the Internet would let them reduce personnel costs and centralize inventory, reducing inventory costs.

Once again the potential market would be large. Computer owners with Internet access would be candidate customers. Even the general public could be target customers, since the Internet can be accessed from many libraries, Internet cafes, and sometimes within conventional book stores.

b. Desirable features would include: a simple user interface, a fast response, and a vast offering of books. Modern-day Internet booksellers go even further by building an online virtual community through which customers can exchange comments and opinions about books. Some undesirable features would be lack of security for credit information, insensitivity to user privacy, and a buggy/unavailable Web site.

c. An Internet bookseller must work with credit card processing systems, order fulfillment systems, and financial accounting software.

d. One risk would be illegal hacking of customer information. Another risk would be excessive competition and an inadequate profit margin to cover overhead and operating costs.

11.3 Here is elaboration for kitchen-design software.

a. Basic software could be targeted at do-it-yourself homeowners. Advanced software could target kitchen remodeling firms.

The potential customers would be stakeholders. The sponsoring firm would have a financial stake in the cost of development and the subsequent revenue. Kitchen suppliers could benefit if convenient software created additional demand for their merchandise and services; as a consequence they also might be willing to fund development.

There is only a modest number of kitchen remodeling firms so there would be a limited market for the advanced software. However, if the software delivered substantial benefit they might pay a premium price. Similarly, our guess is that only a small fraction of homeowners would consider using the software themselves. Consequently, kitchen-design software would only be profitable if it could be developed at a low cost—maybe by adding to general-purpose drawing software—or the software development firm derived revenue from remodeling projects.

b. Desirable features would include: 3-D graphics, ease of use, and an ability to estimate project cost, both for do-it-yourself and subcontracting. Undesirable features would be: buggy software, a slow user response, and a lack of rich kitchen features.

c. Kitchen-design software must run within an operating system, such as on PCs or Apple Macs. The design software might be based on general-purpose graphics software, in which case, it would have to operate satisfactory with that. It would be helpful if the design software could export bills-of-material to other systems that might manage raw materials (especially for kitchen remodeling firms).

d. One risk would be that the software would be too difficult for the typical person to use—this would limit sales. Another risk would be that the software would have insufficient features and not be of significant help.

11.4 Here is elaboration for an online auction system.

a. The application would be targeted at the mass market.

Stakeholders would be the customers, the sellers of goods, and the company owning the online auction software.

Most everyone with a PC and Internet access could be a customer, as well as persons who visit libraries and Internet cafes.

b. Auction software should make it easy to find desired goods, it should have the trust of both buyers and sellers, it must run fast, be highly available, and have low cost. Undesirable features would be: an awkward user interface, a lack of sufficient inventory, sale of illegal items, and slow shipment.

c. Systems with which an online auction system must work are: credit card processing, Web browsers, and help software.

d. One risk is that online auctions could involve illegal products; countries have various rules that must be enforced. Another risk is that the buyer or seller might cheat—buyers may not honor successful bids and sellers may not deliver merchandise which has been paid for.

11.5 [Student answers may vary widely from the answers given below.]

a. bridge player. Develop a computerized bridge playing system. The system will support as many as four players. From zero to four of the players will be computer generated. The computer must deal random hands, bid using standard conventions, play any hand, and keep score. All computer generated opponents must be “honest” and not take advantage of any knowledge of the hidden cards of opponents. There should be a setting for level of difficulty which the human user can adjust. The program must have an excellent, user-friendly interface with high resolution color graphics and quickly determine bids and plays. It is permissible for performance to degrade as the level of play becomes more difficult. The system should keep an optional log in which it records the cards dealt and the bids and plays for a given game. The system should also accept a predefined deal and bid and play list; this is useful for studying fine details of a bridge game.

b. change-making machine. Design the software for a machine that accepts paper currency and returns change. Important design goals in order of importance are: rejection of counterfeit and foreign currency, determination of denomination, correct dispensing of change, software versatility, and low cost. The software may be custom written for a particular microprocessor chip. Versatility refers to the fact that the software must be configurable to a variety of conditions. The software must be easy to reconfigure for international use with different types of currency and different formulas for dispensing change. The software must allow a fee to be imposed for change-making service.

c. **car cruise control**. Design a cruise control system for an automobile. The control has four buttons: *on/off*, *set*, *coast*, and *resume*. Once the control system is on, the driver accelerates to a desired speed and presses *set*. The speed will be maintained within a fixed tolerance until the driver hits the brake or presses the *off* button.The driver may accelerate above the preset speed by using the accelerator; once the accelerator is released the car will resume the preset speed. If the driver hits the brake, the cruise control is disabled until the driver presses the *resume* button at which time the car will resume the preset speed. If the driver holds the *coast* button, the car will decelerate until the button is released at which point the car's speed becomes the new desired speed.Abrupt changes or oscillations in speed will be avoided.

d. **electronic typewriter**. Design the software and hardware for an electronic typewriter. The typewriter only has to support the standard *QWERTY* arrangement. Keys that are not letters and numbers can be arranged in the manner that seems most appropriate. Cost of the typewriter is paramount; intentions are to aim for the low end of the marketplace. The power supply need only handle the standard 120 volts of North America. Color of the typewriter is irrelevant—choose an inoffensive color that is inexpensive. The typewriter should be lightweight and easy to assemble; it need not be easy to repair. The typewriter should have a one-line electronic display; the line does not print until the carriage return is pressed. This buffering simplifies correction of minor typing errors.

e. **spelling checker**. Design the software for a spelling checker. The spelling checker must find incorrect words in a document and suggest corrections for all misspelled words. The spelling checker must use a word dictionary and permit the user to add new words. The software must run on PCs and integrate with a variety of word processors. It must be memory resident and easily activated with a few keystrokes. The spelling checker must accept commands from both a mouse and keyboard; keyboard commands should be redefinable by modifying a configuration file. The software must be easy to use and present a polished pull-down menu interface. It is important that the system occupy as little memory as possible.

f. **telephone answering machine**. Design the software for a telephone answering machine. The software must provide the following services as a minimum: answer the phone after a predetermined number of rings, play a recorded message, record the caller's message, and hang up after a predefined length of recording. The software should support remote dial-in and identification by password to hear any recorded calls. The software should be suitable for burning into ROM. As such it is important that the software be small in size and extremely reliable since it would be very costly to update the equipment once it is in a customer's hands. The software must operate in real-time but early projections are that this goal is easy to meet with modern microprocessors. You may choose any CPU chip for developing the software that you choose, but the wholesale quantity price of the CPU chip must be $10 or less.

11.6a. A system to transfer data from one computer to another over a telecommunication line. The system should transmit data reliably over noisy channels with a failure rate of less than 1 in 10^9. Data must not be lost if the receiving end cannot keep up or if the line drops out. The system must keep pace with the fastest dial-up rate (currently about 60K baud). The system should support several common error correcting protocols.

b. A system for automating the production of complex machined parts. The parts will be designed using a three-dimensional drafting editor that is part of the system. The system will produce tapes that can be used by numerical control (N/C) machines to actually produce the parts. The system must be developed in x months and cost no more than y dollars. The system will support the following machining operations...

c. A desktop publishing system, based on a what-you-see-is-what-you-get philosophy. The system will support text and graphics. Graphics include lines, squares, rectangles, polygons, circles, and ellipses. The system should support interactive, graphical editing of documents. The system must run on the following hardware configurations... It must be capable of printout in *Postscript* and *PDF* formats. The software must support the following operations... and be extensible and maintainable.

d. Software for generating nonsense. The input is a sample document. The output is random text that mimics the input text by imitating the frequencies of combinations of letters of the input. The user specifies the order of the imitation and the length of the desired output. For order N, every output sequence of N characters is found in the input and at approximately the same frequency. As the order increases, the style of the output more closely matches the input. Working memory must be constant and not proportional to the size of input or output text, but the entire input document is randomly accessible at run time. Time performance should be proportional to N and to the number of output characters.

e. A system for distributing electronic mail over a network. Each user of the system should be able to send mail from any computer account and receive mail on one designated account. There should be provisions for answering or forwarding mail, as well as saving messages in files or printing them. Also, users should be able to send messages to several other users at once through distribution lists. Messages must not be lost even if the target computer is down.

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.

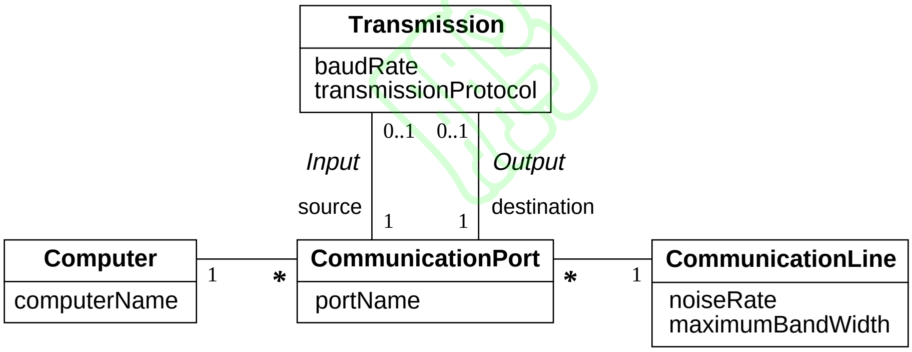


Figure A12.1 Class diagram for a data transfer system

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.

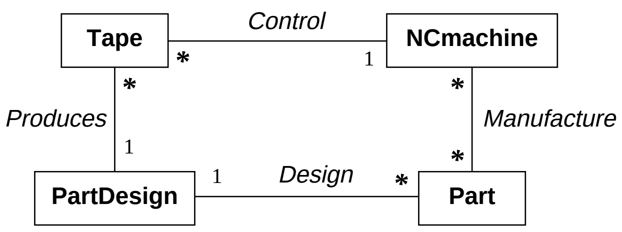


Figure A12.2 Class diagram for a parts machining automation system

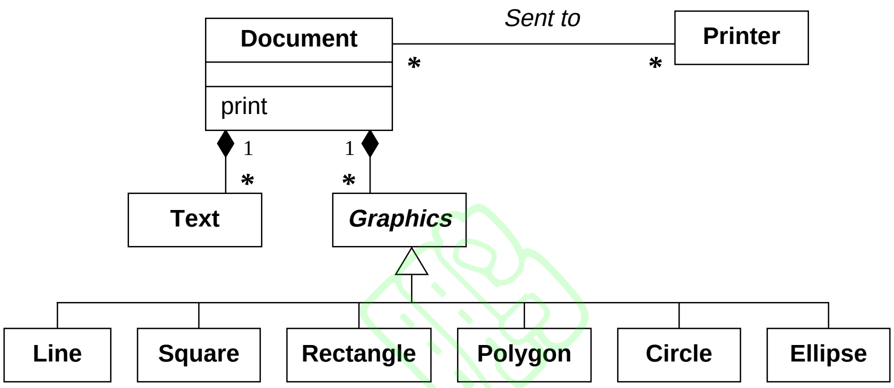


Figure A12.3 Class diagram for a desktop publishing system

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.

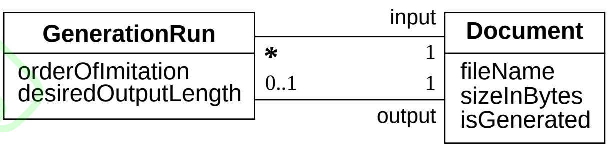


Figure A12.4 Class diagram for a nonsense generator

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.

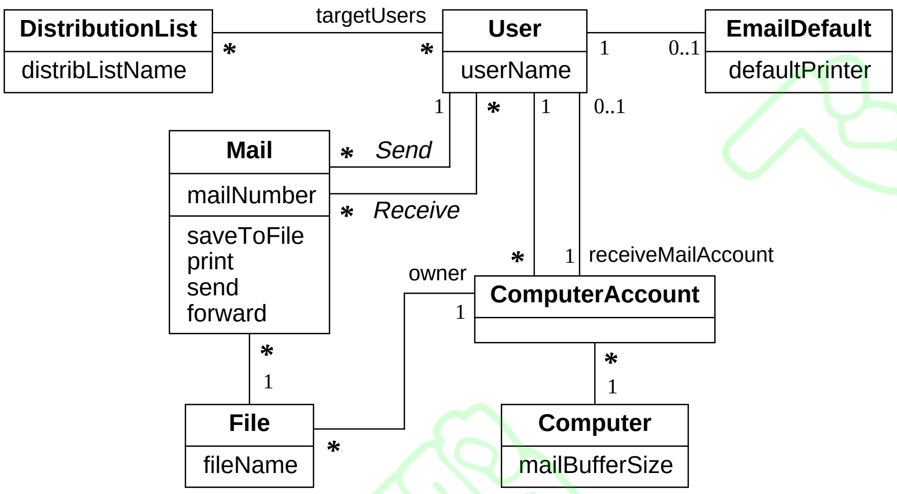


Figure A12.5 Class diagram for an electronic mail system

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*, *lineSegmentCoordinate*, *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.

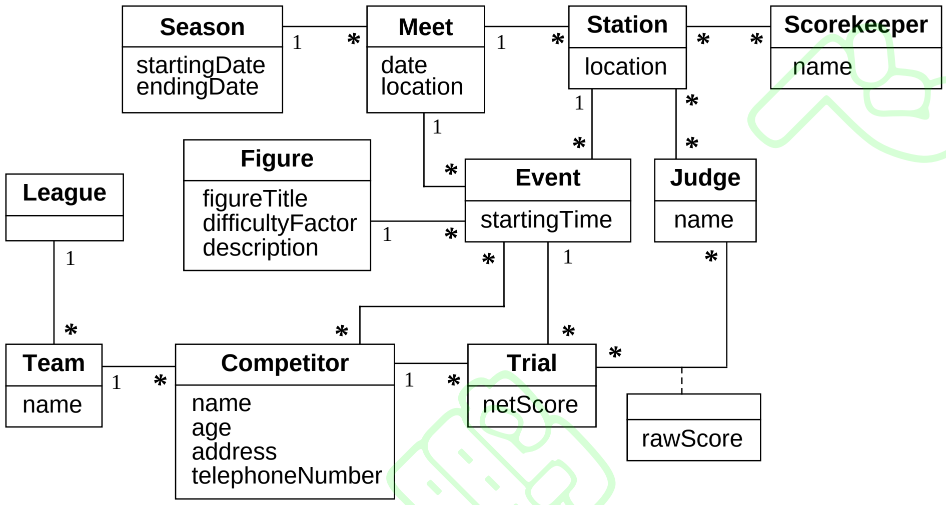


Figure A12.6 Class diagram for a scoring system that supports event registration

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.

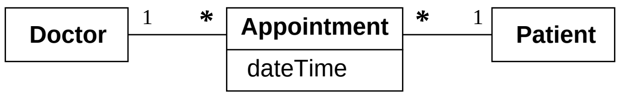


Figure A12.7 Class diagram for appointments

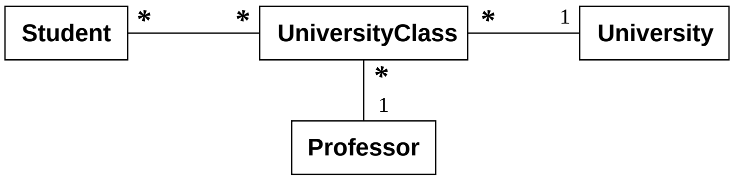


Figure A12.8 Class diagram for university classes

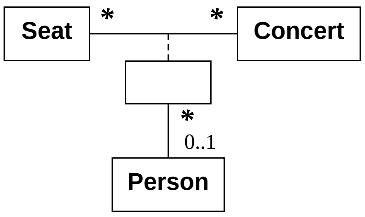


Figure A12.9 Class diagram for reservations

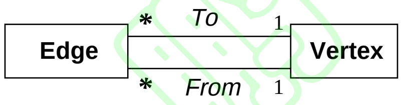


Figure A12.10 Class diagram for directed graphs

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.

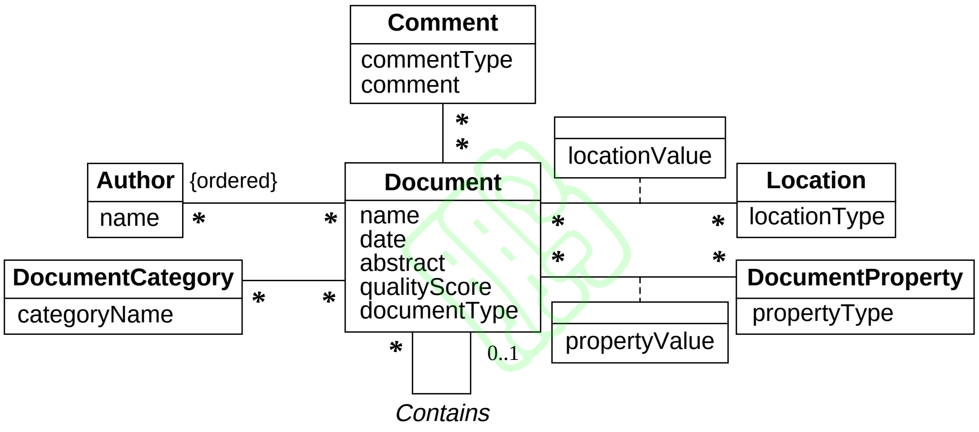


Figure A12.11 Class diagram for a document manager

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.

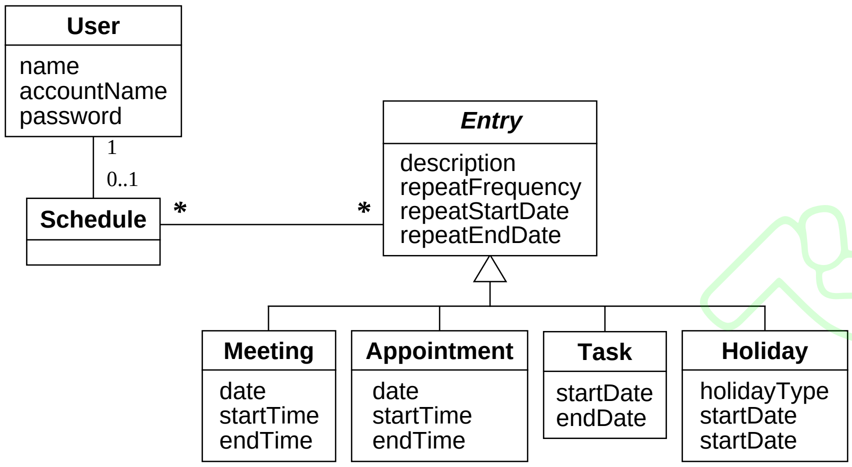


Figure A12.12 Class model for scheduler software

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.

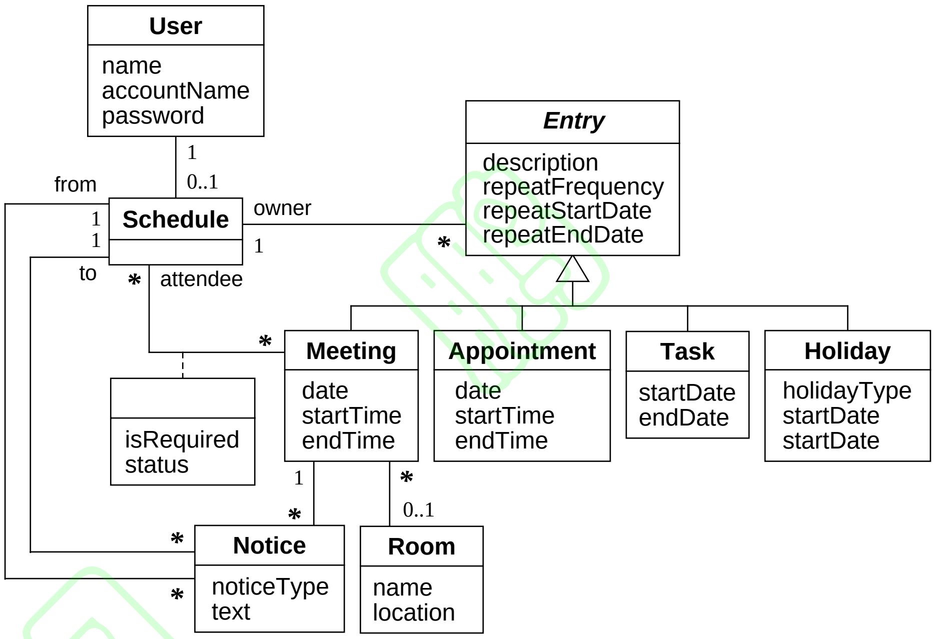


Figure A12.13 Class model for an extension to the scheduling software

13 Application Analysis

13.1 Here are scenarios for the variations and exceptions. The precise student answers will vary a bit.

- **The ATM can't read the card**.
  The ATM asks the user to insert a card.
  The user inserts a cash card.
  The ATM accepts the card and cannot read it.
  The ATM displays an error message.
  The ATM keeps the card.
  The ATM asks the user to insert a card.
- **The card has expired**.
  The ATM asks the user to insert a card.
  The user inserts a cash card.
  The ATM accepts the card and reads its expiration date.
  The ATM notes that the expiration date is before today and that the card has expired.
  The ATM displays an error message.
  The ATM keeps the card.
  The ATM asks the user to insert a card.
- **The ATM times out waiting for a response**.
  The ATM asks the user to insert a card.
  The user inserts a cash card.
  The ATM accepts the card and reads its serial number.
  The ATM requests the password.
  The user does not respond and the ATM requests the password again.
  The user does not respond and the ATM displays an error message.
  The ATM keeps the card.
  The ATM asks the user to insert a card.

- **The amount is invalid**.
  The ATM displays a menu of accounts and commands.
  The user selects an account withdrawal.
  The ATM asks for the amount of cash.
  The user enters $0.
  The ATM complains that this is an illegal amount.
  The user enters $100.
  The ATM verifies that the withdrawal satisfies its policy limits.
  The ATM contacts the consortium and bank and verifies that the account has sufficient funds.
  The ATM dispenses the cash and asks the user to take it.
  The user takes the cash.
  The ATM displays a menu of possible commands.
- **The machine is out of cash or paper**.
  The ATM asks the user to insert a card.
  The user inserts a cash card.
  The ATM accepts the card and reads its serial number.
  The ATM requests the password.
  The user enters “1234.”
  The ATM verifies the password by contacting the consortium and bank.
  The ATM displays a warning that it cannot process withdrawals and is out of cash.
  The user chooses the command to terminate the session.
  The ATM ejects the card and asks the user to take it.
  The user takes the card.
  The ATM asks the user to insert a card
- **The communication lines are down**.
  The ATM asks the user to insert a card.
  The user inserts a cash card.
  The ATM accepts the card and reads its serial number.
  The ATM requests the password.
  The user enters “1234.”
  The ATM displays the warning that the communication lines have gone down.
  The ATM ejects the card and asks the user to take it.
  The user takes the card.
  The ATM displays the warning that the communication lines are down.
- **The transaction is rejected because of suspicious patterns of card usage**.
  The ATM asks the user to insert a card.
  The user inserts a cash card.
  The ATM accepts the card and reads its serial number.
  The ATM displays the message that it is keeping the card and that the user should contact the issuing bank.
  The ATM asks the user to insert a card

13.2 Figure A13.1, Figure A13.2, and Figure A13.3 show the state diagrams for *Deposit*, *Withdrawal*, and *Query* respectively.

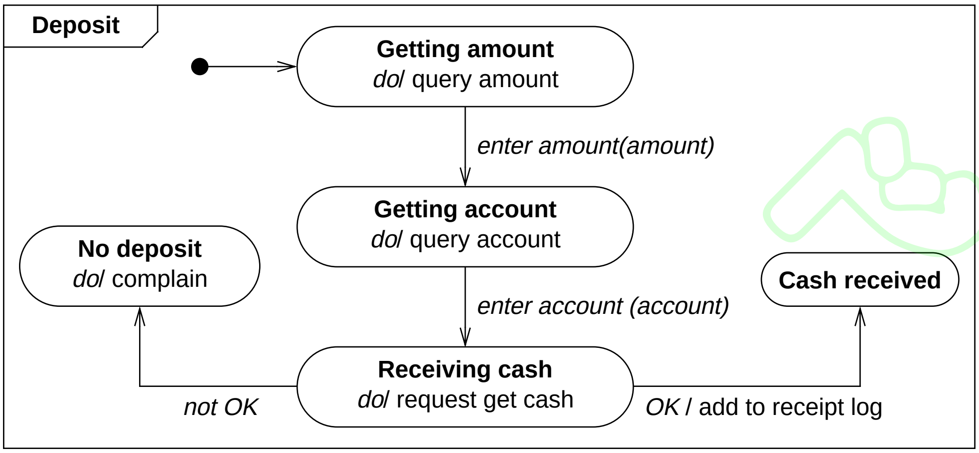


Figure A13.1 State diagram for *Deposit*

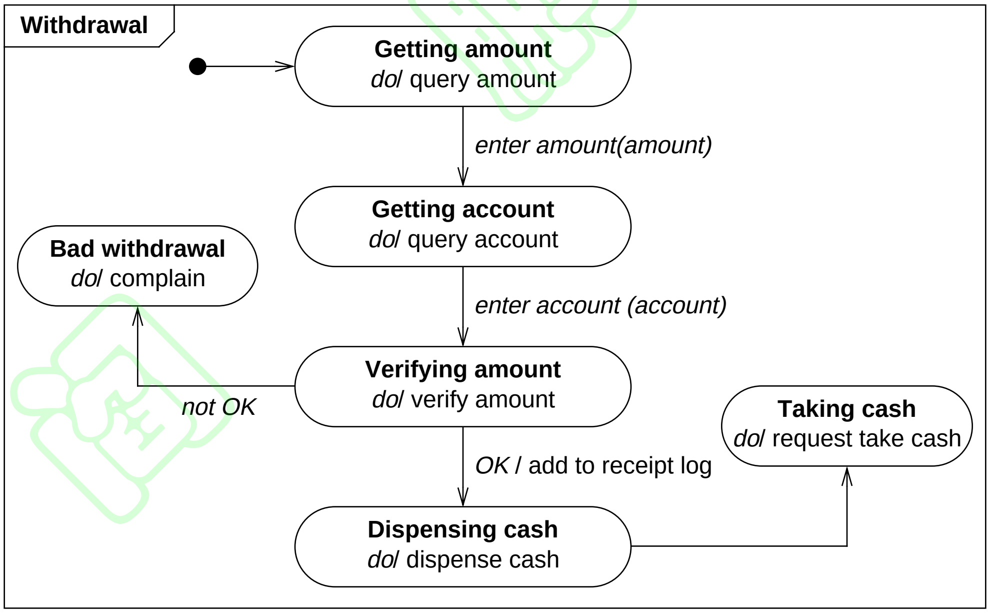


Figure A13.2 State diagram for *Withdrawal*

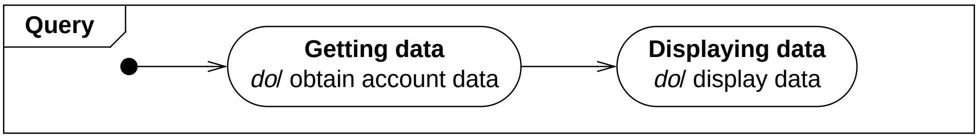


Figure A13.3 State diagram for *Query*

13.3 Figure A13.4 shows the sequence diagram for the given scenario.

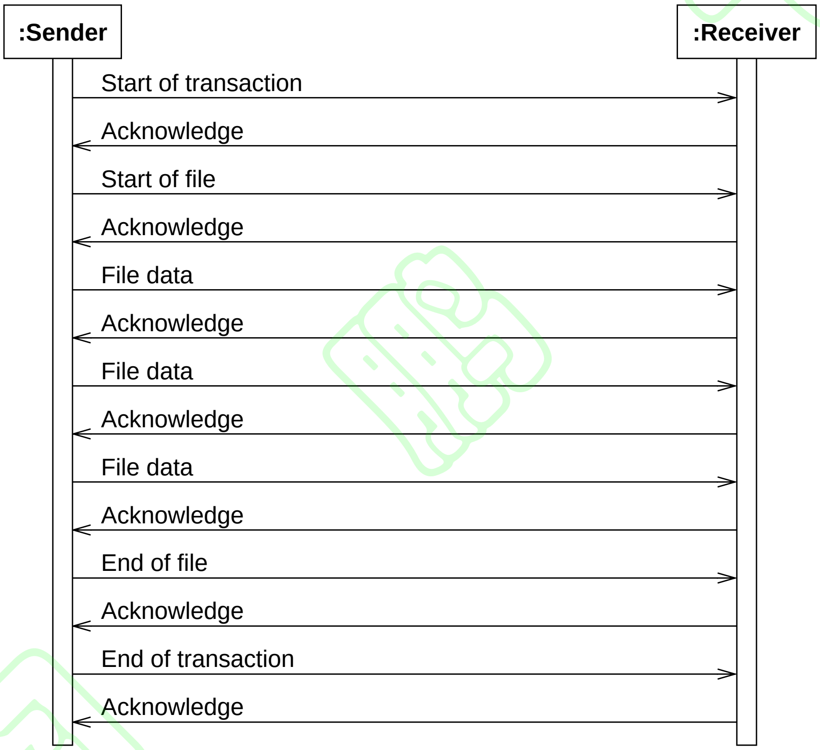


Figure A13.4 Sequence diagram for a data transmission protocol

13.4 Figure A13.5 shows a sequence diagram for a scenario in which several packets are garbled. Note that the exercise did not ask for errors caused by corruption of receiver packets. In an actual protocol, this would have to be taken into account.

13.5 Both the sender and receiver have state diagrams. We assume that a separate process supplies the sender with files to be sent and that another process stores the data on the

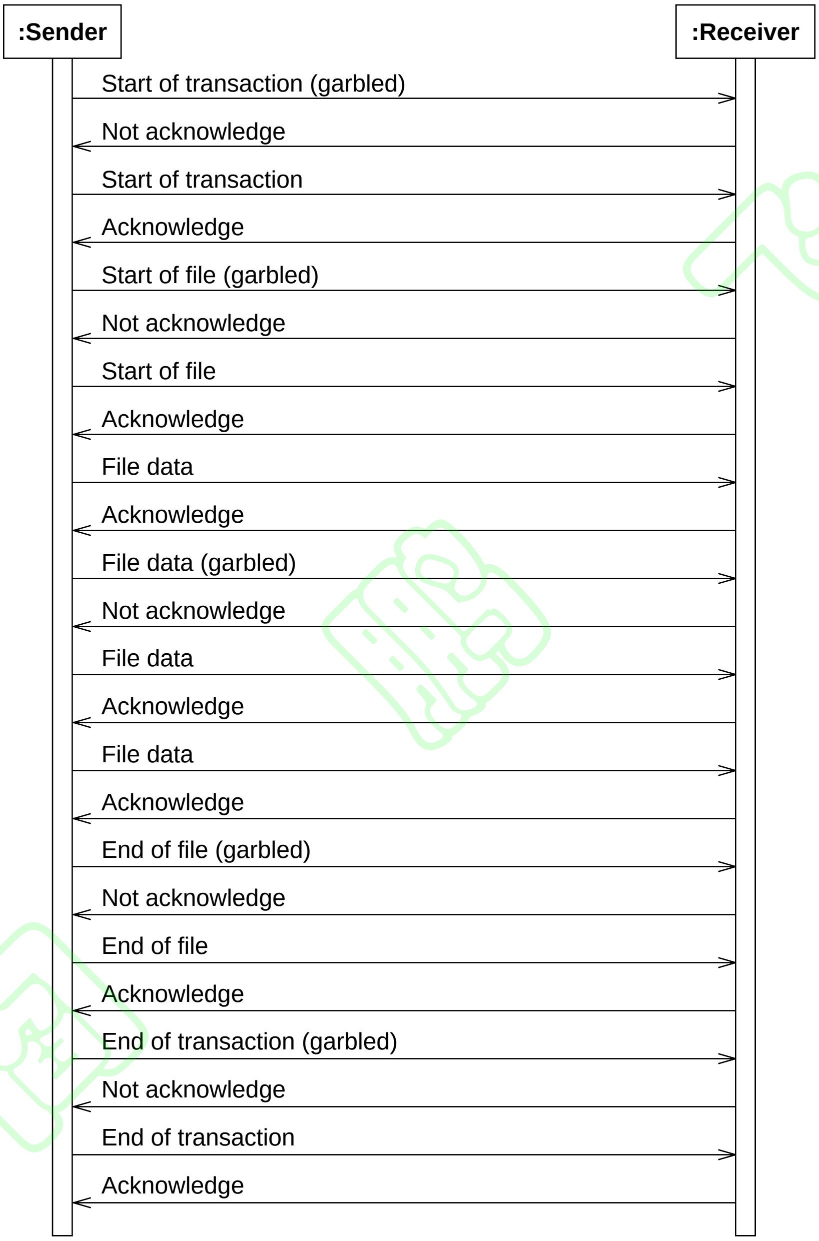


Figure A13.5 Sequence diagram for a data transmission protocol with errors

receiver end. Figure A13.6 shows a simplified state diagram for the sender. Figure A13.7 shows the state diagram for the receiver. The abbreviations *ack* and *nack* indicate acknowledged and not acknowledged. The diagram does not show the states needed to control the flow of data to processes that supply the sender and service the receiver. Note that neither of the state diagrams deals with complete system failure. In practice, time outs and retry counts would probably be used to make the system more robust.

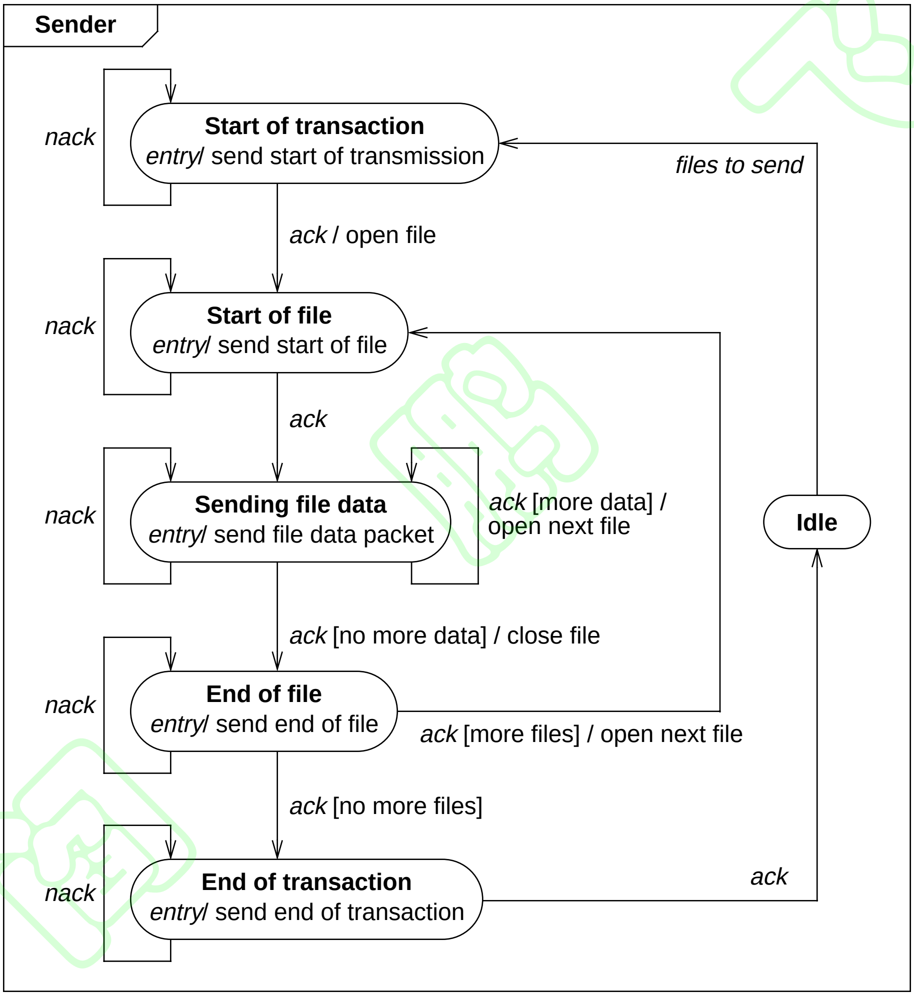

Sender
Start of transaction
entry/ send start of transmission
Start of file
entry/ send start of file
Sending file data
entry/ send file data packet
End of file
entry/ send end of file
End of transaction
entry/ send end of transaction
Idle
nack
nack
nack
nack
nack
files to send
ack / open file
ack
ack [more data] / open next file
ack [no more data] / close file
ack [more files] / open next file
ack [no more files]
ack

Figure A13.6 State diagram for the sender

13.6 Figure A13.8 shows a state diagram consistent with the scenarios.

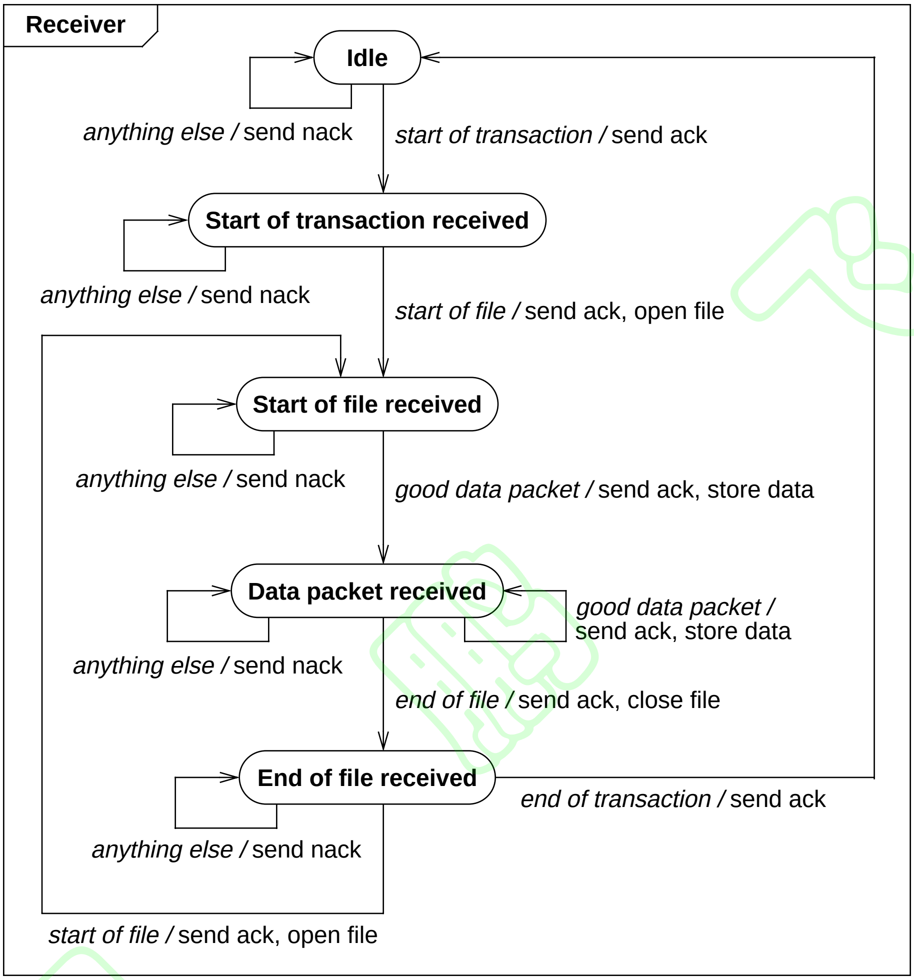

Receiver
Idle
anything else / send nack
start of transaction / send ack
Start of transaction received
anything else / send nack
start of file / send ack, open file
Start of file received
anything else / send nack
good data packet / send ack, store data
Data packet received
good data packet / send ack, store data
anything else / send nack
end of file / send ack, close file
End of file received
end of transaction / send ack
anything else / send nack
start of file / send ack, open file

Figure A13.7 State diagram for the receiver

13.7 The application is a simple editor that supports only boxes, links, and text. Text is allowed only in boxes and the text size and font is fixed. Boxes automatically adjust to fit the enclosed text. Links consist solely of horizontal and vertical lines.

13.8 Actors are the user and the file system.

13.9 [Instructor's note: you should give the students the actors from the previous exercise.]
Use cases represent a kind of service that the system provides and should be at the same level of detail. Consequently the use cases should not reflect detailed operations,

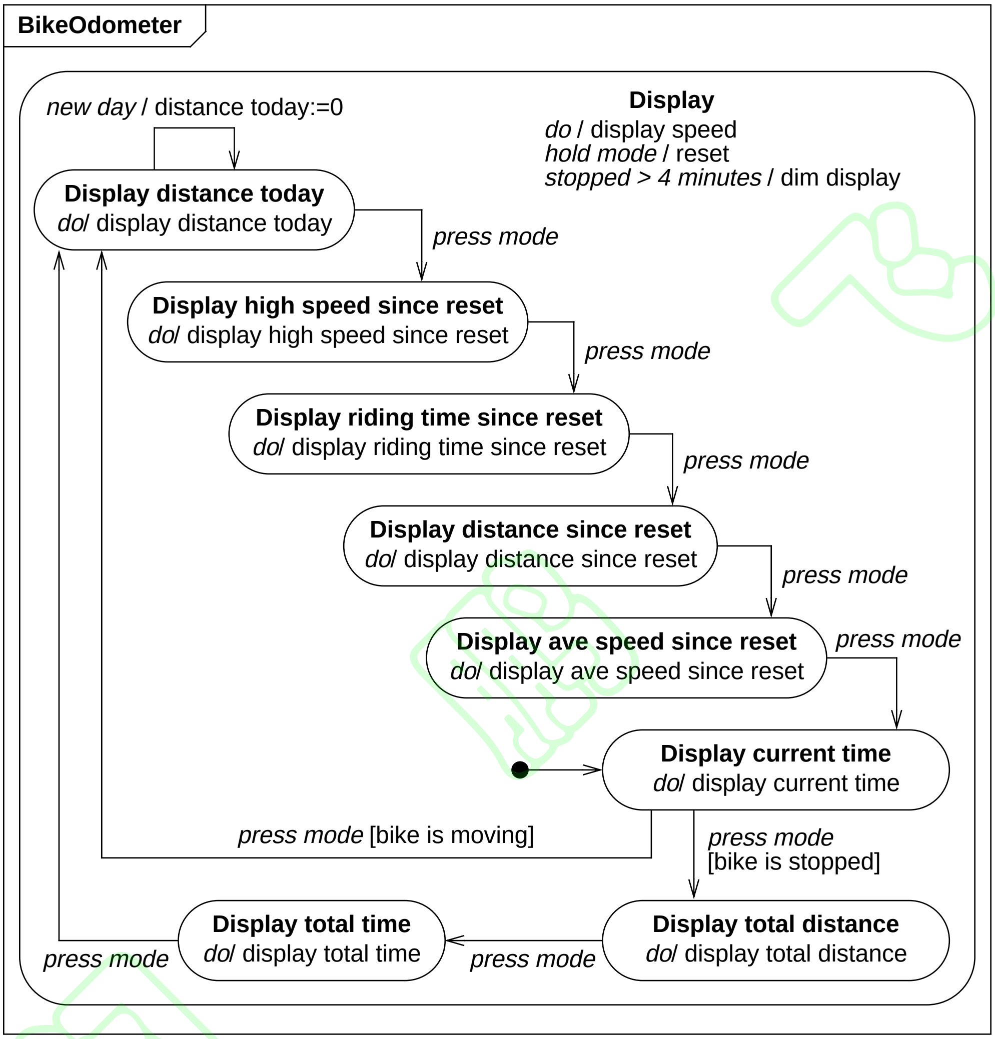

BikeOdometer
Display
do / display speed
hold mode / reset
stopped > 4 minutes / dim display
new day / distance today:=0
Display distance today
do/ display distance today
press mode
Display high speed since reset
do/ display high speed since reset
press mode
Display riding time since reset
do/ display riding time since reset
press mode
Display distance since reset
do/ display distance since reset
press mode
Display ave speed since reset
do/ display ave speed since reset
press mode
Display current time
do/ display current time
press mode [bike is moving]
press mode [bike is stopped]
Display total distance
do/ display total distance
press mode
Display total time
do/ display total time
press mode

Figure A13.8 State diagram for a bike odometer

but instead should focus on high level tasks that the user can perform. Here are the use cases. Figure A13.9 shows a use case diagram.

- **Create drawing**. Start a new, empty drawing in memory and overwrite any prior contents. Have the user confirm, if there is a prior drawing that has not been saved.
- **Modify drawing**. Change the contents of the drawing that is loaded into memory.
- **Save to file**. Save the drawing in memory to a file.

- **Load from file**. Read a file and load a drawing into memory overwriting any prior contents. Have the user confirm, if there is a prior drawing that has not been saved.
- **Quit drawing**. Abort the changes to a drawing and clear the contents of memory.

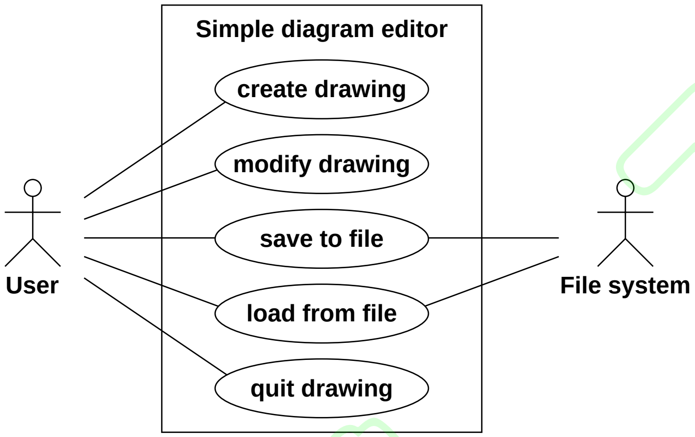


Figure A13.9 Use case diagram for the simple diagram editor

13.10 Figure A13.10 organizes the use cases. The overwrite confirmation is another use case that is included in creating a drawing and loading from a file.

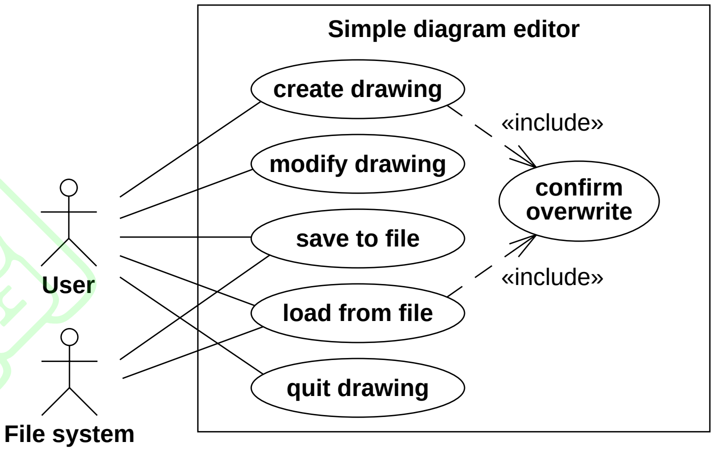


Figure A13.10 Use case diagram with relationships for the simple diagram editor

13.11 There are an infinite number of correct answers to this exercise, one of which is listed below.

- User loads an existing drawing. Editor retrieves the document and sets the cursor to the last referenced sheet.
  User goes to the first sheet. Editor moves the cursor.
  User goes to the next sheet. Editor moves the cursor.
  User deletes this sheet. Editor requests confirmation. User confirms.
  User goes to the last sheet. Editor moves the cursor.
  User goes to the previous sheet. Editor moves the cursor.
  User deletes all existing sheets. Editor requests confirmation. User confirms.
  User creates a new sheet. Editor sets the cursor to this sheet.
  User creates a box. Editor highlights newly created box. User enters text “x”.
  User copies x-box. Editor highlights new copy of box. User moves selected box.
  User selects text in box. Editor highlights text and unhighlights box. User cuts text. User selects empty box. Editor highlights empty box. User enters text “y”.
  User copies y-box. Editor highlights new copy of box. User moves selected box. User edits the “y” and changes it to “+”.
  User selects y-box. Editor highlights y-box. User copies y-box. Editor highlights new copy of box. User moves selected box. User edits “y” and changes it to “x+y”.
  User cuts x+y-box. User changes his/her mind and pastes the x+y-box back in.
  User selects the x-box. Editor highlights x-box. User also selects the +-box. Editor also highlights +-box. User links the boxes.
  User selects the y-box. Editor highlights y-box. User also selects the +-box. Editor also highlights +-box. User links the boxes.
  User selects the +-box. Editor highlights +-box. User also selects the x+y-box. Editor also highlights x+y-box. User links the boxes.
  User selects all boxes and links. Editor highlights all boxes and links. User groups the selections. User aligns the grouped selection with regard to the left-right center of the page.
  User renames the drawing file and saves the drawing.

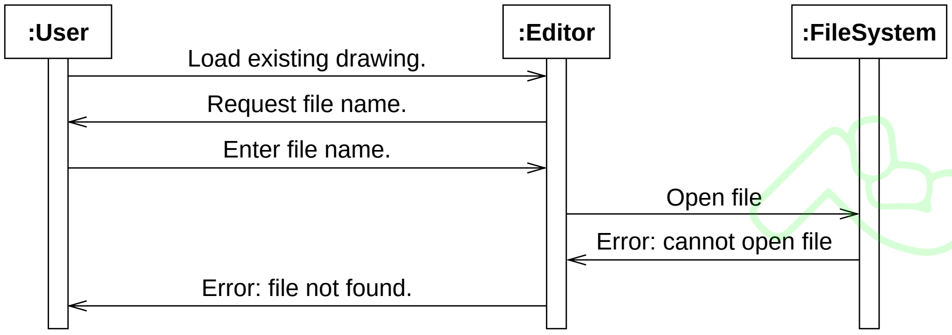
13.12 Error scenario 1:
User enters command: *load existing drawing file* and supplies a file name.
Command fails: file not found.

Error scenario 2:
User selects the x-box. Editor highlights x-box.
User also selects the y-box. Editor also highlights y-box.
User also selects the +-box. Editor also highlights +-box.
User tries to select the *link box* command.
Command fails: must pick exactly two boxes for linking.

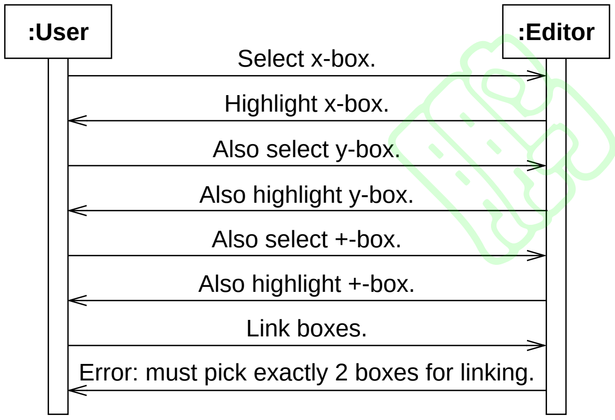
Error scenario 3:
User selects the x-box. Editor highlights x-box.
User tries to select the *enter text* command.
Command fails: box already has text.

13.13 Figure A13.11 has the sequence diagrams for Exercise 13.12.

Sequence diagram for error scenario 1:



Sequence diagram for error scenario 2:



Sequence diagram for error scenario 3:

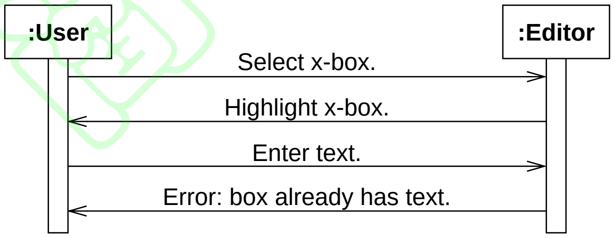


Figure A13.11 Sequence diagram for error editor scenarios

13.14 The application manages data for competitive meets in a swimming league. The system stores swimming scores that judges award and computes various summary statistics.

13.15 Actors are competitor, scorekeeper, judge, and team.

13.16 [Instructor's note: you should give the students the actors from the previous exercise.] Here are definitions for the use cases. Figure A13.12 shows a use case diagram.

- **Register child**. Add a new child to the scoring system and record the name, age, address, and team name. Assign the child a number.
- **Schedule meet**. Assign competitors to figures and determine their starting times. Assign scorekeepers and judges to stations.
- **Schedule season**. Determine the meets that comprise a season. For each meet, determine the date, the figures that will be performed, and the competing teams.
- **Score figure**. A scorekeeper observes a competitor's performance of a figure and assigns what he/she considers to be an appropriate raw score.
- **Judge figure**. A judge receives the scorekeepers' raw scores for a competitor's performance of a figure and determines the net score.
- **Compute statistics**. The system computes relevant summary information such as top individual score for a figure and total team score for a meet.

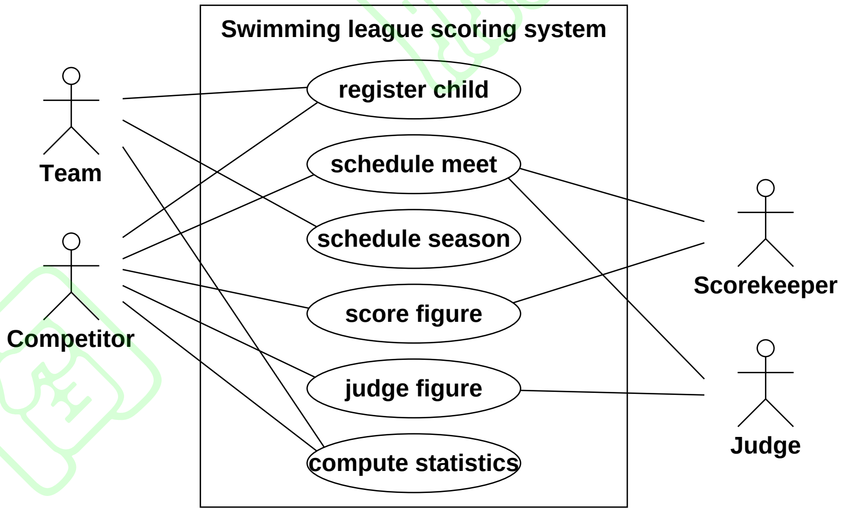


Figure A13.12 Use case diagram for the swimming league scoring system

13.17 Scenarios are intrinsically less interesting for the swimming league than for the earlier diagram editor exercise. The diagram editor is an interactive application that has an important interaction model. In contrast, the swimming league is a data management problem, where most operations merely store and retrieve data. The swimming league has a complex class model and a rich variety of queries, but relatively simple scenarios for acquiring data.

At the beginning of the 1991 swimming season, the following data was entered into the system. The two teams were the Dolphins and the Whales. The six competitors were Heather Martin, Elissa Martin, Cathy Lewis, Christine Brown, Karen Solheim, and Jane Smith. The name, age, address, and telephone number of each competitor was entered into the system. Heather Martin, Elissa Martin, and Cathy Lewis were members of the Dolphins team. The other three competitors were members of the Whales team. The three judges were Bill Martin, Mike Solheim, and Jim Morrow.

The meets were scheduled for July 6 at Niskayuna, July 20 at Glens Falls, and August 3 in Altamont. In each of the three meets the same four figures were scheduled to be performed for a total of twelve events. These figures are the Ballet Leg (1.0), the Dolphin (1.2), the Front Pike Somersault (1.3), and the Back Pike Somersault (1.5). Difficulty factors for figures are parenthesized.

For the July 6 meet at Niskayuna, the Ballet Leg event was scheduled to begin at 9:00 AM, the Dolphin at 9:00 AM, the Front Pike Somersault at 10:00 AM, and finally the Back Pike Somersault at 10:00 AM.

For the July 20 meet at Glens Falls, the Back Pike Somersault begins at 9:00 AM, the Front Pike Somersault at 9:00 AM, the Dolphin at 10:00 AM, and the Ballet Leg at 10:00 AM.

For the August 3 meet at Altamont, the Ballet Leg event was scheduled to begin at 9:30 AM, the Dolphin at 9:30 AM, the Front Pike Somersault at 10:30 AM, and the Back Pike Somersault at 10:30 AM.

13.18 First error scenario. Someone tries to schedule a meet for July 20 at Voorheesville. This is not permitted, since there is already a meet scheduled for that date at Glens Falls.

Second error scenario. Competitor Karen Solheim tries to give the registrar two phone numbers, one for her home and another for her father's office. The current class model cannot accept this data, since *telephoneNumber* is a simple attribute of *Competitor*.

Third error scenario. Competitor Cathy Lewis informs the registrar that she would like to compete with the Dolphins team on July 6 and 20 and the Whales team on August 3. Christine Brown and Jane Smith will miss the meet on August 3 due to vacation plans, and Cathy Lewis would like to even up the teams. The class model forbids this data entry, since a *Competitor* is specified as belonging to exactly one *Team*.

13.19 Printing forms. For each competitor currently in the system:

- Print out the current name, age, address, and telephone number, then
- Mail the form to each competitor.

Processing forms. There are a number of possibilities, such as adding a new competitor, deleting an old competitor, keeping records for a competitor but deactivating the competitor for the upcoming season, and changing information for a competitor. Note that *age* is a poor choice of base attribute for the *Competitor* class. The model would be improved by making *birthdate* a base attribute and *age* a derived attribute from the *currentDate* and *birthdate*.

Karen Solheim returned her form and changed her address from 722 State Street to 1561 Northumberland Drive in anticipation of an upcoming move.

Heather Martin called to complain because her sister Elissa Martin received a form and she did not. After some checking, it became apparent that the wrong address was entered in the computer for Heather Martin. Heather's address was corrected.

Both Ann Davis and Loren Jones called and said they would be unavailable to compete in the 1990 season. They were kept as competitors in the system, but were no longer assigned to a team. This required a minor change to the class model to make team optional for a competitor.

The class model also requires extension to track contestant number. We decided to add another attribute to the *Competitor* class called *number* to accomplish this. Heather Martin was assigned number 3; Elissa Martin was assigned number 4; and Cathy Lewis was assigned number 1. Christine Brown was assigned number 5; Karen Solheim was assigned number 1; and Jane Smith was assigned number 9. All these numbers were the same as last year. Two competitors may have the same number as long as they are members of different teams.

13.20 Figure A13.13 shows the activity diagram.

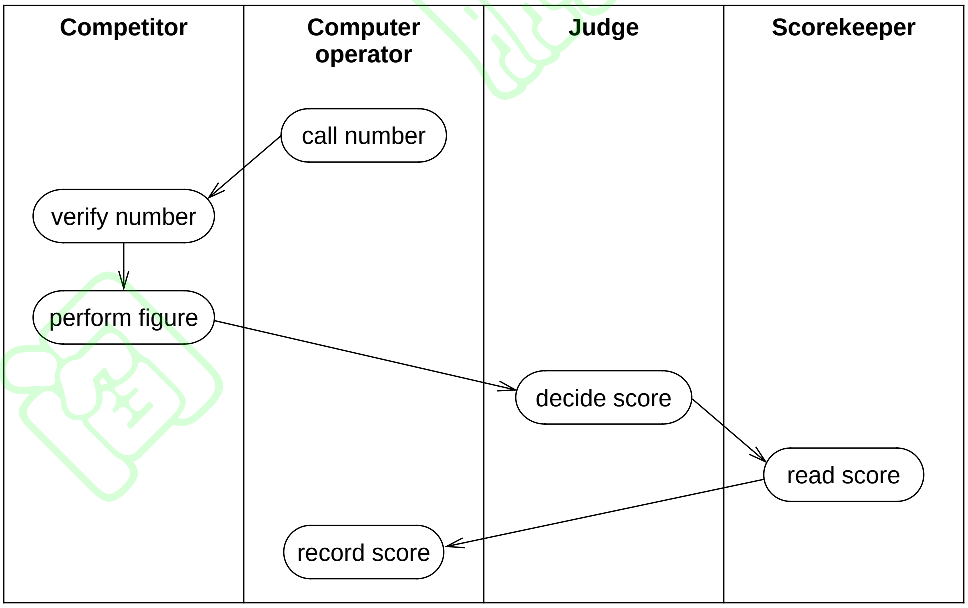


Figure A13.13 Activity diagram for recording swim scores

13.21 Figure A13.14 shows a partial shopping list of operations.

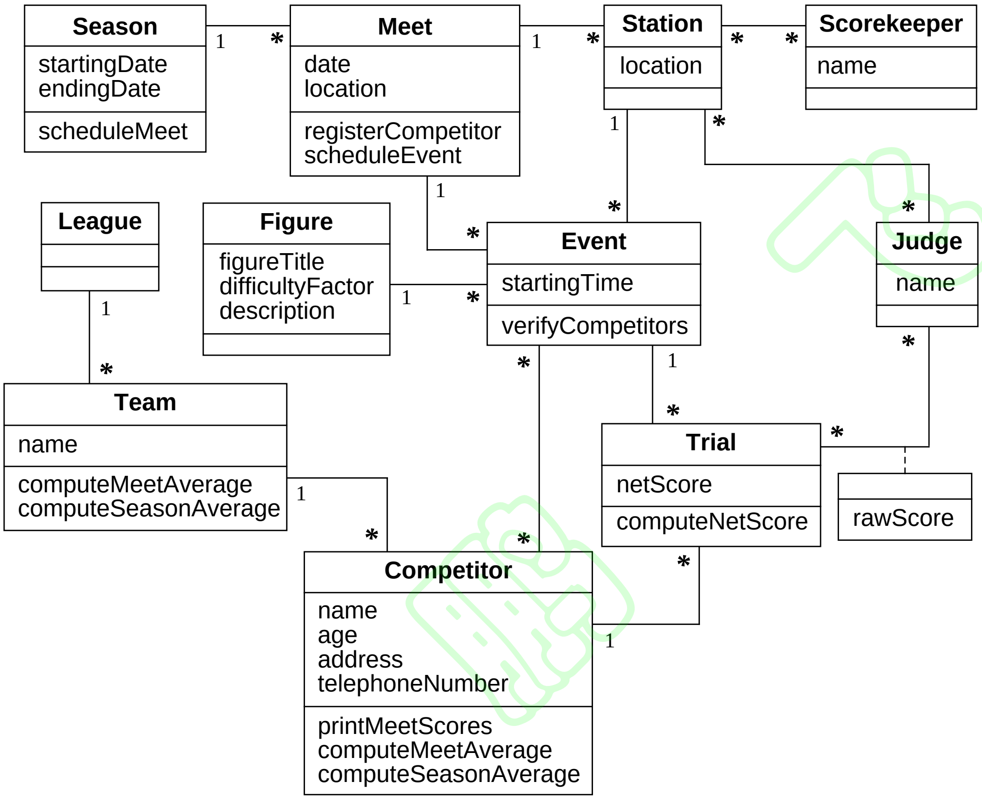


Figure A13.14 Partial class diagram for a scoring system including operations

13.22 The following bullets summarize what the operations should do.

- *scheduleMeet*. If the meet is not already known, add it to the system along with its required link. (The class model specifies that a meet must have exactly one season.) Specify a date and location for the meet.
- *registerCompetitor*. Given the name of a competitor, make sure that a *Competitor* object is entered into the system. Make sure that the competitor is assigned a team. Note that there is no explicit association between *Meet* and *Competitor*; thus we can make sure that a competitor is known to the system, but the model cannot specifically register a competitor for a meet. (Exercise 12.12l addresses this.)
- *scheduleEvent*. If the event is not already known, add it to the system along with its required links. (The class model specifies that an event must have exactly one figure, meet, and station.) Assign the event a starting time.

- *verifyCompetitors*. Make sure all attributes are filled in for each competitor. Make sure that each competitor is assigned a team.
- *computeNetScore*. For a given trial, retrieve the set of raw scores from the judges. Delete the high and low score. Average the remaining scores.
- *Team.computeMeetAverage*. For the given team, retrieve *team.competitor.trial.netScore*. For the given meet, retrieve *meet.event.trial.netScore*. Intersect these two sets of scores and compute the average of the resulting set. (Note that we have described the *computeMeetAverage* operation with a procedure. The operation need not be computed in the manner that we have specified but may be computed with any algorithm that yields an equivalent result.)
- *Team.computeSeasonAverage*. For the given team, retrieve *team.competitor.trial.netScore*. For the given season, retrieve *season.meet.event.trial.netScore*. Intersect these two sets of scores and compute the average of the resulting set.
- *printMeetScores*. For the given competitor and meet, retrieve the set of trials. For each trial, print the competitor name, meet date, meet location, and net score.
- *Competitor.computeMeetAverage*. For the given competitor and meet, retrieve the set of trials. Compute the average of the net scores for these trials.
- *Competitor.computeSeasonAverage*. For the given competitor and season, retrieve the set of trials. Compute the average of the net scores for these trials.

14

System Design

14.1a. An electronic chess companion combines features of interactive interface and batch transformation architectures. Entering and displaying moves is dominated by interactions between the chess companion and the human user. Once a human has entered a move, the activity of the chess companion is a batch transformation in which the positions of the pieces on the board are used to compute the next move.

b. An airplane flight simulator for a video game system combines features of interactive interface and dynamic simulation architectures. Interactive features include interpreting the joystick input and displaying cockpit information and the view of the terrain. Simulation involves computing the motion of the airplane. Another acceptable answer is interactive interface and continuous transformation, since computing the motion of the airplane is done by a continuous transformation of the joystick input.

c. A floppy disk controller chip is predominately a real-time system. With the possible exception of computing a cyclic redundancy code, there are practically no computations. Strict timing constraints must be satisfied to avoid losing data. The most important part of the design is the state diagram.

d. A sonar system is mainly a continuous transformation with some real time and some interactive interface aspects. Converting a continuous stream of pulses and echoes into range information is a continuous transformation. Because the system must process the data as fast as it comes in, there is a real-time flavor. Display of the range information involves interactive interface issues.

14.2a. Procedural control is adequate for the chess companion since the user interface is simple.

b. Event-driven control would be best for the flight simulator. Then it could accept push button events and a timing interrupt could pace the display update so that the apparent flight velocity does not change with the complexity of the calculations. Procedural control can be used, but the quality of the simulation would suffer. We assume that the terrain is static; otherwise we would consider concurrent control to handle the motion of other objects.

c. A floppy disk controller requires hardware control, not software control. Hardware controls are typically designed using state diagrams, which are relatively easy to convert into hardware. One way to do this is to use micro-coding. The state diagrams are converted into a procedure which in turn is implemented as a sequence of micro-instructions. Each output of the controller is typically controlled by one of the bit fields in an instruction. Instructions for branching and looping based on inputs and controller registers are usually available.

d. The real time aspect of the sonar system makes concurrent control desirable.

14.3 It is totally out of the question to store the data samples without buffering. For example, during the 10 milliseconds that the read/write head takes to move from one track to the next, 160 bytes would be lost. It is clear that a buffer is needed. We will discuss some of the constraints and control issues. Then we will present a couple of approaches.

A system architect would probably question the decision to limit the buffer size to 64000 bytes. Memory is cheap, and 160000 bytes would be enough to store all of the samples, greatly simplifying the design. However, within the context of this problem, 64000 bytes is a constraint.

Careful system design can reduce the size of the buffer. It is a good idea to store the data in sequential tracks to minimize the time spent in moving from one track to the next. Beyond that, there are special track formats to reduce the time lost moving between tracks. For example, if only a portion of a track is used or if the tracks are carefully arranged, the beginning of one track could be at the read/write head just as the head has completed an adjacent track.

Another dimension to consider is whether or not the buffer can be emptied while a track is being written. If it can, there is some hope of making the buffer only as large as required to store data that would be lost during the head motion between tracks. Otherwise the buffer may have to hold several tracks of data. First, estimate the rate that data can be written to the disk. The average time to find the beginning of a track, 83 milliseconds, is approximately equal to the time for the disk to turn one half of a revolution, so it takes about 166 milliseconds to make a complete turn. Therefore, the disk is revolving 6 times in a second. The number of bytes per track is approximately equal to the total storage, 243000 bytes, divided by the number of tracks, 77, or about 3155 bytes per track. (51 tracks are needed, leaving 26 tracks on the disk for other functions.) The rate at which data could be written to the disk is equal to the bytes per track times the spinning rate, or about 18935 bytes per second. This is greater than the rate of data input so there is some hope of sizing the buffer for a single track.

There are two control aspects to consider: initiating data storage and servicing the analog to digital converters and the disk drive.

We must assume that the initiation of data storage is to be controlled externally, because there is no reason for us to believe that our product will be able to control the events that are generating the data.

Another aspect of control is servicing the converters and the disk drive. Two common approaches are polling and interrupt driven. Polling involves continuously checking the status of both devices and transferring data as needed. This approach simplifies the hardware design at the expense of tying up the CPU during the capture of data. The interrupt driven approach relies on a hardware interrupt controller to notify the CPU when external devices need servicing, freeing up the CPU to do other tasks in the meantime. Our choice will depend on whether or not there is anything to do while data is being collected. It is also entirely possible that the interrupt driven approach has been selected to satisfy the requirements of other functions of the product, so we would want to explore further before making a decision.

We now analyze a couple of approaches:

- Sequential tracks, standard formatting with a buffer that holds several tracks

The simplest solution is to not use any special disk formatting, for which a worst case analysis is indicated. For each 3155 byte track that is written, it will take 10 milliseconds to position the head, up to 166 milliseconds for the beginning of the track to come around, and 166 milliseconds to write the track. During this time, 5488 bytes come in, so the buffer is filling faster that it can be emptied.

Without special formatting, it takes 342 milliseconds to complete a track. During the 10 seconds that it takes for all data to come in, a little over 29 tracks can be written. To be on the safe side, plan on writing 91495 bytes to the disk in 10 seconds. This leaves over 68000 bytes for the buffer, so this approach will not work unless we increase the amount of memory in the system.

- Sequential tracks, special disk formatting with a buffer for one track

If the disk is formatted carefully, the time to find the beginning of a track after completing an adjacent track can be reduced. This time will not be zero, because we will want some margin for error. The problem statement does not give any information about variations in disk speed, but 5 milliseconds (representing a 3 percent variation in disk speed) should give us sufficient margin for error. By careful formatting, the total time to write a track could be cut to 181 milliseconds. During this time, only 2896 bytes is input, which is less than a track, so the strategy could work. Because the system has no control over when the data starts coming in, it must be prepared to buffer one complete track of data. This approach will work with a buffer of 3155 bytes.

14.4 This system combines features of an interactive interface with a batch transformation. An interactive interface is needed to edit drawings. Converting a drawing to an N/C tape is a batch transformation. Determining an efficient sequence of drilling operations is a form of the traveling salesman problem. The optimum solution would take a long time to compute, and is not really needed for this application. There are several engineering solutions that produce a reasonably efficient sequence of operations without using a lot of computing resources. An excellent algorithm is given by John D. Litke in “An Improved Solution to the Traveling Salesman Problem with Thousands of Nodes”, *Communications of the ACM*, Volume 27, Number 12, December 1984, 1227–1236.

It is convenient to break the system into three major subsystems: editor, planner, and puncher. The editor could be subdivided into user interface, graphics display and file interface. Everything runs on a PC, so there is no question of assigning subsystems to hardware devices. The editor is used to prepare drawings. The planner determines a sequence of operations to accomplish the drilling. The puncher, which incorporates a driver for operating a punch, prepares an N/C tape.

There are no concurrency issues. A sequential approach is adequate. Experience has shown that a PC is adequate for two dimensional drafting.

Control must be considered for the editor and the puncher. Either procedural control or event driven control is suitable for the puncher. The editor should be event driven. Global resources are adequately handled by the operating system of the PC.

14.5 The interface is simplified by being line oriented, leaving the bulk of the work to be done on the batch transformations required to execute the commands.

There are three major subsystems: user interface, file interface and expression handling. There is no inherent concurrency. We assume that the system operates on a sufficiently powerful computer. Files are adequate for storing previous work. One design task is to devise a language for representing expressions. A procedural approach is adequate for control.

14.6 Figure A14.1 shows one possible partitioning.

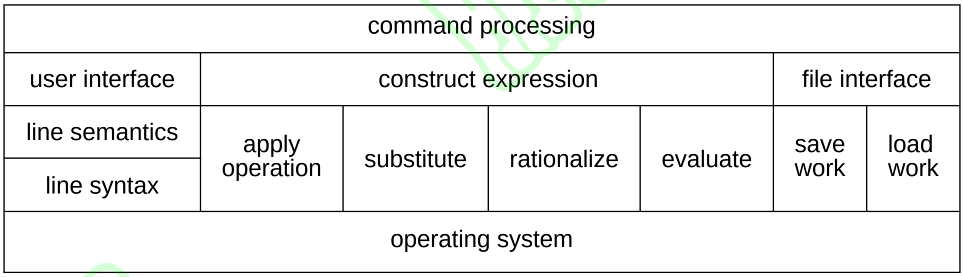


Figure A14.1 Block diagram for an interactive polynomial symbolic manipulation system

14.7 A single program provides faster detection and correction of errors and eliminates the need to implement an interface between two programs. With a single program, any errors that the system detects in the process of converting the class diagram to a database schema can be quickly communicated to the user for correction. Also, the editing and the conversion portions of the program can share the same data, eliminating the need for an interface such as a file to transfer the class diagram from one program to another.

Splitting the functionality into two programs reduces memory requirements and decouples program development. The total memory requirement of a single program would be approximately equal to the sum of the requirements of two separate programs.

Since both programs are likely to use a great deal of memory, performance problems could arise if they are combined. Using two separate programs also simplifies program development. The two programs can be developed independently, so that changes made in one are less likely to impact the other. Also, two programs are easier to debug than one monolithic program. If the interface between the two programs is well defined, problems in the overall system can be quickly identified within one program or the other.

Another advantage of splitting the system into two programs is greater flexibility. The editor can be used with other back ends such as generating language code declarations. The relational database schema generator can be adapted to other graphical front ends.

14.8a. A single geometrical model completely describes a class diagram, but would be too low level a representation. It is tedious to construct class diagrams by drawing lines, boxes, text, and so forth. Deriving the logical model from the physical representation is possible, but is time consuming and ambiguous.

b. Two separate models. This is the best solution, because it decouples two separate aspects of a class diagram, making both aspects less cluttered and easier to understand during system design. The geometrical model is useful for preparing printouts. The logical model is useful for semantic checking and interacting with the backend programs which need to know what the diagram means but do not care about the precise manner in which it is drawn.

14.9 The fastest way to implement the system is to use a commercially available desktop publishing system to edit the class diagrams. This eliminates the need to design and implement interactive graphics software, a step that is time consuming and error prone. As a bonus, there will be probably be good user documentation for the desktop system and vendor support. The impact of future enhancements made by the vendor is uncertain. On the one hand, the added functionality may make the overall system more attractive. On the other hand, future releases of the commercial system may change the markup language.

Implementing your own editor will result in a more robust system because the system can be designed to allow the user to draw only valid class diagrams. With the desktop editor, it is possible to draw pictures which cannot be interpreted as class diagrams. Also, because your own editor can be customized to the semantics of class diagrams, it will be easier and faster to use than the general-purpose desktop publishing system.

14.10 Here is an evaluation of each solution.

a. **Do not worry about it at all. Reset all data every time the system is turned on**. This is the cheapest, simplest approach. It is relatively easy to program, since all that is needed is an initialization routine on power up to allow the user to enter parameters. However, this approach cannot be taken for systems which must provide continuous service or which must not lose data during power loss.

b. **Never turn the power off if it can be helped. Use a special power supply, including backup generators, if necessary**. This approach is used for critical systems which must provide continuous service no matter what. This kind of power supply is called an UPS (uninterruptable power supply) and is relatively expensive. A bonus of this approach is that the application program is simplified, since it is assumed to be always running.

c. **Keep critical information on a magnetic disk drive. Periodically make full and/or incremental copies on magnetic tape**. This approach is moderately expensive and bulky. In the event of a power failure, the system stops running. An operating system is required to cope with the disk and tape drive. An operator is required to manage the tapes, which would preclude applications where unattended operation is required.

d. **Use a battery to maintain power to the system memory when the rest of the system is off. It might even be possible to continue to provide limited functionality**. This solution is relatively cheap, compact, and requires no operator. The main drawback of this approach is having to remember to change the battery periodically. In many applications, the user may not be aware that there is a battery that needs to be changed.

e. **Use a special memory component**. This approach is relatively cheap and is automatic. However, the system cannot run when power is off. Some restrictions may apply such as a limit on the number of times data can be saved or on the amount of data that can be saved. A program may be required to save important parameters as power is failing.

f. **Critical parameters are entered by the user through switches**. This approach is well suited to certain types of data, such as options, that are set only once, and is relatively cheap. Usually only a limited amount of data can be entered this way. Of course, the program cannot save any data through this scheme.

14.11a.Four function pocket calculator. Do not worry about permanent data storage at all. All of the other options are too expensive to consider. This type of calculator sells for a few dollars and is typically used to balance checkbooks. Memory requirements are on the order of 10 bytes.

b. **Electronic typewriter**. Memory requirements are on the order of 10,000 to 100,000 bytes per document. The system does not need to function when power is off, but the work in progress should not be lost. Commercially available electronic typewriters typically use floppy disk drives, backup batteries, or special memory components. Each of these methods will preserve data for more than a year. The advantage of floppy disk drives is that they can store many documents. Backup batteries or special memory components cost less than disk drives, and are used in some of the less expensive models even though they have limited storage capacity. Some electronic typewriters use memory modules that can be removed from the typewriter. Memory modules are more expensive than floppy disks, but are more immune to the hazards that smoke, dirt, bending and liquid spills present to floppy disk drives, so are better suited to users who are not familiar with floppy disk drives.

c. **System clock for a personal computer**. Only a few bytes are required, but the clock must continue to run with the main power off. Battery backup is an inexpensive solution. Clock circuits can be designed that will run for 5 years from a battery.

d. **Airline reservation system**. The amount of data to be stored is very large. Use a combination of uninterruptable power supplies, disk drives, and magnetic tapes. This application requires that the system always is available. Expense is a secondary consideration.

e. **Digital control and thermal protection unit for a motor**. On the order of 10 to 100 bytes are needed. This application is sensitive to price. An uninterruptable power supply is too expensive to consider. Tape and disk drives are too fragile for the harsh environment of the application. Use a combination of switches, special memory components, and battery backup. Switches are a good way to enter parameters, since an interface is required anyway. Special memory components can store computed data. A battery can be used to continue operation with power removed but presents a maintenance problem in this application. We would question the last requirement, seeking alternatives such as assuming that the motor is hot when it is first turned on or using a sensor to measure the temperature of the motor.

14.12a. A description of the diagram, ignoring tabs, spaces, and line feeds, is:

```
(DIAGRAM
   (CLASS
      (NAME “Polygon”))
   (CLASS
      (NAME “Point”)
      (ATTRIBUTE “x”)
      (ATTRIBUTE “y”))
   (ASSOCIATION
      (END (NAME “Polygon”) ONE)
      (END (NAME “Point”) MANY)))
```

b. Data in storage and data in motion are similar in that they can convey the same information. The same format can be used to store data or to transmit it from one location to another. In fact, many operating systems provide services that work equally well on files or on streams of data. The answer to the first part of this problem could be interpreted as either a format for a file or for a data transmission. Both data in storage and data in motion have semantics and syntax that can be described using the same tools.

c. We give both production rules and a diagram for the language to describe polygons. Each production rule is a nonterminal, followed by ::=, followed by a (possibly empty) string of nonterminals and terminals. Nonterminals are lower case identifiers. The productions rules for the language are:

```
diagram ::= ( DIAGRAM list_of_polygons_opt );
list_of_polygons_opt ::=
                          | list_of_polygons;
list_of_polygons ::= polygon
```

```
                         | list_of_polygons polygon;
polygon ::= ( POLYGON list_of_points );
list_of_points ::= point
                 | list_of_points point;
point ::= ( POINT x y );
```

We have chosen to allow an empty list of polygons but a polygon must contain at least one point. Other definitions are possible. The nonterminals *x* and *y* are integers. The corresponding BNF is shown in Figure A14.2.

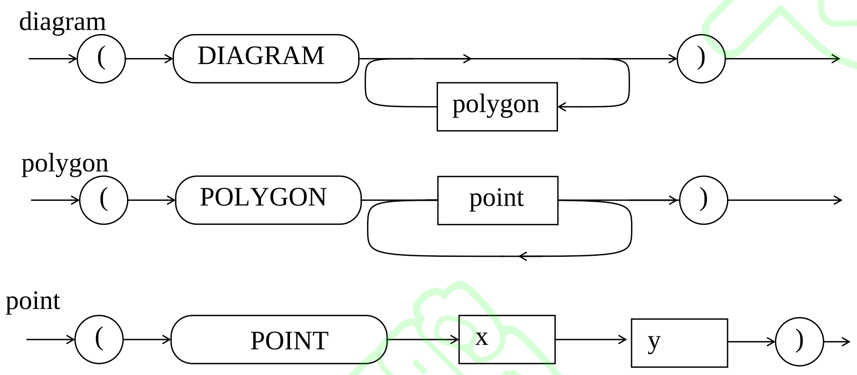


Figure A14.2 BNF diagram for a language to describe polygons

An example of a square in this language is:

```
( DIAGRAM
    ( POLYGON
        ( POINT 0 0 )
        ( POINT 10 0 )
        ( POINT 10 10 )
        ( POINT 0 10 )))
```

An example of a triangle in this language is:

```
( DIAGRAM
    ( POLYGON
        ( POINT 10 0 )
        ( POINT 10 10 )
        ( POINT 0 10 )))
```

In both cases the points are listed in the order of their connectivity.

14.13 The hardware approach is fastest, but incurs the cost of the hardware. The software approach is cheapest and most flexible, but may not be fast enough. Use the software approach whenever it is fast enough. General purpose systems favor the software approach, because of its flexibility. Special purpose systems can usually integrate the added circuitry with other hardware.

Actually, there is another approach, firmware, that may be used in hardware architectures. Typically, in this approach a hardware controller calculates the CRC under the direction of a microcoded program that is stored in a permanent memory that is not visible externally. We will count this approach as hardware.

a. **Floppy disk controller**. Use a hardware approach. Flexibility is not needed, since a floppy disk controller is a special purpose system. Speed is needed, because of the high data rate.

b. **Transmission over telephone lines**. Use a software approach. This system is a general purpose one, capable of running on a variety of computers. Data rates are relatively modest.

c. **Memory board in the space shuttle**. Use hardware to check memory. This is an example of a specific application, where the function can probably be integrated with the circuitry in the memory chips. The data rate is very high.

d. **Magnetic tape drive**. Use hardware or software, depending on the data rate and quality of the tape drive, and the application. If the drive is tightly integrated into a larger, specialized system, and the data rate is not too high, it may be cheaper to compute the CRC in software. On the other hand, if the tape drive is a general purpose one, or if the data rate is high, a hardware approach is better.

e. **Validation of an account number**. Use a software approach. The data rate is very low. The system handling the account number is probably running on a general purpose computer.)

14.14 Figure A14.3 extends the scheduler class models so that permissions can be assigned by group. An *AccessEntity* can belong to multiple groups. A *Group* can consist of multiple access entities. This kind of model leads to a directed graph and lets us recursively specify the composition of a group.

A schedule is owned by a user. Multiple entities can be given permission to access a schedule. There can be various kinds of permissions, such as read, insert, delete, and update. An individual user can access a schedule if he or she has direct permission or belongs to a group that has permission.

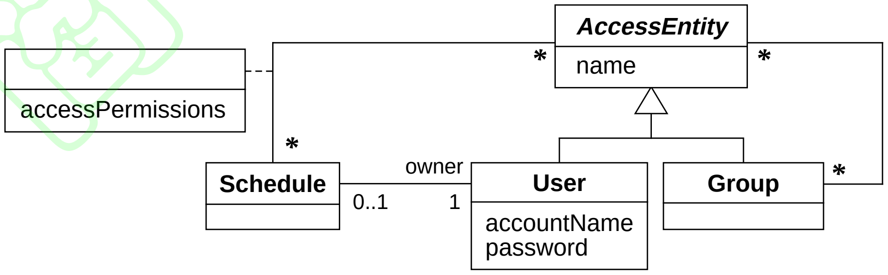


Figure A14.3 Class model for scheduler software extended for group permissions

15

Class Design

15.1 Here are responsibilities for the use cases.

- **Create drawing**. If there is an unsaved drawing, warn the user, and then clear the memory if the user confirms. Create a single sheet and set the current sheet to this sheet. Set the color to black and the line width to thin. Turn the automatic ruler on. Turn gravity off. (Gravity causes lines to connect to figures when the end of a line is near.)
- **Modify drawing**. Make the change as indicated by the user on the screen. Also update the underlying memory representation. Reset the undo buffer to recover from the modification. Set the *hasBeenUpdated* flag in memory. (A drawing with the *hasBeenUpdated* flag set cannot be quit without explicit user confirmation.) If the current sheet has been deleted, reset the current sheet to the prior sheet or to NULL if there are no longer any sheets.
- **Save to file**. Obtain the base filename from the user, if it is not already known. Delete the file with name *filename.bak*. Rename the existing file *filename.dwg* to *filename.bak*. Save the contents of memory to *filename.dwg*. Clear the *hasBeenUpdated* flag in memory.
- **Load from file**. If there is an unsaved drawing, warn the user, and then clear the memory if the user confirms. Find the file on the disc and complain if the file is not found. Build a drawing in memory that corresponds to the file. Display an error message if the file is corrupted. Display a warning if some of the grammar is unrecognized (can happen when an old release of software reads a file from a new release). Show a drawing on the screen that corresponds to the contents in memory. Set the current sheet to the first sheet.
- **Quit drawing**. If there is an unsaved drawing and the user declines to confirm, terminate *quit drawing*. Clear the contents of the screen. Clear the contents of memory. Clear the *undo* buffer.

15.2 Here are responsibilities for the use cases.

- **Register child**. Add a new child to the scoring system and record their data. Verify their eligibility for competition. Make sure that they are assigned to the correct team. Assign the child a number.
- **Schedule meet**. Choose a date for the meet within the bounds of the season. Ensure that there are no conflicting meets on that date. Secure access to the swimming pool for holding the meet. Determine the figures that will be held. Notify the league and participating teams of the meet. Arrange for judges to be there. Assign competitors to figures and determine their starting times. Assign scorekeepers and judges to stations.
- **Schedule season**. Determine starting and ending dates. Determine the leagues that will be involved. Schedule a series of meets that comprise the season. Check that each team in the participating leagues has a balanced schedule (same number of meets and same ratio of home and away meets). Reach agreement on the figures that can be chosen for meets. Obtain a pool of judges for staffing the meets.

15.3a. For a circle of radius *R* centered at the origin, we have $x^2 + y^2 = R^2$. Solving for *y*, we get $y = \pm\sqrt{(R^2 - x^2)}$. We can generate a point for each *x* coordinate by scanning *x* from -*R* to *R* in steps of one pixel and computing *y*. The center point of the circle must be added to each generated point. This simple algorithm is both inefficient and leaves large vertical gaps in the circle near $x = \pm R$ where the slope is great. More sophisticated algorithms compute 8 points at once, taking advantage of the symmetry about the axes, and also space the points so that there are no gaps. See a computer graphics textbook for details.

b. For an ellipse centered at the origin with axes *A* and *B* parallel to the coordinate axes, we have $(x/A)^2 + (y/B)^2 = 1$. We can solve for *y* in terms of *x* as for the circle. The equations are more complicated if the axes are not parallel to the coordinate axes. The same objections apply to this simple algorithm as to the previous one, and better algorithms exist.

c. Drawing a square is simply drawing a rectangle whose sides are equal. Unlike a circle, there is little advantage to treating a square as a special case. See Part d.

d. To draw a rectangle of width *2A* and height *2B* with sides parallel to the coordinate axes centered at (*X*, *Y*), fill in all pixels with ordinates *Y*-*B* and *Y*+*B* between abscissas *X*-*A* and *X*+*A*, and fill in all pixels with abscissas *X*-*A* and *X*+*A* between ordinates *Y*-*B*+1 and *Y*+*B*-1. If the rectangle is not parallel to the coordinate axes, then line segments must be converted to pixel values. This is more complicated than it seems because of the need for efficiency, avoiding gaps, and avoiding a jagged appearance (“aliasing”).

[This problem is either very easy or very hard. Since the problem was rated as a “4” then very simple answers such as the ones given here should be acceptable. In reality converting real-valued functions to pixels is a subtle and difficult topic because of the incompatibility of mapping real numbers into integers. It is covered at length in most graphics texts under the title of “scan conversion.”]

15.4 Certainly any algorithm that draws ellipses must draw circles, since they are ellipses, and any algorithm that draws rectangles must draw squares. The real question is whether it is worthwhile providing special algorithms to draw circles and squares. There is little or no advantage in an algorithm for squares, because both squares and rectangles are made of straight lines anyway. An algorithm for circles can be slightly faster than one for ellipses. This may be of value in applications where high speed is required, but is probably not worthwhile otherwise.

15.5 A general n-th order polynomial has the form $\sum_{i=0}^{n} a_i x^i$. Each term requires *i* multiplications and one addition (except the 0-th term), so computing the sum of the individual terms requires $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ multiplications and *n* additions. Computing the sum by successive multiplication and addition requires one multiplication and one addition for each degree above zero, or a total of *n* multiplications and *n* additions. The second approach is not only more efficient than the first approach, it is better behaved numerically, because there is less likelihood of subtracting two large terms yielding a small difference. There is no merit at all to the first approach and every reason to use the second approach.

15.6 Figure A15.1 enforces a constraint that is missing in Figure E15.1: Each *BoundingBox* corresponds to exactly one *Ellipse* or *Rectangle*. One measure of the quality of an class model is how well its structure captures constraints.

We have also shown *BoundingBox* as a derived object, because it could be computable from the parameters of the graphic figure and would not supply additional information.

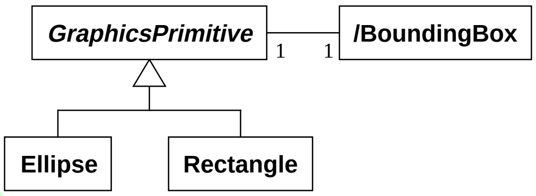


Figure A15.1 Revised class diagram for a bounding box

15.7 All of the classes will probably implement a *delete* method, but the *GraphicsPrimitive* class must define *delete* as an externally-visible operation. A typical application would contain mixed sets of *GraphicsPrimitive* objects. A typical operation would be to delete a *GraphicsPrimitive* from a set.The client program need not know the *GraphicsPrimitive* subclass; all that matters is that it supplies a delete operation.

Of course, *Ellipse* and *Rectangle* may have to implement *delete* as distinct methods, but they inherit the protocol from *GraphicsPrimitive*. Class *BoundingBox* must also define a *delete* operation, but this should not be visible to the outside world. A *BoundingBox* is a derived object and may not be deleted independently of its *GraphicsPrimitive*. *BoundingBox.delete* is visible only internally for the methods of *GraphicsPrimitive*.

15.8 In Figure A15.2 each page has an explicit *PageCategory* object that determines its size. Each *PageCategory* object has an associated *PageMargin* object that specifies the default margin settings. Any page can explicitly specify a *PageMargin* object to override the default settings.

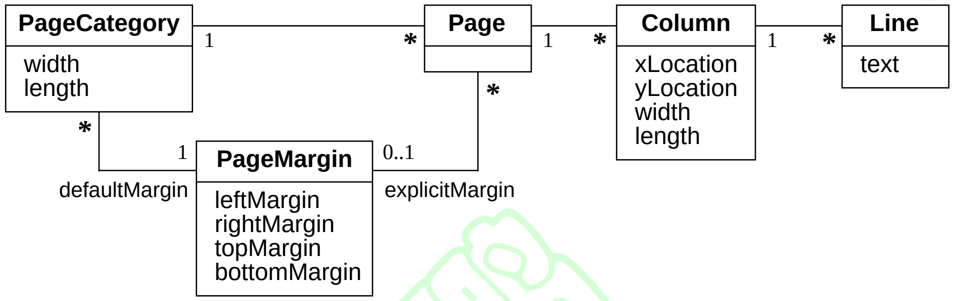


Figure A15.2 Portion of a newspaper model with a separate class for margins

15.9 The derived association in Figure A15.3 supports direct traversal from *Page* to *Line*. Derived entities have a trade-off—they speed execution of certain queries but incur an update cost to keep the derived data consistent with changes in the base data. The *Page_Line* association is the composition of the *Page_Column* and *Column_Line* associations.

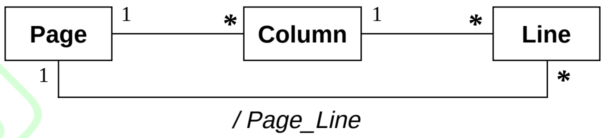


Figure A15.3 A revised newspaper model that can directly determine the page for a line

15.10 Note: We use OCL notation to traverse associations. We assume that there is a generic routine to sort arrays taking as an argument a comparison function between pairs of array elements. The comparison function returns LESS_THEN, EQUAL_TO, or GREATER_THAN.

```
Card :: display (aLocation: Point)
{
   Render a card on the screen at aLocation;
}

Card :: compare (otherCard: Card): ordering
   // Compares two cards by suit, then by rank
   // suits and ranks are ordered according to their
      enumeration values
{
   if self.suit < otherCard.suit then return LESS_THAN
   else if self.suit > otherCard.suit then return
      GREATER_THAN
   else if self.rank < otherCard.rank then return LESS_THAN
   else if self.rank > otherCard.rank then return
      GREATER_THAN
   else return EQUAL_TO;
}

Card :: discard (aDrawPile: DrawPile)
{
    aCardCollection := self.CardCollection;
    aCardCollection.delete (self);
    aDrawPile.insert (self);
}

CardCollection :: initialize ()
{
   // This method is inherited by all classes except Deck
   Clear the set self.Card
}

Hand :: insert (aCard: Card)
{
   add aCard to self.Card according to sort order of cards;
   shift the images of any cards to the right of the insert
      point one position to the right;
   display the image of the inserted card;
}

Hand :: delete (aCard: Card)
{
   delete aCard from self.Card;
   if nothing was deleted, then error and return;
   // we assume that a hand is displayed from left-to-right
   // starting at the given location
   erase the image of the deleted card;
   shift images of any remaining cards one position to left;
}
```

```
Hand :: sort ()
{
   sort array self.Cards using Card::compare;
}

Pile :: topOfPile (): Card
{
   if the pile is empty, then return null;
   return self.Card.(last);
   // The set of Cards is ordered so we can get the last one
}

Pile :: bottomOfPile (): Card
{
   if the pile is empty, then return null;
   return self.Card.(first);
   // The set of Cards is ordered so we can get the first one
}

Pile :: draw (aHand: Hand)
{
   aCard := self.topOfPile();
   self.delete (aCard);
   aHand.insert (aCard);
}

Pile :: insert (aCard: Card)
{
   // insertions onto piles go on the top
   append aCard to end of self.Card;
   aCard.display (self.location, self.visibility);
}

Pile :: delete (aCard: Card)
{
   if aCard ≠ self.topOfPile() then error and return;
   delete aCard from self.Card;
   topcard := self.topOfPile();
   topcard.display (self.location, self.visibility);
   // the new top picture overwrites the old one
}

Deck :: shuffle ()
{
   ncards := number of cards in self.Cards;
   generate a random permutation of size ncards;
   rearrange self.Cards using the random permutation;
}
```

```
Deck :: deal (nhands: Integer, hsize: Integer): Array of Hand
   // Deals nhands hands of hsize cards each.
   // Returns the hands. Any leftover cards remain in deck.
{
   if nhands * hsize > self.size(), then error and
      return null;
   create and initialize array of nhands empty Hand objects;
   for isize := 1 to hsize do:
         for ihand := 1 to nhands do:
            delete top card from deck and add it to hand
               ihand;
   return the array of hands;
}

Deck :: initialize ()
{
   // This method overrides the inherited method.
   // The deck still needs to be shuffled explicitly.
   Clear the set self.Card;
   for each aSuit in the enumeration Suit do:
      for each aRank in the enumeration Rank do:
         create aCard := Card (aSuit, aRank);
         add aCard to self.Card;
      end;
   end;
   visibility := BACK;// assume deck is face down
   location := defaultDeckLocation;
   // place deck on a default screen location
}
```

15.11 Here is the pseudocode.

```
Trial::computeNetScore ()
   Scan all rawScores for a trial finding the sumOfScores,
      minimumScore, maximumScore, and numberOfScores;
   If numberOfScores <= 2, report an error and stop;
   adjustedSum := sumOfScores - minimumScore - maximumScore;
   averageScore := adjustedSum / (numberOfScores - 2);
   return averageScore * self.Event.Figure.difficultyFactor;
```

15.12a.

```
Figure::findEvent (aMeet: Meet) : Event;
   return self.Event INTERSECT aMeet.Event;
   // Note that 0, 1, or more events could be returned.
```

b. When the event is held, the *RegisteredFor* association is used to generate the list of competitors, in the order of registration. After each competitor competes in the event, a trial is created and scored for the competitor.

```
Competitor::register (anEvent: Event);
   If self.Event is not null then return;
      // already registered
   create new RegisteredFor link between self and anEvent
```

c.

```
Competitor::registerAllEvents (aMeet: Meet)
   For each anEvent in aMeet.Event
      self.register(anEvent);
```

d. We presume that scheduling is manual. Someone must decide which events are to be held at which meets. To track the decisions, we need an operation:

```
Meet::addFigure (aFigure: Figure)
   Create an event object.
   Associate it with the meet.
   Associate it with the figure.
```

We will not assign starting times until all the figures are selected. A simple scheduling algorithm assuming equal blocks of time for each event is:

```
Meet::scheduleAllEvents ()
   while events have not been scheduled do
      for each aStation in self.Station do
         anEvent := get next unscheduled event in
            self.Event
         if no more events then return;
         associate anEvent to aStation;
```

A more sophisticated algorithm would take into account the number of competitors registered for an event and the average time required for a trial of the given figure.

e. We again assume that scheduling is manual and that what is wanted is an operation to keep track of decisions. We also assume that simultaneous meets are not held.

```
Season::addMeet (aDate: Date, aLocation: Location)
   If there is already a meet on the date, report error
   Create a new Meet object with the date and location
   Associate the Meet object to the Season self
```

f. We assume that a set of available judges and a set of available scorekeepers is provided. We also assume that there is a minimum and a maximum number of judges and scorekeepers permitted for a station (the two numbers may be the same). We assume that there is some priority rule for selecting personnel if there are too many volunteers for a meet.

```
Meet::assignPersonnel (judges, scorekeepers)
   Sort the judges and scorekeepers in priority order;
   averageJudges := numberOfJudges / numberOfStations;
   averageScorekeepers :=
      numberOfScorekeepers / numberOfStations;
   if averageJudges < minimumPermitted then error;
   if averageScorekeepers < minimumPermitted then error;
   neededJudges:= minimum (averageJudges, maximumPermitted);
   neededScorekeepers :=
      minimum (averageScorekeepers, maximumPermitted);
```

```
for each station:
   assign the next neededJudges judges and
      neededScorekeepers scorekeepers.
Notify any remaining judges and scorekeepers that they are
   not needed.
```

15.13 The code listed below sketches out a solution. This code lacks internal assertions that would normally be included to check for correctness. For example, error code should be included to handle the case where the end is a subclass and the relationship is not generalization. In code that interacts with users or external data sources, it is usually a good idea to add an error check as an else clause for conditionals that “must be true.”

```
traceInheritancePath (class1, class2): Path
{
   path := new Path;
// try to find a path from class1 as descendent of class2
   classx := class1;
   while classx is not null do
      add classx to front of path;
      if classx = class2 then return path;
      classx := classx.getSuperclass();
// didn’t find a path from class1 up to class2
// try to find a path from class2 as descendent of class 1
   path.clear();
   classx := class2;
   while classx is not null do
      add classx to front of path;
      if classx = class1 then return path;
      classx := classx.getSuperclass();
   // the two classes are not directly related
   // return an empty path
   path.clear();
   return path;
}

Class::getSuperclass (): Class
{
   for each end in self.connection do:
      if the end is a Subclass then:
         relationship := end.relationship;
         if relationship is a Generalization then:
            otherEnds := relationship.end;
            for each otherEnd in otherEnds do:
               if otherEnd is a Superclass then:
               return otherEnd.class
   return null;
}
```

15.14 Figure A15.4 shows the revised model. The revised model is more precise with the associations between the subclasses; the additional associations also are more cumber-

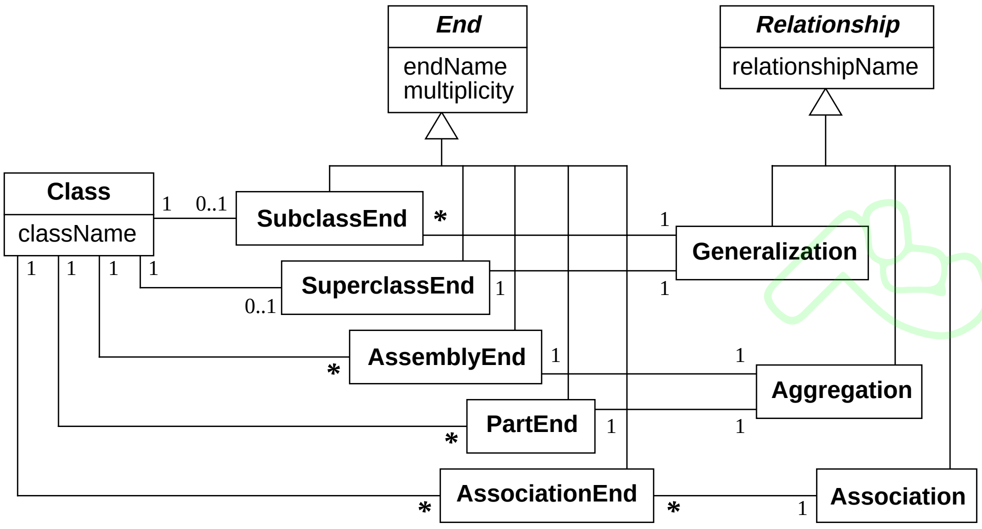


Figure A15.4 A revised model with associations to the subclasses

some. The appropriate model would depend on application needs and developer preference. The *traceInheritancePath* method is the same as in the previous answer. This is a good example of modular code, as a change to the model affected only one method.

```
Class::getSuperclass (): Class
{
   subEnd := self.subclassEnd;
   if subEnd = null then return null;
   superclass :=
      subEnd.generalization.superclassEnd.class;
   return superclass;
}
```

15.15 We make the following assumptions: The length of a name is unrestricted, names consist of alphanumerics and underscores, input names do not contain double underscores but generated association names may have double underscores. The algorithm:

```
If the association has a relationshipName,
   then return the name,
else do
   Get the two ends in the association
   Get the two class names from the classes in the ends
   Sort the two class names in alphabetical order
   Form a string class1.className & “__” & class2.className
If the string is unique among explicit association names
   then return the string
```

```
Else form the string class1.className & “__” &
   end1.uniqueEndName() & “__” & class2.className & “__” &
   end2.uniqueEndName() and return the string
```

(The operator & indicates string concatenation.)

15.16 Figure A15.5 shows the revised model. Political party membership is not an inherent property of a voter but a changeable association. The revised model better represents voters with no party affiliation and permits changes in party membership. If voters could belong to more than one party, then the multiplicity could easily be changed. Parties are instances, not subclasses, of class *PoliticalParty* and need not be explicitly listed in the model; new parties can be added without changing the model and attributes can be attached to parties.

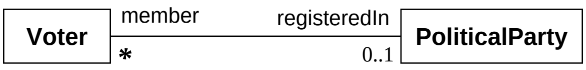


Figure A15.5 A revised model that reifies political party

15.17 The algorithm may best be expressed as two coroutines, one which scans the passengers requiring reassignment and the other which scans the empty seats to reassign. A coroutine is a subroutine that keeps its internal state and internal location even after a call or return statement.

```
oldRowCount = number of rows in the old plane
newRowCount = number of rows in the new plane
// By specification, newRowCount < oldRowCount

reassignPassengers ()
   for each row from newRowCount+1 to oldRowCount do
      for each seat in a row from left to right
         if a passenger is assigned to the seat do
            newSeat := getEmptySeat()
            if newSeat ≠ null
               then reassign it to the passenger
            else put the passenger on standby

getEmptySeat (): seat
   for each row from 1 to oldRowCount do
      for each seat in a row from left to right do
         if no passenger is assigned to the seat
            then return it
         else go on to the next seat
   report a seat shortage error
   return null to indicate we have run out of seats
```

15.18 The left model in Figure A15.6 shows an index on points using a doubly-qualified association. The association is sorted first on the *x* qualifier and then on the *y* qualifier. Because the index is an optimization, it contains redundant information also stored in the *Point* objects.

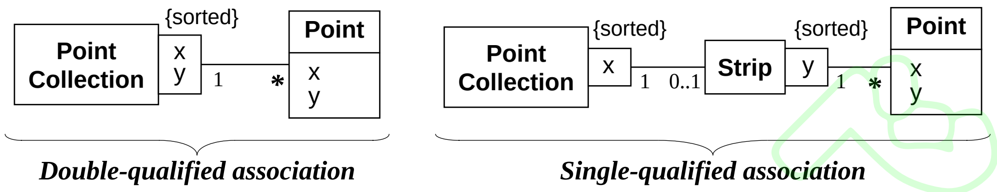


Figure A15.6 Models for sorted collections of points

The right model shows the same diagram using singly-qualified associations. We introduced a dummy class *Strip* to represent all points having a given x-coordinate. The right model would be easier to implement on most systems because a data structure for a single sort key is more likely to be available in a class library. The actual implementation could use B-trees, linked lists, or arrays to represent the association.

The code listed below specifies search, add, and delete methods.

```
PointCollection::search (region: Rectangle): Set of Point
{
   make a new empty set of points;
   scan the x values in the association until x ≥ region.xmin;
   while the x qualifier ≤ region.xmax do:
      scan the y values for the x value until y ≥ region.ymin;
      while the y qualifier ≤ region.ymax do:
         add (x,y) to the set of points;
         advance to the next y value;
      advance to the next x value;
   return the set of points;
}

PointCollection::add (point: Point)
{
   scan the x values in the association until x ≥ point.x;
   if x = point.x then
      scan the y values for the x value until y ≥ point.y
   insert the point into the association at the current
      location;
}

PointCollection::delete (point: Point)
{
   scan the x values in the association until x ≥ point.x;
   if x = point.x then
      scan the y values for the x value until y ≥ point.y
      if y = point.y then
```

```
              for each collection point with the current x,y value
                 if collection point = point
                    then delete it and return
        report point not found error and return
    }
```

Note that the *scan* operation should be implemented by a binary search to achieve logarithmic rather than linear times. A scan falls through to the next statement if it runs out of values.

15.19 The analysis is a bit complicated. We can look at several different cases.

There is an initial search in x of cost = log (number of columns containing points) ≤ log N, where N is the total number of points. We must search every column between the top and bottom of the target rectangle. Within each column, there is an initial search cost of log (number of points in the column). There is no additional cost within a column, except to scan out the points within the target rectangle, but there is no waste in doing this, because we stop after the first non-output point. So the total cost is log (#columns) + (#columns in rectangle) * log (#points in column) + #output points. Note that the middle term is really a sum over the column because the number of points in the column is not constant, but you get the idea.

We can ignore the first term, which will be small compared to the other terms for most practical cases. We will also separate out the final term, which is the number of output points and represents the useful work, and concentrate on the wasted searches represented by the middle term. For an algorithm like this, we really want to know the ratio between the wasted work and the useful work. The time will vary a lot depending on the number of output points and not just N, so it is not productive to characterize it by just N.

In the worst case, every point will be on a different column. If the target rectangle spans the entire y dimension, the total cost will be N, the number of points. To make matters worse, if the rectangle is wide and thin, it might contain no points at all, so the yield is 0 output points. The cost per output point is infinite! (Of course, this is a bit unfair, because the cost per point of any algorithm is infinite in this case.) This algorithm is good for tall narrow rectangles but not so good for short wide ones.

In a medium case, assume that points are randomly but sparsely distributed, with less than one point per column on average. Then we essentially end up searching all the points between the left and right lines of the rectangle. The x-ordering helps us avoid searching unnecessary columns, but the y-ordering within the column doesn't help because there is only a single point per column. On average, the fraction of points that fall within the rectangle is equal to its height divided by the total height of the search space, so the cost per yield point is the reciprocal of that fraction. It is still bad to have short rectangles.

In a dense case, assume that there are many points per column, Pcolumn, on average in the search space and the target rectangle contains many points per column, Phit, although Phit may be much less than Pcolumn. Then the waste of searching each column is log Pcolumn for a yield of Phit, so the waste is small provided Pcolumn < exp(Phit).

For example, assume 100 points per column, then log(100) = 7 is the waste per column; a target rectangle that is 10% of the height will yield 10 points, so the waste is less than the cost and an order of magnitude better than searching all the points.

What can we say about this algorithm? It is highly asymmetric with respect to the x and y axes, and not very good if the target rectangles will be horizontal lines. It is also not very good if there are not many points per column, but in that case we can increase the interval of x-values per column so that more points are included. In general, we would like the column interval to be approximately the width of the rectangle, so that we need to scan as few columns as possible yet do not scan unnecessary points. Then the waste will be fairly small. If the shapes of the target rectangles vary a lot, then this algorithm will not always be so good.

To summarize, ordering points and using binary search works well for one-dimensional searches, but not so well for two-dimensional searches, because the points may be ordered first on the wrong dimension, reducing the search to a linear search. The proper approach to 2-d searching is not to nest two 1-d searches but to build a new 2-d algorithm, called a quadtree. Details can be found in graphics texts.

15.20a. In the worst case, if we search the wrong way first by chance, then the search will go to the top of the class hierarchy, so the cost is linear in the depth. If we modify the algorithm so that we search up from both classes at the same time, then the search cost will be no more than twice the difference in depth of the two classes and will not depend at all on the depth of the class hierarchy (except as a limit on the difference in depths). If the class hierarchy is very deep, then the algorithm should be modified to limit the cost, otherwise the added complexity is probably not worth the bother.

b. The algorithm we described is proportional to the number of seats in the old plane (we must scan the “missing” seats for passengers to reassign and the “common” seats for potential reassignment). It does not depend at all on the number of passengers. Any attempt to be more clever is misplaced zeal because this algorithm is fast enough for any practical purpose.

16 Process Summary

[There are no exercises in Chapter 16.]

Part 3: Implementation

17 Implementation Modeling

17.1 An arrow indicates that the association is implemented in the given direction. Because of the large amounts of data for an ATM we used two-way pointers for associations that are traversed in both directions—we avoided one-way pointers combined with backward searching. Note that most of the associations in the domain model may be traversed either way, but associations for the application model are traversed only one way.

- **ATM <–> RemoteTransaction**. This association would be implicit for the ATM. However, it would be needed for the consortium and bank computers. Given a transaction we must be able to find the ATM. Given an ATM we might want to find all the transactions for a summary report.
- **RemoteTransaction <–> Update**. Obviously we must be able to find the updates for a transaction. However, we also must be able to go the other way. We might find the updates for an account and then need to find the transactions, such as to get the date and time.
- **RemoteTransaction <–> CardAuthorization**. Given a transaction, we might want to find how it was authorized. Given an authorization we might want to see all of its activity (such as for a fraud investigation).
- **RemoteTransaction <– RemoteReceipt**. Given a receipt, we should be able to find the transactions. (This might happen if a customer brings in a receipt and has questions.) We presume that the receipt is unimportant to the bank, and is just an artifact for the customer's benefit, so we do need to traverse from transaction to receipt.
- **RemoteReceipt –> CashCard**. The receipt should indicate which cash card was used. We presume there is no need to traverse from a cash card to a receipt. (Note, however, that we can traverse from cash card to authorization to transaction.
- **CashCard <–> CardAuthorization**. A cash card must know its authorization. An authorization must also know about its cash cards.

- **Bank <–> CardAuthorization**. An authorization must be able to reference its bank and a bank must be able to find all authorizations (to facilitate issuing of card codes).
- **Consortium <–> ATM**. The consortium must track all of its ATMs and the ATM must know its consortium.
- **Consortium <–> Bank**. Same logic as the prior bullet.
- **Bank <–> Account**. A bank must know all its accounts. An ATM must be able to find the bank for a card authorization's account.
- **Customer <–> Account**. An account must be able to retrieve customer data. A customer must be able to look up all of his or her accounts.
- **Customer <–> CardAuthorization**. A customer must be able to find their authorizations and an authorization must be able to find customer data.
- **CardAuthorization <–> Account**. A card authorization may cover only some of a customer's accounts. So a card authorization must be able to find those accounts. An account may need to find the card authorizations to which it is subject.
- **Account <–> Update**. An account must be able to tally its updates. An update must know which account is affected.
- **CashCardBoundary <– AccountBoundary**. This association is purely an artifact for the convenience of importing and exporting data. Consequently, there is no need to implement the association in both directions. We will choose to have pointers that retrieve cash card data for an account. (The decision on which way to implement is arbitrary and we could do it the other way.
- **TransactionController –> RemoteTransaction**. A transaction controller must know about its transactions. There is no need to go the other way.
- **ATMsession –> CashCard**. An ATM session must know which cash card is involved. There is no need to go the other way, since the ATM session is transient and there is at most one active per ATM machine.
- **ATMsession –> Account**. Same logic as previous bullet.
- **SessionController –> ATMsession**. A session controller has at most one ATM session active at a time, but has many ATM sessions over time. The session controller must be able to find the current, active ATM session. There is no need to go the other way.
- **SessionController –> ControllerProblem**. The session controller must be able to find its problems. There is no need to go the other way.
- **ControllerProblem –> ProblemType**. We must be able to find the type for a controller problem, but do not need to go the other way.

17.2 An arrow indicates that the association is implemented in the given direction.

- **Text <–> Box**. The user can edit text and the box must resize, so there should be a pointer from text to box. Text is only allowed in boxes, so we presume that a user may grab a box and move it, causing the enclosed text to also move. So there should be a pointer from box to text.

- **Connection <–> Box**. A box can be dragged and move its connections, so there must be pointers from box to connections. Similarly, a link can be dragged and move its connections to boxes, so there must also be a pointer from connection to box. There is no obvious ordering.
- **Connection <–> Link**. Same explanation as above bullet.
- **Collection –> ordered Box**. Given a collection we must be able to find the boxes. There does not seem to be a need to traverse the other way. There likely is an ordering of boxes, regarding their foreground / background hierarchy for visibility.
- **Collection –> ordered Link**. Same explanation as above bullet.

17.3 The answer depends on whether the model is used only to generate output (going from a document to the individual lines) or whether the model is also used to handle input (the user picks a line at random and operates on it). For the first case, the associations are likely to be traversed in only one direction, from page to column and from column to line. In the latter case, it would be best to implement each association in both directions.

The lines within a column should be ordered. We need to know more about the problem to determine if the columns on a page should be ordered. If text is to flow from page to page when changes are made, columns should be ordered. Otherwise there is little advantage to having ordered columns.

17.4 This association would likely be traversed in both directions. Operations on a collection of cards, such as *insert*, require access to the ordered set of cards. On the other hand, the user may be able to select a card from the screen, in which case the collection containing the card must be found.

17.5 The simplest approach is to implement every association in both directions. In the rest of this answer we indicate when a one-way implementation of an association might be feasible. Note that the choice of one-way or two-way traversal depends on the details of an application and that some of our decisions may be arguable. An arrow indicates that the association is implemented in the given direction. Association ends are unordered, except where specified.

- **Season –> ordered Meet**. Probably want to order meets by date. Probably do not need to go from meet to season.
- **Meet –> Station**. There is no reason to order stations nor any basis to do so. An implementation might introduce station numbers but they have no intrinsic meaning. Probably do not need to go from station to meet.
- **Station <–> Scorekeeper**. Need to go both ways. Need to know which scorekeepers are assigned to a station. Each scorekeeper needs to know their assigned stations.
- **Station <–> Judge**. Same comments as previous bullet.

- **Station <–> ordered Event**. Probably want to order events by time because each station can support only one event at a time. We will probably want to query an event to see what station it is on.
- **Meet <–> Event**. Because a meet (unlike a station) may have simultaneous events, ordering is not so useful. We may need the back pointer to answer questions such as “Which meets did a competitor participate in during a season?”
- **Figure <–> ordered Event**. We need to go both ways; an event needs to know its figure, and we may want to know the meets at which a figure is held. Events could be ordered by meet date.
- **Event <–> ordered Trial**. We will need to go both ways. The trials are held in a specific sequence that is important for scheduling.
- **Trial –> (Judge, rawScore)**. No need to order the scores. No reason to go backwards from judge to (Trial, rawScore), unless possibly we need to audit each judge’s performance.
- **Competitor <–> ordered Trial**. Obviously a competitor can only do one thing at a time, and the trials are ordered by time.
- **League <–> Team**. Need to go both ways, no obvious ordering.
- **Team <–> Competitor**. Same as previous bullet.

18

OO Languages

Chris Kelsey not only authored Chapter 18 in the text, but also prepared most of these answers. Brian Blaha also prepared some of the answers.

18.1 Figure A18.1 and Figure A18.2 add C++ and Java data types for each attribute. Measurements are real number types to allow fractions of inches/centimeters. The C++ *string* requires use of the Standard Template Library. The Java *String* is the class java.lang.String.

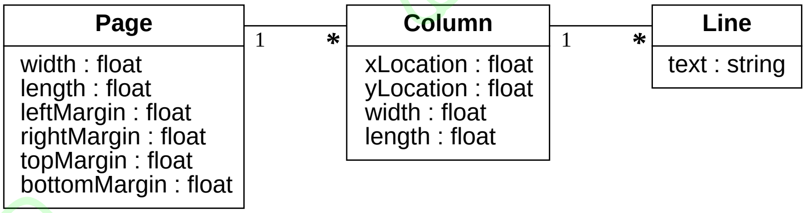


Figure A18.1 Portion of a class diagram of a newspaper with C++ data types

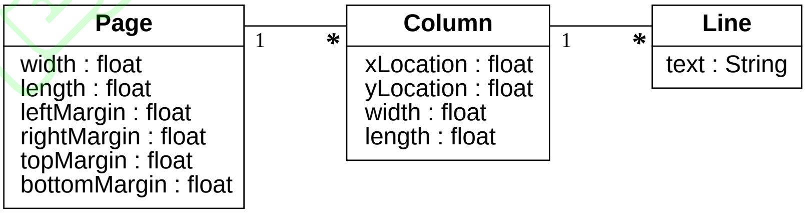


Figure A18.2 Portion of a class diagram of a newspaper with Java data types

18.2 Here are C++ declarations.

```
class Card {
};
class CardCollection {
};
class Hand : public CardCollection {
};
class Pile : public CardCollection {
};
class Deck : public Pile {
};
class DiscardPile : public Pile {
};
class DrawPile : public Pile {
};
```

18.3 Here are C++ classes. We omit *CardCollection.location* because a better model would handle that aspect in a separate graphics rendering layer. Note that the enumerations are public and the attributes are private. We implement the *CardCollection—Card* association by adding an attribute for each class.

```
class Card {
public:
   enum Suit { CLUB, DIAMOND, HEART, SPADE };
   enum Rank { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
      NINE, TEN, JACK, QUEEN, KING, ACE };
   enum CompareResult { LESS_THAN, EQUAL_TO, GREATER_THAN };
      //FOR 18.5
   CompareResult compare(Card& otherCard) const; // 18.5
   void discard(DrawPile& drawpile); // 18.5
private:
   Suit suit;
   Rank rank;
   CardCollection* cc; // current card collection
};

#include<list> // for collection, 18.4

class CardCollection {
public:
   enum Visibility { FRONT, BACK };
   virtual void initialize();
      // virtual to override in Deck 18.5
   virtual void insert(Card& card) = 0;
   virtual void discard(Card& card) = 0;
protected:
   Visibility visibility;
   // Point location;
   list<Card*> cardsInCollection; // exercise 18.4
};
```

```
class Hand : public CardCollection {
   int initialSize;
public:
   void insert(Card& card); // 18.5
   void discard(Card& card); // must override pure virtual
   bool deleteCard(Card& card); // "delete" is c++ keyword
   void sort();
};

class Pile : public CardCollection { // still abstract class
public:
   Card* topOfPile();
      // return pointer to card, so can be null, 18.5
   Card* bottomOfPile();
   void draw(Hand& aHand);
   void insert(Card& card);
   bool deleteCard(Card& card);
};

class Deck : public Pile {
public:
   void initialize(); // override to "no-op" 18.5
   void shuffle();
   Hand* deal( int nhands, int hsize );
   void discard(Card& card); // must override pure virtual
};

class DiscardPile : public Pile {
public:
   void discard(Card& card); // must override pure virtual
};

class DrawPile : public Pile {
public:
   void discard(Card& card); // must override pure virtual
};
```

18.4 See answer to Exercise 18.3.

18.5 See answer to Exercise 18.3.

18.6a. One to one association traversed in both directions. The solutions presented in the book (see p. 465) are extended here.

Several issues arise with bidirectional one-to-one associations. It is essential to build in safeguards to ensure that both sides of a link are linked or unlinked symmetrically, especially at construction and destruction of objects. Implementations differ according to whether links are mandatory or optional, whether they are changeable or of object-lifetime duration, and whether linking itself is a public operation or whether constraints dictate when or which objects may be linked.

The Java example in the book places emphasis on type independence. Classes *A* and *B* reside in different packages, and each uses a flag to terminate mutually recursive calls for linking and unlinking. Linking and unlinking operations are publicly available. The C++ example in the book allows *A* and *B* to accept requests to link (e.g. `A::setB(B& newB)`), but unlinking is performed only in the context of setting a new link, and is conducted through friendship. The linking object accesses the private data member of its linkee and directly manipulates it (e.g. `b->a = this;`).

In the extended versions we have assumed the link is 1) not mandatory (and so objects may be constructed without linking) and is 2) changeable, such that an object *A* may be linked to different objects of type *B* at different times, and 3) linking and unlinking can be publicly requested. Note the linking and unlinking method names have been changed to promote matching syntax across the classes.

In the extended C++ version, both *A* and *B* can accept public requests to link or unlink to their respective associates. The *A* side is responsible for actually executing the link; if a *B* is asked to do so, it delegates the duty to its prospective *A*. *A* has a small, private method that wraps the actual link exchange; the wrapper encapsulates the specifics of pointer exchange and can serve as an lockable block in a threaded environment. Note that copying and assignment are prohibited: To allow either (without redefining the default semantics of the operations) would allow the possibility of multiple links to a single target object. The destructor ensures unlinking to prevent dangling pointers in objects that outlive their linked partners. For unlinking, we've demonstrated a mutually recursive technique that ensures both sides detach, but without a need for private access.

The extended Java example parallels the C++. The book example’s fine-grained packaging is relaxed—associated classes typically share a package, which allows friendship-like access by default. It is important that the class author tighten default access with explicit private specifiers to ensure collaborative classes do not remain unguarded against the ability to excessively and randomly reach into one another. Java does not allow object copying by default nor assignment by value, and so does not require the construction and assignment safeguards needed in C++.

In both, we've included dummy data and data accessor methods, should the reader wish to readily implement clients.

```
C++:

#include <string>
using namespace std;

class B;

class A {

   // prohibit copying/assigning A's to prohibit
   // multiple A's linked to same B...

   A(const A&);
```

```
      A& operator=(const A&);

      inline void exchangeLinks(B& aB);
         // A side has responsibility
         // of executing the link on both sides
      string data;
      B* b;

   public:

      A(const char* dataStr) : data(dataStr), b(0) {}
      ~A() { UnLink(); } // ensure unlinking at destruction

      bool hasB() { return b != 0; }

      string Data() { return data; }
      B* myB() { return b; }

      inline string BData();
         // define after B implementation known

      inline bool Link(B& aB); // ensure bidirectional link...
      inline bool UnLink(); // and bidirectional unlinking too!
   };

   class B {

      // friendship declared by B grants to A::exchangeLinks the
      // privilege of access to B's non-public members, i.e. it
      // "disencapsulates" B for A only in the context of A's
      // link-exchanging operation.

      friend void A::exchangeLinks(B& aB);

      B(const B&);
      B& operator=(const B&);

      string data;
      A* a;

   public:

      B(const char* dataStr) : data(dataStr), a(0) {}

      ~B(){ UnLink(); }

      string Data() { return data; }
      bool hasA() { return a != 0; }

      bool Link(A& anA) {
         if (a || anA.hasB() ) return false; // already linked
         return anA.Link(*this);
```

```
    }

    bool UnLink() {
       if (!a) return false; // nothing to unlink!
       A* aptr = a; a = 0;
       return aptr->UnLink();
    }

       // navigate to A data through B, or
       // allow B to expose it's linked A publicly...

    string AData(){ return a ? a->Data() : ""; } // someB.AData()
    inline A* MyA() { return a; } // someB.MyA->Data();
};


// deferred A implementations:

string A::BData() { return b->Data(); }

bool A::Link(B& aB) {
    if (b || aB.hasA() ) return false; // already linked
    exchangeLinks(aB); return true;
}

bool A::UnLink() {
    if (!b) return false; // nothing to unlink!
    B* bptr = b; b = 0;
    return bptr->UnLink();
}

void A::exchangeLinks(B& aB) { b = &aB; aB.a = this; }

int main()
{ ... }
```

Java:

```
// A.java

package oneone;

public class A {

    public A(String dataStr) { data = dataStr; }

    public boolean IsLinked(){ return b != null; }

    public String Data() { return data; }
    public B myB() { return b; }
```

```
      public String BData(){ return b.Data(); }

      public boolean Link(B aB){
         if (b != null || aB.IsLinked() ) return false;
            // already linked
         exchangeLinks(aB); return true;
            // ensure bidirectional link...
      }

      public boolean UnLink(){
         if ( b == null ) return false; // nothing to unlink!
         B bptr = b; b = null;
         return bptr.UnLink(); // and bidirectional unlinking too!
      }

      private void exchangeLinks(B aB){
         // A does the linking on both sides
         b = aB; aB.a = this;
            // B's x is package-access, not private
      }

      private String data;
      private B b = null;
   };


   // B.java

   package oneone;

   public class B {

      public B(String dataStr) { data = dataStr; }

      public String Data() { return data; }
      public boolean IsLinked() { return a != null; }

      public boolean Link(A anA) {
         if (a != null || anA.IsLinked() ) return false;
            // already linked
         return anA.Link(this); // let A do it
      }

      public boolean UnLink() {
         if (a == null) return false; // nothing to unlink!
         A aptr = a; a = null;
         return aptr.UnLink();
      }

         // navigate to A data through B, or
```

```
        // allow B to expose it's linked A publicly...

    public String AData(){ return a != null ? a.Data() : ""; }
    public A myA() { return a; }

    private String data;

    A a = null; // note package-level access instead of private!
    // or, could have private data + package-access
    // "mutator" method
}
```

b. Unordered one-to-many association traversed from the one to the many.

In navigating from one to many, note that the *Collection* is aware of its *Items*, but the *Items* have no awareness that they may be in a *Collection*. This presents a problem in the destruction of objects: If an *Item* is destroyed while linked into a *Collection*, the *Collection* will retain a dangling pointer to an *Item* that no longer exists.

To avoid these dangers, a robust implementation would exert some control over the creation and destruction of *Items* (i.e. the *Collection* might manage the lifecycles of *Items*, or a managing class would oversee the links), or *Items* would have links to *Collections* in which they appear. The dangers of dumb *Items* must be weighed against the additional dependencies created in having *Collections* manage *Items* or using bidirectional links, or the complexity of adding manager classes.

To more clearly demonstrate the logistics of links, we've elected to forgo the additional code that management would entail.

C++'s Standard Template Library includes the <*set*>. The set precludes inclusion of duplicates. A multiset is provided to implement bags.

Here, the one *Collection* may contain links to many *Items*:

```
#include <iostream> // for ListEm() debugging method

#include<string>
#include<set>
using namespace std;

class Item {

public:
    Item(const char* str) : data(str) {}
    string Data() { return data; }

private:
    string data;
};

class Collection {

public:
    bool Add(Item& item) { return items.insert(&item).second; }

    bool Remove(Item& item) {
       return(items.erase(&item) != 0);
    }

    bool InCollection(Item& item) {
       set<Item*>::iterator it = items.find(&item);
       return it != items.end();
    }

    Item* Retrieve(const char* dataStr) { // find by value
       /* c++'s STL employs value semantics -- containers retain
          copies of inserted objects. Here, we create a set that
          retains the pointer values of the inserted items
          rather than a set<Item> that would create copies of
          entire Items. Typically, a class encapsulating a
          collection of pointers provides a method to retrieve
          a pointer based on a data key. We leave this as a
          exercise for the reader. See also exercise 18.6d.
       */
    }

    // for debugging -- print item data values to console
    void ListEm() {
       set<Item*>::iterator it = items.begin();
       while (it != items.end())
          cout << (*it++)->Data() << endl;
    }

private:
    set<Item*> items;
};
```

Java: The issue of dangling pointers does not apply in Java. When an object is added to a Java *Collection*, the system creates a reference to it. The object will not be destroyed and garbage-collected until all references to it are destroyed. However, programmers may unwittingly keep objects alive if they are stored and forgotten in a Java *Collection*. (Note: *Collection* is a Java interface; the *Coll* in this example corresponds to *Collection* in the C++ version of the exercise.)

```
// Coll.java
package onemany;

import java.util.*;

public class Coll {

   private Set<Item> items = new HashSet<Item>();
```

```
      public boolean Add(Item item) { return items.add(item); }

      public boolean Remove(Item item) {
         return(items.remove(item));
      }

      public boolean InCollection(Item item) {
         return items.contains(item);
      }

      public void ListEm() {
         Iterator<Item> it = items.iterator();
         while (it.hasNext())
            System.out.println(it.next().Data());
      }
   }

   // Item.java
   package onemany;

   public class Item {

      public Item(String str) { data = str; }
      public String Data() { return data; }

      private String data;
   }
```

c. Ordered one-to-many association traversed from the one to the many.

The inheritance structure of the C++ STL dictates that all containers are manipulated in similar fashion. The implementation of the ordered association would be essentially the same as the above, but users can impose an order on a <*set*> by specifying that order as an argument to the template instance. Alternatively, one might use the <*list*>, with similar syntax, but with the additional responsibilities of coding to locate the appropriate insertion point for an object/pointer.

See Exercise 18.6d, where the *ShareHolder* side demonstrates ordering of its many *Horses*. Supporting code demonstrates template support for determining the order of objects through their pointers.

Java's *Collections* framework distinguishes between the *Set* and *SortedSet* interfaces. To be eligible for inclusion in a *SortedSet*, a class *E* must implement the *Comparable* interface to describe the natural order of a user-defined type, or a *Comparator<E>* which describes the comparison that may be specified as an argument to the constructor of the implementing *TreeSet<E>* class, much as the C++ set can be constructed with a predicate object. See 18.6d.

d. Many-to-many, ordered one way and unordered the other.

Consider racehorse syndication. A horse can have many owners who hold shares in the horse, and an owner may invest in many horses. The following code demonstrates M:M where objects directly maintain links. No association class is used.

Sets are used for both Java and C++. Although the mathematical definition of set implies an unordered collection, both languages supply standardized library routines to impose and maintain order on sets in efficient time.

These simplified implementation assume insert/removal of a *Horse* in a *ShareHolder's* collection of *Horses* is successful; if not, we would need to undo insert/erase of *ShareHolder* within *Horse*. Because changes in *Horse* ownership can occur only through the (publicly available) *buy* and *sell* methods of the *ShareHolder*, updates occur only from the *ShareHolder* side (after all, a *Horse* does not determine its own ownership!). Therefore, we'll assume if the *Horse* accepts the *ShareHolder* change at the request of the *ShareHolder*, the *ShareHolder* can and will accept the update of its *Horses*. In circumstances where updates can be triggered from either side of an association, methods on both sides ensure the entire transaction is acceptable and complete on both sides.

```
#include <iostream>
#include<string>
#include<set>

using namespace std;

class ShareHolder;
class Horse {

   friend class ShareHolder;

   // to enforce buyer/seller control, these are private...
   bool AddShareHolder(ShareHolder& sh)
      {return shs.insert(&sh).second; }
   bool RemoveShareHolder(ShareHolder& sh)
      {return shs.erase(&sh) != 0;}

public:

   Horse(const char* nm): name(nm) {}
   string Name() const { return name; }
   void ListShareHolders();

   // specifies < for ordering of set
   bool operator<(const Horse& rhs) const
      { return name < rhs.name; }

private:

   string name;
   set<ShareHolder*> shs;
};
```

```
// Define a function-object to compare T's by pointer...
// Works with any class that implements operator <
// See class Horse, where < is implemented as alphabetical order

template<class T>
struct lessPtr : public less<T> {
   bool operator()(const T* const lhs, const T* const rhs) const
   { return *lhs < *rhs ;}
};

class ShareHolder {

public:

   ShareHolder(const char* nm) : name(nm) {}
   string Name() { return name; }

   bool Buy(Horse& horse)
      { return horse.AddShareHolder(*this) ?
      horses.insert( &horse).second : false ;
   }
   bool Sell(Horse& horse)
      { return horse.RemoveShareHolder(*this) ?
      horses.erase( &horse) != 0 : false;
   }

   // Typical C++ implementation of a derived attribute...
   int HorseCount() { return horses.size(); }

   void ListHorses();

private:

   string name;
   set<Horse*,lessPtr<Horse> > horses; // set order specified!!
};

void Horse::ListShareHolders()
{
   cout << "\n" << Name() << "'s owners:\n";
   set<ShareHolder*>::iterator it = shs.begin();
   while ( it != shs.end() ) cout << "\n " << (*it++)->Name();
   cout << endl;
}

void ShareHolder::ListHorses()
{
   cout << "\n" << Name() << "'s horses:\n";
   set<Horse*, lessPtr<Horse> >::iterator it = horses.begin();
   while ( it != horses.end() )
      cout << "\n " << (*it++)->Name();
   cout << endl;
}
```

Java:

```
//Horse.java

package syndicate;
import java.util.*;

/* We do not use the Comparator here, but instead rely on the
Horse's implementation of the Comparable interface, which is
expressed by the compareTo method. see notes in 18.6c
*/

public class Horse implements Comparable<Horse> {

   // to enforce buyer/seller control, these are package
   // access.
   boolean AddShareHolder(ShareHolder sh)
      { return shs.add(sh); }
   boolean RemoveShareHolder(ShareHolder sh)
      { return shs.remove(sh); }

   public Horse(String nm){ name = nm; }
   public String Name() { return name; }

   // implementing Comparable interface here:
   public int compareTo( Horse rhs ) {
      return Name().compareTo( rhs.Name()) ;
   }

   public void ListShareHolders()
   {
      System.out.println( Name() + "'s owners:");
      Iterator<ShareHolder> it = shs.iterator();
      while (it.hasNext())
         System.out.println(it.next().Name());
      System.out.println();
   }

   private String name;
   private Set<ShareHolder> shs = new HashSet<ShareHolder>();
}

// ShareHolder.java

package syndicate;
import java.util.*;

public class ShareHolder {
   public ShareHolder(String nm) { name = nm; }
   public String Name() { return name; }

   // Simplified implementation -- see C++ note

   public boolean Buy(Horse horse) {
      return horse.AddShareHolder(this) ?
      horses.add(horse) : false ;
   }
   public boolean Sell(Horse horse) {
      return horse.RemoveShareHolder(this) ?
      horses.remove(horse) : false;
   }

   public int HorseCount() { return horses.size(); }

   public void ListHorses() {
      System.out.println( Name() + "'s horses:");
      Iterator<Horse> it = horses.iterator();
      while (it.hasNext())
         System.out.println(it.next().Name());
      System.out.println();
   }

   private String name;
   private SortedSet<Horse> horses = new TreeSet<Horse>();
}
```

18.7a. The system should deallocate strings as they are no longer needed. The methods that modify strings could perform the deallocation. One way to combine strings is as follows.

```
Determine total size of the strings that are to be combined.
Allocate enough memory for the result.
Copy the original strings into the allocated memory.
Deallocate memory assigned to the original strings.
```

We assume that the old strings are no longer needed.

b. (1) **Forego garbage collection and rely on virtual memory**. The programmer does nothing and lets the operating system manage memory. Without garbage collection, virtual memory performance is bound to degrade over time. If, however, objects tend to be accessed and created sequentially, this approach may be acceptable, because references to contiguous objects will cluster and limit memory swapping.
(2) **Recover memory for each compiler pass**. You can allocate a large block of memory for all the objects in a pass and deallocate it all at once. Application software must manage the block of memory for each compiler pass.

c. You cannot let the operating system allocate a large amount of virtual memory and forget about garbage collection in systems that run indefinitely. Eventually all of the memory will be consumed. Long-running software must deallocate memory for objects that are no longer referenced. Consider writing your own memory allocation/deallocation function to replace the library function, taking the characteristics of your own objects into account.

d. The first approach works only in special circumstances in which the lifetime of the object is short. It is a dangerous approach that builds many assumptions into the code. A safer approach is for the caller to explicitly destroy the object.

18.8 Although Java packages affect access control, the primary logical purpose of packages is to group together classes that are used together, particularly those that have interdependencies among themselves. Applications can then readily import all related classes with a reasonable number of statements, rather than having to know and declare each and every class that will be required. To prevent unintentional dependencies among classes, it is important to specify access control as appropriate for each package member's individual data and methods rather than simply allow Java's default-when-unspecified package-level access control. Failure to privatize allows accidental package-wide disencapsulation and unwitting dependencies, while failure to publicize can produce a useful class that is virtually unusable by clients outside its package.

Package Competitors: Competitor, Team, League
Package Competition: Season, Meet, Event, Figure, Station
Package Scoring: Trial, Judge, Scorekeeper

18.9 The exercise specifies that all associations are bidirectional, requiring updates in both directions as links change. In this application, there are situations where certain updates should be prohibited or controlled. For example, the deletion of a *Figure* implies the deletion of an *Event* for that *Figure*, which implies the deletion of all *Trials* in that *Event*. This is not acceptable (at least under normal circumstances) for *Events* where the *Trials* have already occurred. Access control can help constrain unacceptable operations by encapsulating possibly dangerous methods from some or all callers. We demonstrate such techniques here; see comments in the code. We omit declarations for routine maintenance and queries to more clearly show the relevant operations. Skeleton implementation is shown for certain methods.

Java declarations would be similar to the C++ below, but packaged as described in Exercise 18.8. Although Java does provide garbage collection, the programmer must detach all links to ensure the object can be destroyed. Operations found in C++ destructors would be executed in a “regular” Java method to ensure full and timely unlinking of objects. In Java, actual destruction takes place under the control of the garbage collector (see a detailed Java reference for an explanation of Java object lifecycles), and so logical end-of-object-life operations may not coincide with the physical destruction of the object.

C++: In most cases, we have used a <*set*> to maintain the many side of an association. When queries are expected, the <*set*> offers efficient order/search facilities. See Exercise 18.6d for a demonstration of imposing order on a <*set*> (omitted here).

```
#include<string>
#include<list>
#include<set>

using namespace std;

// forward declarations and assumed utility class decl's omitted

// Figure has many events; events involve a single figure
class Figure {
   string figureTitle;
   float difficultyFactor;
   string description;
   set<Event*> events; // events in which figure appears

// The model dictates that Figures can stand alone, but each
// Event must specify a Figure. The destruction of any Figure
// that appears in an Event should propagate to the Event, which
// in turn dictates the destruction of any Trials hat have
// already performed in that Event...

// To ensure scores persist, then, Figures should only be
// destroyed if they have no links to Events, or links only to
// destructible Events (events that have no scored Trials). To
// enforce this, we privatize destruction and control it
// through a static member function that implements the
// appropriate constraints.

// note that a private destructor forces dynamic creation of
// Figures, i.e. Figure* f = new Figure(...); not Figure f(...)

~Figure() { /* delete events dependent on this figure */ }

public:
   Figure(const char* title, float factor, const char* descrip) :
      figureTitle(title), difficultyFactor(factor),
      description(descrip) {}

   static bool DeleteFigure(Figure& fig) {
      // control delete-figure constraints here... e.g.
      if (fig.events.size() == 0) { delete &fig; return true; }
      // ...
      return false;
   }

   // add event to figure's events
   bool AddEvent(Event& ev);
   // other maintenance and query methods
};

// Competitor has many trials
class Competitor {
   string name;
   int age;
   Address addr; // presumes Address utility class
   Phone telephoneNumber; // presumes Phone utility class
   list<Trial*> trials; // this competitor's trials
public:
   Competitor(const char* nm) : name(nm) { }

   bool AddTrial(Trial& t); // add trial to competitors list...

   // other maintenance methods...
   // query methods...
};

// utility class; although a struct's members are public by
// default, instances will remain encapsulated within a Trial
struct RawScore {
   float score;
   Judge* judge;
   RawScore(Judge& j) : score(0), judge(&j) {}

   // required for set<RawScore> as any set must be "orderable".
   bool operator<(const RawScore& rhs) const;
};

// a Trial is the participation of a single competitor in a
// single Event and has * judges associated with it...
class Trial {
   friend class Event; // allow Event to manage Trials

   float netScore;
   Competitor* who;
   Event* event;
   set<RawScore> scoring; // judge-score pairs

   // Trials exist only in the context of Events, so we delegate
   // the responsibility of creating and removing them to their
   // Event. Friendship ensures that only Events can call
   // Trial's ctor/dtor. See Event::AddTrial
   Trial(Competitor& comp, Event& ev) : who(&comp) {}
   ~Trial() { /* delete trial from competitor's and
                each judge's trial lists... */ }

public:
   void computeNetScore();
   bool AddJudge(Judge& j);
   // maintenance and query methods
};

// A judge participates in * trials, and assigns a single
// raw score per trial, i.e. judge and rawscore is 1:1, while
// trial and judge-score-pair is 1:Many
```

```
class Judge {
   friend class Trial; // Trial::AddJudge inserts trial
                       // in Judge's trial roster
   string name;
   set<Trial*> trialsJudged;

public:
   Judge(const char* nm) : name(nm) {}
   string Name() const { return name; }
   // query methods
};

inline bool Trial::AddJudge(Judge& j) {
   scoring.insert(*new RawScore(j));
   j.trialsJudged.insert(this);
   return true;
}

// an event is a group of trials (competitor preformances)
// over a particular figure
class Event {
   Time startingTime; // presumes a Time utility class
   Figure& figure; // reference member is a permanent link --
      // cannot change figures for event
      // trials in this event:
      // because Trials can exist only in the context of an
      // Event, we would typically use a list of instances, not
      // pointers, but the privatized Trial destructor forces
      // dynamic allocation to obtain pointers for explicit
      // deletion. Events will manage the construction and
      // destruction of their Trials.

   list<Trial*> trials; // trials in this event

// see Figure note on private destructor. Events should not
// be deleted if linked to scored trials.
~Event() {
   // delete this event from figure's event registry
   // cascade-delete trials dependent on this event
}

public:
   // an event is created for a specific figure...
   Event(Figure& fig, Time& start) : figure(fig),
      startingTime(start) { fig.AddEvent(*this); }

   static bool DeleteEvent(Event& ev) {
      /* ensure any linked trials are unscored; if ok, delete
      trials, then event */
   }

   bool AddTrial(Competitor& who) {
```

```
            trials.push_back(new Trial(who), this);
        }

        bool DeleteTrial(Competitor& who);
            // interface to Trial, e.g.
            // GetTrial(someone)->AddJudge(judgename);
            Trial* GetTrial(Competitor& who);
    };
```

18.10a. Both the C++ and Java examples below include simple implementations of the pseudocode from Exercise 15.11. Additional methods are implemented as needed for support or testing. C++ Note: Many of the methods here would be defined out of line; they are defined within the class declarations (unless declaration order prohibits it) for ease of reading.

```
#include<iostream>
#include<list>
#include<set>
#include<algorithm>
#include<iterator>
#include<string>

using namespace std;

#include <cstdlib>
#include <ctime>

class Random { // supports shuffle


public:
    Random() { srand( (unsigned)time( NULL )); }
        // seed generator
        // 0 <= i < max + 1
    int getRandom(int maxplus1) { return rand() % maxplus1; }
};

template<class T>
struct lessPtr : public less<T> { // supports sorting hands

    bool operator()(const T* const lhs, const T* const rhs)
        const { return *lhs < *rhs ;}
};

class Card;

class CardCollection;

class Pile;
class Card {
    friend class CardCollection; // collection sets card's cc
    static const char * SuitStr[]; // support console display
    static const char * RankStr[];

public:
    enum Suit { CLUB, DIAMOND, HEART, SPADE };
    enum Rank { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
        NINE, TEN, JACK, QUEEN, KING, ACE };
    enum CompareResult { LESS_THAN, EQUAL_TO, GREATER_THAN };
    Card(Rank r, Suit s) : suit(s), rank(r) { }

    // exercise specifies enum order as criterion for <. == , >
    CompareResult compare(const Card& otherCard) const {
        if (index() == otherCard.index()) return EQUAL_TO;
        return ( index() < otherCard.index() ) ? LESS_THAN :
           GREATER_THAN ;
    }

    // for ordering. < operator qualifies class for use with STL
    bool operator<(const Card& otherCard) const {
        return compare(otherCard) == LESS_THAN;
    }
    bool operator>(const Card& otherCard) const {
        return compare(otherCard) == GREATER_THAN;
    }

    // support dummy console display
    string Name() const {
        return string(RankStr[rank]) + string(SuitStr[suit]);
    }

    // dummy display for console output...
    void display() const { cout << Name() << " "; }

    // remove self from current collection;
    // insert in destination pile
    // delegates work to card's current collection;
    // defined after Pile
    void discard(Pile& destination);

    // supports testing..., not required
    // bool IsIn(const CardCollection& coll) const
        { return cc == &coll; }

private:
    int index() const { return suit * 13 + rank; }
    void setCollection(CardCollection* coll) { cc = coll; }

    Suit suit;
    Rank rank;
```

```
      CardCollection* cc;
   };

   // define statics
   const char * Card::SuitStr[] = {"C","D","H","S" };
   const char * Card::RankStr[] = {
         "2","3","4","5","6","7","8","9","T","J","Q","K","A" };

   class CardCollection {
   public:
      enum Visibility { FRONT, BACK }; // NOT USED HERE...
      typedef list<Card*> CC;

      // default "initialize" clears out collection
      // beware MEMORY LEAK!! don't clear out collections
      // without having stored the Card*'s elsewhere...
      // client application should have TrashPile to collect cards
      // for destruction...
      virtual void initialize() {
         cards.erase( cards.begin(), cards.end() );
      }

      // insert and discard/delete renamed to insertCard and
      // deleteCard to avoid confusion with c++ insert/delete and
      // Card::discard by default, add at top
      virtual void insertCard( Card& card ) {
         cards.insert(cards.begin(), &card);
            setCardCollection(card);
      }

      // error/exception not implemented.
      // could return "false" if card not found...
      virtual void deleteCard(Card& card) {
         CC::iterator it =
            find(cards.begin(), cards.end(), &card) ;
         if (it != cards.end() ) {
            cards.erase(it); clearCardCollection(card);
         }
      }

      // encapsulate implementations; expose utilities
      bool empty() { return cards.size() == 0; }
      int size() { return cards.size(); }

      // simple output of collection onto console
      void Show() {
         cout << endl;
         CC::iterator it = cards.begin();
         while (it != cards.end()) (*it++)->display();
      }

   protected:
```

```
   Visibility visibility;
   // Point location;
   CC cards;

   // subs invoke base-level, befriended method to set card's
   // collection at insert. friendship is not inherited, so
   // overrides cannot access Card::setCollection

   void setCardCollection(Card& card)
      {card.setCollection( this ); }

   void clearCardCollection(Card& card)
      {card.setCollection( 0 ); }
};

class Hand : public CardCollection {
   int initialSize;

public:
   void insertCard(Card& card) {
      CC::iterator it = cards.begin();
         // find insert point
      while (it != cards.end() && **it < card ) it++;
      cards.insert(it,&card);
      setCardCollection(card); /*display...*/;
   }

   // no need to override except to add display.
   void deleteCard(Card& card){
      CardCollection::deleteCard(card) ; /*display()...*/;
   }

   // sort() renamed to avoid STL name confusion...
   // inserts are done in sort order, so this won't make any
   // changes...

   void sortCards() {
      // we can't use the STL's sort or list::sort because types
      // being compared must match the list item type, and we
      // can't define a comparison for built-in types -- which
      // all pointer types are! so... we'll make a <set> out of
      // the hand and copy it back to the list:

      set<Card*,lessPtr<Card> > tempset;
      CC::iterator it = cards.begin();
      while (it != cards.end()) tempset.insert(*it++);
      cards.erase( cards.begin(), cards.end() );
         // clear the list
      set<Card*,lessPtr<Card> >::iterator sit =
         tempset.begin();
      while (sit != tempset.end()) cards.push_back(*sit++);
   }
```

```
};

class Pile : public CardCollection {
public:
   // note: top of pile is head of list, not last in list...
   Card* topOfPile() { return empty() ? 0 : *cards.begin();}

   Card* bottomOfPile() {
      if (empty()) return 0;
      CC::iterator it = cards.end(); return *--it;
   }

   // draw from Pile into Hand
   void draw(Hand& aHand) {
      Card* card = topOfPile();
      if (!card) return; // return false...
      deleteCard(*card);
      aHand.insertCard(*card); // return true...
   }

   // no need to override except display;
   // CardCollection inserts are at top

   // void insertCard(Card& card) {
      CardCollection::insertCard(card) ; /*display()*/; }


   // remove only top card...
   void deleteCard(Card& card){
      if (&card != topOfPile()) return;
      CardCollection::deleteCard(card) ; /*display()*/;
   }
};

void Card::discard(Pile& destination){
   cc->deleteCard(*this);
   destination.insertCard(*this);
}

class Deck : public Pile {
public:
   Deck() { initialize(); } // constructor builds a deck...

   Hand* deal( int nhands, int hsize ) {
      if (nhands * hsize > cards.size()) return 0;
         // not enough cards!
      Hand* hands = new Hand[nhands]; // create the hands
         // conventional "around the table" deal
      for (int j = 0; j < hsize; j++ )
         for (int i = 0; i < nhands; i++)
            draw(hands[i]);
               // draw from deck to appropriate hand
            return hands;
        }

        // swap cards in existing deck N times
        void shuffle(int nTimes = 5) {
            CC::iterator it,it2;

            for (int i = 0; i < nTimes; i++ ) {
                it = cards.begin();
                while ( it != cards.end()) {
                    it2 = cards.begin();
                    advance(it2,r.getRandom(52));
                    swap(*it++,*it2);
                }
            }
        }

        void initialize() {
            CardCollection::initialize();
                // clear what's left; see note at base
            Card* card;
            for (int i = 0; i < 52 ; i++) { create and insert cards
                cards.insert( cards.end(),
                    card = new Card( static_cast<Card::Rank>(i % 13) ,
                        static_cast<Card::Suit>(i / 13 )));
                setCardCollection( *card );
            }
        }

    private:
        static Random r; // supports shuffling
    };

    Random Deck::r;
    // int main { ... }
```

b. Here is the answer in Java.

```
    // Card.java:

    package cards;

    public class Card implements Comparable<Card> {

        static String [] SuitStr = {"C","D","H","S" };
        static String [] RankStr = {
            "2","3","4","5","6","7","8","9","T","J","Q","K","A" };

        public enum Suit { CLUB, DIAMOND, HEART, SPADE };
        public enum Rank { TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
            NINE, TEN, JACK, QUEEN, KING, ACE };
        public Card(int r, int s) { suit = s; rank = r; }
        public Card(Rank r, Suit s) { suit = s.ordinal(); rank =
            r.ordinal(); }

        // Java requires an int be returned to meet contract of
        // Comparable interface
        public int compareTo(Card otherCard) {
            if (index() == otherCard.index()) return 0;
            return ( index() < otherCard.index() ) ? -1 : 1 ;
        }

        // support dummy console display
        public String Name() {
            return new String(RankStr[rank]) +
               new String(SuitStr[suit]);
        }

        // dummy display for console output...
        public void display(){ System.out.print( Name()+ " "); }

        // supports testing..., not required
        boolean IsIn(CardCollection coll) { return cc == coll; }

        // remove self from current collection; insert in
        // destination pile delegates work to card's current
        // collection;

        public void discard(Pile destination){
            cc.deleteCard(this);
            destination.insertCard(this);
        }

        private int index() { return suit * 13 + rank; }
        void setCollection(CardCollection coll) { cc = coll; }

        private int suit;
        private int rank;
        private CardCollection cc;
    };

    // CardCollection.java:

    package cards;
    import java.util.*;

    class CardCollection {

        enum Visibility { FRONT, BACK }; // NOT USED HERE...

        void initialize() { cards.clear(); }

        // by default, add at top
```

```
   // note package access for insert/delete;
   // transfers occur publicly through
   // Card.discard(destination) and
   // Pile.draw(destination hand)

   void insertCard(Card card ) {
      cards.add(0,card); setCardCollection(card);
   }

   void deleteCard(Card card) {
      cards.remove(card); clearCardCollection(card);
   }

   public boolean empty() { return cards.size() == 0; }
   public int size() { return cards.size(); }

   // simple output of collection onto console
   public void Show() {
      System.out.println();
      for(Card c : cards) c.display();
      System.out.println();
   }

   Visibility visibility;
   // Point location;

   ArrayList<Card> cards = new ArrayList<Card>();

   void setCardCollection(Card card)
      {card.setCollection( this ); }
   void clearCardCollection(Card card)
      {card.setCollection( null ); }
}

// Deck.java:

package cards;
import java.util.*;

public class Deck extends Pile {

   public Deck() { initialize(); }
      // constructor builds a deck...

   public Hand[] deal( int nhands, int hsize ) {
      if ( nhands * hsize > cards.size()) return null;
         // not enough!
      Hand hands[] = new Hand[nhands]; // create the hands
      for (int i = 0; i < nhands; i++) hands[i] = new Hand();
      for (int j = 0; j < hsize; j++ ) // deal "around the table"
      for (int i = 0; i < nhands; i++)
      draw(hands[i]); // draw from deck to appropriate hand
```

```
         return hands;
      }

      // swap cards in existing deck N times
      public void shuffle(int nTimes) {
         int ri;
         Card tmp;
         for (int i = 0; i < nTimes; i++ ) {
            for (int j = 0; j < 52; j++) {
               ri = r.nextInt(52);
               tmp = cards.get(j);
               cards.set(j, cards.get(ri));
               cards.set(ri, tmp );
            }
         }
      }

      public void shuffle() { shuffle(5); }

      void initialize() {
         super.initialize();
            // clear what's left; see note at base
         Card card;
         for (int i = 0; i < 52 ; i++) { //create and insert cards
            cards.add( card = new Card( i % 13, i / 13 ));
            setCardCollection( card );
         }
      }

      private static Random r = new Random(); // supports shuffling
   }

   // Pile.java:

   package cards;
   import java.util.*;

   class Pile extends CardCollection {

      // note: top of pile is head of list, not last in list...
      public Card topOfPile() {
         return cards.isEmpty() ? null : cards.get(0);
      }

      public Card bottomOfPile() {
         return cards.isEmpty() ? null :
            cards.get(cards.size() - 1);
      }

      // draw from Pile into Hand
      public void draw(Hand aHand) {
         Card card = topOfPile();
```

```
            if (card == null) return; // return false...
            deleteCard(card);
            aHand.insertCard(card); // return true...
         }

         // no need to override except display;
         // CardCollection inserts are at top

         void insertCard(Card card) {
            super.insertCard(card) ; /*display()*/; }

         // remove only top card...
         void deleteCard(Card card){
            if (card != topOfPile()) return;
            super.deleteCard(card) ; /*display()*/;
         }
      }

      // Hand.java:

      package cards;
      import java.util.*;

      public class Hand extends CardCollection {

         int initialSize;

         public void insertCard(Card card) {
            int ndx = 0;
            ListIterator<Card> it = cards.listIterator();
               // find insert point
            while (it.hasNext() && card.compareTo(it.next())> 0 )
               ndx++;
            cards.add(ndx, card);
            setCardCollection(card); /*display...*/;
         }

         // no need to override except to add display.
         void deleteCard(Card card){
            super.deleteCard(card) ; /*display()...*/;
         }

         // inserts are done in sort order,
         // so this won't make any changes...

         void sortCards() {
            TreeSet<Card> tempset = new TreeSet<Card>(cards);
            cards.clear();
            Iterator<Card>it = tempset.iterator();
            cards.addAll(tempset);
         }
      }
```

18.11 For a typical application, these kinds of queries would normally be handled using database code. So we do not provide a C++ or Java answer to this question.

18.12 Implement all associations involving *Box*, *Link*, *LineSegment* and *Point*.

Note that the composition relationships for *Box* and *Link* a in the exercise re incorrect. According to the definition of composition, a constituent part has a coincident lifetime with its assembly. This certainly is not the case for a diagram editor as a *Box* can be in a *Sheet* and then cut and placed in the *Buffer*. We should have used an aggregation relationship (a hollow diamond instead of a solid diamond). Our solution to the exercise assumes this correction.

- *Box* has * (0..n) *Links*. This is implemented as *set<Links*>* links within *Box*. The link destructor detaches the *Link* from its *Boxes* at destruction.
- *Box* has a mandatory aggregation to one *Collection*. This is implemented as a *Collection&* member of *Box*. Using a reference member ensures a *Box* cannot be created in the absence of its *Collection*. In addition, the *Box* constructor is private and *Box* creation is executed by the *Box* friend *Collection*, through the method *Collection::createBox*.
- *Link* must be associated with exactly two *Boxes*. *Link* contains *Box* boxes[2]*. The *Link* constructor takes two *Box&*'s as arguments, ensuring the *Link* is created between two existing boxes.
- *Link* has a mandatory aggregation to one *Collection*. This is implemented by privatizing the *Link* constructor and constraining *Link* creation to the method *Collection::createLink*. The *Collection&* attribute of *Link* is derived through the member function *Link::myCollection()*. Because *Links* cannot exist without their *Boxes*, and because a *Box* cannot exist without its *Collection*, a *Link* can readily determine its *Collection* by asking its *Box(es)*.
- *Link* has * (0..n) *LineSegments*. The points defining a link's line segments are embedded as a *list<Point>* member within the link itself.
- *LineSegment* has a mandatory composition to one *Link*. This is implemented by embedding the segment-defining points within the *Link* itself.
- *LineSegment* is associated with exactly two *Points*. *LineSegment* objects could be dynamically created as needed from two of a link's points. We have not implemented a physical *LineSegment* class, which would most likely be associated with the graphical elements of the editor and used as an interface to the link.
- *Point* can participate in one or two *LineSegments*. *Points* are implemented as “value objects”—they have no logical behavior or associations of their own. They serve purely as data values. We could enforce the model’s prescription that *Points* participate in at least, and no more than, one or two *LineSegments* by comparing their data values to existing *Points* in *Links*, or by promoting *Points* to “reference objects” with constrained construction and/or behavior. It is likely that any constraints on the loci of drawing objects would be imposed and translated not by the application’s internal

model, but by the display components of the application. The costs of promoting *Points* are not justified by their behaviorless use.

The following code answers Exercise 18.12 through 18.15.

```
#include<list>
#include<set>

#include<string>
#include<iostream>

//----------------------------------------------

// POINT IS A SIMPLE, UTILITARIAN VALUE-BASED OBJECT...
// IT WILL BE USED AS AN ENCAPSULATED ATTRIBUTE BY ITS CLIENTS

struct Point {
   int x, y;

   Point(int xCoord = 0, int yCoord = 0) :
      x(xCoord), y(yCoord) {}

   bool operator==(const Point& other) const {
      return ( x == other.x && y == other.y );
   }
};

//--------------------------------------------------

class Box{
   friend class Collection;
      // COLLECTION MANIPULATES ITS BOXES...
   friend class Link;

   string text; // not used ...
   int left, top, width, height;
   bool selected;

   set<Link*> links; // A BOX CAN PARTICIPATE IN 0..n LINKS
   Collection& collection;
      // A BOX MUST PARTICIPATE IN ONE COLLECTION

   Box(); // PROHIBIT DEFAULT, COPY CONSTRUCTION
   Box( const Box& other );
   Box& operator=( const Box& other ); // PROHIBIT ASSIGNMENT

   set<Link*>& myLinks() { return links; }
      // ACCESSOR FOR FRIENDS

   // UNLINKING NOT PUBLICLY AVAILABLE;
   // ONLY FRIENDS CAN DO THIS...
```

```
      // REMOVES A SINGLE LINK FROM links SET
      bool unLink(Link& lnk) {
         set<Link*>::iterator it = links.find(&lnk);
         if (it == links.end()) return false; // invalid link

         links.erase(it); // ELIMINATE THE LINK FROM THIS BOX
         return true;
      }

      // REMOVE ALL LINKS FROM links SET
      void unLinkAllLinks() {
         links.erase(links.begin(), links.end());
      }

      // FOR USE BY PUBLIC AllConnectedBoxes()
      void AllConnectedBoxes(set<const Box*>& tmp) const;

      // BOX MUST EXIST WITHIN CONTEXT OF COLLECTION...
      // TO ENFORCE MODEL, CTOR IS PRIVATE AND COLLECTION IS FRIEND
      Box(Point& upperLeft, int wid, int ht, Collection& coll ) :
         top(upperLeft.y), left(upperLeft.x), width(wid),
         height(ht), selected(false),collection(coll){
      }

   public:

      // EXERCISE 18.13
      inline void cut(); // DELEGATES TO COLLECTION
         // BEWARE DANGLING PTRS/REFS TO CUT BOXES!

      ~Box();

      const Collection& myCollection() { return collection; }

      void select() { selected = true; }
      void deselect(){ selected = false; }

      bool toggleSelect() {
         return selected = selected ? false : true;
      }
      bool isSelected() { return selected; }

      // EXERCISE 18.15 a & b
      set<const Box*> DirectlyConnectedBoxes() const;
      set <const Box*> AllConnectedBoxes() const;

      // FOR DEBUG AND DEMO
      set<const Link*> Links();
   };

   //---------------------------------------------
```

```
class Link{
   friend class Collection;
      // COLLECTION MANIPULATES ITS LINKS...

   Box* boxes[2]; // LINK LINKS EXACTLY 2 BOXES

   list<Point> segmentPoints;
      // LINE SEGMENTS ARE DEFINED BY
      // A SEQUENCE OF POINTS
   bool selected;

   Link();
   Link( const Link& other );
   Link& operator=( const Link& other );

   // PRIVATE CONSTRUCTOR: PREVENT CONSTRUCTION IF PARAMS
   // ARE INVALID
   // LINK CTOR IS CALLED BY Collection::createLink (EX. 18.14)
   Link( Box& box0, Box& box1, list<Point>& points );

   // DETACH LINK FROM ITS BOXES...
   inline void detach();

public:

   ~Link() { detach(); } // UNATTACH BOXES

   void select() { selected = true; }
   void deselect(){ selected = false; }

   bool toggleSelect()
      { return selected = selected ? false : true; }
   bool isSelected() { return selected; }

   // 18. 11
   // DERIVED ATTRIBUTE: A LINK MUST HAVE BOXES TO LINK, AND
   // THOSE BOXES BELONG TO THE SAME COLLECTION AS THE LINK...
   const Collection& myCollection() const
      { return boxes[0]->myCollection(); }

   // EXERCISE 18.15.c
   // IS THIS LINK PART OF A DIRECT LINK INVOLVING ARGUMENT box?
   bool linksTo(const Box& box) const {
      return( boxes[0] == &box || boxes[1] == &box );
   }

   // EXERCISE 18.15.d
   const Box* OtherBox(const Box& box ) const {
      if (! linksTo(box)) return 0;
         // this DOES NOT LINK TO box IN THIS LINK
      return boxes[0] == &box ? boxes[1] : boxes[0] ;
   }
```

```
    // EXERCISE 18.15.g
    list<Point> SegmentPoints(const Box& start, const Box& end);
};

//---------------------------------------------

class Collection {
    set<Box*> boxes;
    set<Link*> links;

public:

    Box* createBox( Point& upperLeft, int wid, int ht );

    // EXERCISE 18.13
    bool cutBox( Box& box );

    // EXERCISE 18.14: METHOD TO LINK TWO BOXES.
    Link* createLink(Box& b1, Box& b2, list<Point>& pointlist);

    bool cutLink( Link* link );

    set<const Link*> Links() const;
    set<const Box*> Boxes() const;

    // EXERCISE 18.15.e
    list<const Link*> SharedLinks(const Box& b1, const Box& b2);

    // EXERCISE 18.15.f
    list<const Link*> LinksSelectedAndUnselected();
};
```

Implementations:

```
// BOX.CPP ----------------------------------------------

Box::~Box()
{
    set<Link*>::iterator it = links.begin();
    while (it != links.end()) collection.cutLink(*it);
}

// EXERCISE 18.13
void Box::cut()
{
    collection.cutBox(*this); // deleteS BOX; BEWARE DANGLERS!
}

// EXERCISE 18.15.a
// USE SET TO AVOID DUPLICATES WHEN THERE ARE DUPE LINKS!
set<const Box*> Box::DirectlyConnectedBoxes() const {
    set<const Box*> connects;
    set<Link*>::const_iterator it = links.begin();

    while( it != links.end())
       connects.insert( ((*it++)->OtherBox(*this)));

    return(connects);
}

// EXERCISE 18.15.b
set <const Box*> Box::AllConnectedBoxes() const {
    set<const Box*> bxs;
    bxs.insert(this);
    AllConnectedBoxes(bxs);
    bxs.erase(this);
    return bxs;
}

void Box::AllConnectedBoxes(set<const Box*>& bxs) const {
    set<const Box*> directconnects = DirectlyConnectedBoxes();
    set<const Box*>::iterator boxIt = directconnects.begin();

    while( boxIt != directconnects.end()) {
       // FOR EACH DIRECT CONNECT

       if ( bxs.find(*boxIt) != bxs.end() ) return;
          // BEEN THERE

       bxs.insert( *boxIt);
       (*boxIt++)->AllConnectedBoxes(bxs); // GET ITS CONNECTS
    }
}

set<const Link*> Box::Links()
{
    set<const Link*> myLinks;
    set<Link*>::iterator it = links.begin();
    while (it != links.end()) myLinks.insert(*it++);
    return myLinks;
}

// LINK.CPP ---------------------------------------------

// DERIVED ATTRIBUTE...
//const Collection& Link::myCollection() const
// { return boxes[0]->myCollection(); }

// LIST OF POINTS SHOULD BE VALIDATED BEFORE CONSTRUCTION IS
// ALLOWED. LIST GOES FROM BOX0 TO BOX1
Link::Link( Box& box0, Box& box1, list<Point>& points ) :
    selected(false)
{
    boxes[0] = &box0; boxes[1] = &box1;
    segmentPoints = points; // COPY PARAM LIST!
}

// LINK ASKS BOXES TO DETACH THEMSELVES, THEN NULLS
// OWN RECORDS OF BOX ATTACHMENTS
void Link::detach()
{
    if (boxes[0]) { boxes[0]->unLink(*this); boxes[0] = 0; }
    if (boxes[1]) { boxes[1]->unLink(*this); boxes[1] = 0; }
}

// EXERCISE 18.15.g
list<Point> Link::SegmentPoints(const Box& start,
    const Box& end)
{
    if (( &start == boxes[0] ) && (&end == boxes[1] ))
       return segmentPoints;

    list<Point> lst;

    if (( &start == boxes[1] ) && (&end == boxes[0] )) {
       segmentPoints.reverse();
       lst = segmentPoints; segmentPoints.reverse();
    }

    return lst;
       // LST WILL BE EMPTY IF A BOX IS INVALID FOR THIS LINK
}

// COLLECTION.CPP ----------------------------------------
// CONTROLLED CONSTRUCTION: BOX & LINK MUST BE "PART OF" A
// COLLECTION (ALTHOUGH A COLLECTION NEED NOT HAVE ANY OF
// EITHER). TO ENSURE THE ASSOCIATION, AND TO ALLOW SAFE
// DELETION OF BOXES AND LINKS, WE'LL DELEGATE CREATION OF BOXES
// AND LINKS TO THEIR COLLECTION, AND GUARANTEE ALL ARE
// DYNAMICALLY ALLOCATED AND THEREFORE SAFE TO DELETE...

Box* Collection::createBox( Point& upperLeft, int wid, int ht )
{
    Box* newBox = new Box(upperLeft, wid, ht, *this);
    boxes.insert(newBox);
    return newBox;
}

// EXERCISE 18.14
Link* Collection::createLink(Box& b1, Box& b2,
    list<Point>& pointlist)
{
    // NOT ENOUGH POINTS... AT LEAST TWO REQUIRED FOR DIRECT LINK
    if (pointlist.size() < 2 ) return 0;
    // VALIDATE POINTS -- ARE THEY A PROPER POINT SEQUENCE?
    // DO START AND END POINTS FALL IN BOUNDS FOR SPECIFIED
    // BOXES?? DO POINTS CONNECT TO LINE?

    // IF INVALID, return 0;

    // PUT POINTS IN ORDER IF NECESSARY...

    Link* newLink = new Link(b1,b2,pointlist);

    links.insert(newLink); // ADD TO COLLECTION'S LINK REGISTRY

    b1.links.insert(newLink); // ADD TO BOXS' LINK REGISTRIES
    b2.links.insert(newLink);

    return newLink;
}

// EXERCISE 18.13

// THE CUT OPERATIONS WILL BE THE RESPONSIBILITY OF
// THE CONTROLLING COLLECTION...

bool Collection::cutBox( Box& box )
{
    set<Box*>::iterator boxIt = boxes.find(&box);
    if (boxIt == boxes.end()) return false;
       // THIS BOX NOT IN THIS COLLECTION

    // FOR EACH LINK IN BOX, DESTROY THE LINK

    if (box.myLinks().size() > 0 ) {
       set<Link*>::iterator linkIt =
          (*boxIt)->myLinks().begin();
       while (linkIt != (*boxIt)->myLinks().end())
          cutLink(*linkIt++);
}

boxes.erase( boxIt ); // REMOVE FROM boxes REGISTRY
    delete &box; // DESTROY (cut) the box
    return true;
}

bool Collection::cutLink( Link* link )
{
    if (!link) return false;

    set<Link*>::iterator it = links.find(link);
    if (it == links.end()) return false;

    links.erase(it); // REMOVE FROM COLLECTION'S links REGISTRY
    delete link; // DESTROY
```

```
        return true;
    }

    // EXERCISE 18.15.e DETERMINE ALL LINKS BETWEEN TWO BOXES
    list<const Link*> Collection::SharedLinks(const Box& b1,
        const Box& b2)
    {
        list<const Link*> sharedlinks;
        set<Link*>::iterator it = links.begin();

        while (it != links.end()) {
            if ( (*it)->linksTo(b1) && (*it)->linksTo(b2) )
                sharedlinks.insert(sharedlinks.end(),*it);
            it++;
        }

        return sharedlinks;
    }

    // EXERCISE 18.15.f
    list<const Link*> Collection::LinksSelectedAndUnselected()
    {
        list<const Link*> linksEm;
        set<Link*>::iterator it = links.begin();

        while (it != links.end()) {
            if ( ((*it)->boxes[0]->isSelected() &&
                !(*it)->boxes[1]->isSelected()) ||
                ((*it)->boxes[1]->isSelected() &&
                !(*it)->boxes[0]->isSelected()) )

                linksEm.insert(linksEm.end(), *it);

            it++;
        }

        return linksEm;
    }

    // FOR DEMO/DEBUG
    set<const Box*> Collection::Boxes() const
    {
        set<const Box*> bxs;
        set<Box*>::const_iterator it = boxes.begin();
        while (it != boxes.end()) bxs.insert(*it++);
        return bxs;
    }

    set<const Link*> Collection::Links() const {
        set<const Link*> lnks;
        set<Link*>::const_iterator it = links.begin();
        while (it != links.end()) lnks.insert(*it++);
        return lnks;
    }

    //-----------------------------------------------
```

18.13 Implement the *cut* operation on the class *Box*. See answer to Exercise 18.12.

- *Box::cut()* delegates its work to *Collection::cutBox(Box& box)*. The *Box* reports its set of *Links* to the *Collection*. For each *Link* in *Box's set<Link*>*, the *Collection* applies its *cutLink(Link&)* method. In *cutLink(Link&)*, the *Collection* asks the *Link* to detach itself from both of its *Boxes*, then removes the *Link** from its set of *Links*, and deletes the *Link* (recovers memory) itself. The *Link's Points* are not dynamically allocated, and so are automatically deallocated as part of *Link* destruction. The *Collection* then removes the *Box* from its set of *Boxes*, and deletes the (now *Link*-less) *Box*.

18.14 Write a method to create a link between two boxes. Inputs are two boxes and a list of points. Also write a method to destroy a link. See answer to Exercise 18.12.

- *Link* Collection::createLink(Box& b1, Box& b2, list<Point>& pointlist)* performs this service. The *Link* constructor itself is privatized, and *Links* are created only after the *Collection* validates the inputs. The new *Link* creates its own *LineSegments* based on the *Point* list. The *Collection* notifies the *Boxes* to record the *Link* in their *Link* registries, and records the *Link* in its own registry of *Links*.
- *Link* destruction is provided by *bool Collection::cutLink(Link* link)*. The *Collection* removes the *Link* from its *Link* registry, and deletes the *Link*. The *Link* destructor calls *Link::detach()*, which asks the attached boxes to remove the *Link* from their registries.

18.15 See answer to Exercise 18.12.

a. Other boxes directly linked to a box:
```
list<const Box*> Box::DirectlyConnectedBoxes();
```
b. Other boxes indirectly linked to a box:
```
set<const Box*> Box::AllConnectedBoxes() const;
```
c. Is a given box associated with a given link?
```
bool Link::linksTo(const Box& box) const;
```
d. Find the box at the other end of a link:
```
Box* Link::OtherBox(const Box& box ) const;
```
e. Find all links between two boxes:
```
list<const Link*> Collection:
    SharedLinks(const Box& b1, const Box& b2);
```

f. Determine which links connect a selected box and a deselected box: (Note: we implement this query in *Collection*, the superclass for both *Sheet* and *Selection*. The subclasses are not implemented here.)

```
list<const Link*> Collection::LinksSelectedAndUnselected()
```

g. Produce an ordered set of points for two boxes and their link.

```
list<Point> Link::SegmentPoints(const Box& start,
   const Box& end);
```

19

Databases

19.1 All in all, we consider Figure E19.1c and Figure E19.1d most desirable. The first two figures are poor models. Here are some observations.

- Figure E19.1a fails to indicate that each *from-to* node pair can have many edges.
- Figure E19.1a may make it difficult to answer queries that specify an edge name. Figure E19.1b may make it difficult to answer queries that specify a node name.
- The symmetry in Figure E19.1b can be confusing; we had to arbitrarily break the symmetry with the *end1* and *end2* designation. Some implementations of Figure E19.1b may require that instances be entered twice or that an edge be searched through both qualifiers in order to find all pertinent instances.
- Figure E19.1b cannot represent the case where only one edge connects to a node. (Useful for an incomplete diagram.)
- Figure E19.1c and Figure E19.1d are better than the first two figures because they give node and edge equal stature. Nodes and edges seem equally important in the construction of directed graphs so they both should be recognized as classes.
- Figure E19.1c and Figure E19.1d can be extended to permit dangling edges and/or nodes by changing the “1” multiplicity to “0..1” multiplicity. This distinction can be important for software that must support partially completed diagrams.
- Figure E19.1c and Figure E19.1d capture more multiplicity constraints than the first two figures. Each edge has exactly one *from* node and one *to* node.
- An implementation of Figure E19.1d must assign the *end* qualifier an enumeration data type with values: *from* and *to*. Enforcing the enumeration may be awkward for some databases and languages.
- Figure E19.1d can most easily find all edges connected to a given node.

19.2

- Tables for Figure E19.1a. The underlying class model is a poor model.

Node table

nodeID	nodeName (ck1)

ToFrom table

toNodeID (references Node)	**toNodeID** (references Node)	edgeName

Figure A19.1 Tables for Figure E19.1a

- Tables for Figure E19.1b. The underlying class model is a poor model.

Edge table

edgeID	edgeName (ck1)

End1End2 table

edge1ID (references Edge)	**end1**	**edge2ID** (references Edge)	**end2**	nodeName

Figure A19.2 Tables for Figure E19.1b

- Tables for Figure E19.1c.

Edge table

edgeID	edgeName (ck1)	fromNodeID (references Node)	toNodeID (references Node)

Node table

nodeID	nodeName (ck1)

Figure A19.3 Tables for Figure E19.1c

- Tables for Figure E19.1d. *EdgeID* + *nodeID* is not a candidate key for the association table because an edge may connect a node with itself.

Edge table

edgeID	edgeName (ck1)

Node table

nodeID	nodeName (ck1)

Edge_Node table

edgeID (references Edge)	**end**	nodeID (references Node)

Figure A19.4 Tables for Figure E19.1d

19.3a. SQL commands to create database tables and indexes for Figure E19.1c.

```
CREATE TABLE Node
( node_ID   NUMBER(30)   CONSTRAINT nn_node1 NOT NULL,
  node_name VARCHAR2(50) CONSTRAINT nn_node2 NOT NULL,
CONSTRAINT pk_node PRIMARY KEY (node_ID),
CONSTRAINT uq_node1 UNIQUE (node_name));
CREATE SEQUENCE seq_node;

CREATE TABLE Edge
( edge_ID      NUMBER(30)   CONSTRAINT nn_edge1 NOT NULL,
  edge_name    VARCHAR2(50) CONSTRAINT nn_edge2 NOT NULL,
  from_node_ID NUMBER(30)   CONSTRAINT nn_edge3 NOT NULL,
  to_node_ID   NUMBER(30)   CONSTRAINT nn_edge4 NOT NULL,
  CONSTRAINT pk_edge PRIMARY KEY (edge_ID),
  CONSTRAINT uq_edge1 UNIQUE (edge_name));
CREATE SEQUENCE seq_edge;
CREATE INDEX index_edge1 ON Edge (from_node_ID);
CREATE INDEX index_edge2 ON Edge (to_node_ID);
ALTER TABLE Edge ADD CONSTRAINT fk_edge1
  FOREIGN KEY from_node_ID
  REFERENCES Node;
ALTER TABLE Edge ADD CONSTRAINT fk_edge2
  FOREIGN KEY to_node_ID
  REFERENCES Node;
```

b. SQL commands to create database tables and indexes for Figure E19.1d. We cascade deletes for *Edge* to *Edge_Node* because an *Edge* involves only two occurrences of *Edge_Node* so the effect would likely be anticipated. In contrast a *Node* could have a large number of *Edge_Node* occurrences, so we block propagation of deletes to avoid accidental side effects.

```
CREATE TABLE Node
( node_ID   NUMBER(30)   CONSTRAINT nn_node1 NOT NULL,
  node_name VARCHAR2(50) CONSTRAINT nn_node2 NOT NULL,
CONSTRAINT pk_node PRIMARY KEY (node_ID),
CONSTRAINT uq_node1 UNIQUE (node_name));
CREATE SEQUENCE seq_node;

CREATE TABLE Edge
( edge_ID   NUMBER(30)   CONSTRAINT nn_edge1 NOT NULL,
  edge_name VARCHAR2(50) CONSTRAINT nn_edge2 NOT NULL,
CONSTRAINT pk_edge PRIMARY KEY (edge_ID),
CONSTRAINT uq_edge1 UNIQUE (edge_name));
CREATE SEQUENCE seq_edge;

CREATE TABLE Edge_Node
( edge_ID NUMBER(30)  CONSTRAINT nn_edgenode1 NOT NULL,
  end     VARCHAR2(4) CONSTRAINT nn_edgenode2 NOT NULL,
  node_ID NUMBER(30)  CONSTRAINT nn_edgenode3 NOT NULL,
  CONSTRAINT pk_edgenode PRIMARY KEY (edge_ID,end));
CREATE INDEX index_edgenode1 ON Edge_Node (node_ID);
ALTER TABLE Edge_Node ADD CONSTRAINT fk_edgenode1
  FOREIGN KEY edge_ID
  REFERENCES Edge ON DELETE CASCADE;
ALTER TABLE Edge_Node ADD CONSTRAINT fk_edgenode2
  FOREIGN KEY node_ID
  REFERENCES Node;
ALTER TABLE Edge_Node ADD CONSTRAINT enum_edge_node1
  CHECK (end IN ('to', 'from'));
```

19.4 Populated tables for the directed graph in Figure E19.2 using the model of Figure E19.1c.

Node table

node_ID	node_name
1	n1
2	n2
3	n3
4	n4
5	n5

Edge table

edge_ID	edge_name	from_node_ID	to_node_ID
1	e1	n5	n4
2	e2	n3	n4
3	e3	n2	n3
4	e4	n2	n1
5	e5	n1	n5
6	e6	n5	n2

Figure A19.5 Populated database tables for Figure E19.1c

Populated tables for the directed graph in Figure E19.2 using the model of Figure E19.1d.

Edge table

edge_ID	edge_name
1	e1
2	e2
3	e3
4	e4
5	e5
6	e6

Node table

node_ID	node_name
1	n1
2	n2
3	n3
4	n4
5	n5

Edge_Node table

edge_ID	**end**	node_ID
e1	from	n5
e1	to	n4
e2	from	n3
e2	to	n4
e3	from	n2
e3	to	n3
e4	from	n2
e4	to	n1
e5	from	n1
e5	to	n5
e6	from	n5
e6	to	n2

Figure A19.6 Populated database tables for Figure E19.1d

19.5a. Given the name of an edge, determine the two nodes that it connects.

```
SELECT E.edge_name, EN.end, N.node_name
FROM Edge_Node EN, Node N, Edge E
WHERE E.edge_ID = EN.edge_ID AND
      N.node_ID = EN.node_ID AND
      E.edge_name = :anEdgeName;
```

b. Given the name of a node, determine all edges connected to or from it.

```
SELECT E.edge_name, EN.end, N.node_name
FROM Edge_Node EN, Node N, Edge E
WHERE E.edge_ID = EN.edge_ID AND
      N.node_ID = EN.node_ID AND
      N.node_name = :aNodeName;
```

c. Given a pair of nodes, determine the edges that directly connect them.

```
SELECT E.edge_name
FROM Edge_Node EN1, Edge_Node EN2, Edge E
WHERE ((EN1.end = ’from’ AND EN2.end = ’to’) OR
      (EN1.end = ’to’ AND EN2.end = ’from’)) AND
```

```
          EN1.edge_ID = EN2.edge_ID AND
          EN1.edge_ID = E.edge_ID AND
          EN1.node_ID = :aNode1  AND
          EN2.node_ID = :aNode2;
```

d. Given a node, determine the nodes connected through transitive closure. Pseudocode is required because SQL provides no intrinsic support for transitive closure.

```
NodeTransitiveClosure (startNode) RETURNS SET OF nodeID;
   visitedNodes := {};
   FindNextNodes (startNode, visitedNodes);
   RETURN (visitedNodes);
END OF NodeTransitiveClosure


FindNextNodes (startNode, visitedNodes);
   tempSet := FindDbNodes (startNode);

   /* do not revisit a node */
   FOR EACH aNode IN tempSet DO
       IF aNode IS IN visitedNodes THEN
          tempSet -= aNode;
       ENDIF
   END FOR EACH


   /* add remaining nodes to visited set */
   FOR EACH aNode IN tempSet DO
       visitedNodes += aNode;
   END FOR EACH


   /* recurse for newly discovered nodes */
   FOR EACH aNode IN tempSet DO
       FindNextNodes (aNode, visitedNodes);
   END FOR EACH
END OF FindNextNodes


FindDbNodes (startNode) RETURNS SET OF node_ID;
   /* The following code shows a SQL query returning a  */
   /* set. Most SQL programming language interfaces do  */
   /* not permit return of a set and would require      */
   /* looping through a cursor to accumulate the answer. */
   SELECT EN2.NodeID INTO nodeSet
   FROM EdgeNode EN1, EdgeNode EN2
   WHERE EN1.nodeID = :startNode AND
        EN1.edgeID = EN2.edgeID AND
        EN1.end = 'from' AND EN2.end = 'to';
   RETURN (nodeSet);
END OF FindDbNodes
```

19.6 Figure A19.7 shows the tables.

Term table

termID	term Type

Expression table

expressionID (references Term)	binary Operator	firstOperand (references Term)	secondOperand (references Term)

Variable table

variableID (references Term)	name (ck1)

Constant table

constantID (references Term)	value

Figure A19.7 Tables for Figure E19.3

19.7 SQL commands for Figure E19.3. We arbitrarily store values for constants as strings. The strings would be converted to numbers before evaluating expressions.

```
CREATE TABLE Term
( term_ID     NUMBER(30)   CONSTRAINT nn_term1 NOT NULL,
  term_type   VARCHAR2(10) CONSTRAINT nn_term2 NOT NULL,
CONSTRAINT pk_term PRIMARY KEY (term_ID));

CREATE SEQUENCE seq_term;

ALTER TABLE Term ADD CONSTRAINT enum_term1
CHECK (term_type IN ('Expression','Variable','Constant'));

CREATE TABLE Expression
( expression_ID   NUMBER(30)  CONSTRAINT nn_exp1 NOT NULL,
  binary_operator VARCHAR2(10) CONSTRAINT nn_exp2 NOT NULL,
  first_operand   NUMBER(30)  CONSTRAINT nn_exp3 NOT NULL,
  second_operand  NUMBER(30)  CONSTRAINT nn_exp4 NOT NULL,
CONSTRAINT pk_exp PRIMARY KEY (expression_ID));

CREATE INDEX index_exp1 ON Expression (first_operand);

CREATE INDEX index_exp2 ON Expression (second_operand);

ALTER TABLE Expression ADD CONSTRAINT fk_exp1
  FOREIGN KEY expression_ID
  REFERENCES Term ON DELETE CASCADE;

ALTER TABLE Expression ADD CONSTRAINT fk_exp2
  FOREIGN KEY first_operand
  REFERENCES Term;
```

```
ALTER TABLE Expression ADD CONSTRAINT fk_exp3
  FOREIGN KEY second_operand
  REFERENCES Term;

CREATE TABLE Variable
( variable_ID   NUMBER(30)   CONSTRAINT nn_var1 NOT NULL,
  variable_name VARCHAR2(50) CONSTRAINT nn_var2 NOT NULL,
CONSTRAINT pk_var PRIMARY KEY (variable_ID),
CONSTRAINT uq_var1 UNIQUE (variable_name));
ALTER TABLE Variable ADD CONSTRAINT fk_var1
  FOREIGN KEY variable_ID
  REFERENCES Term ON DELETE CASCADE;

CREATE TABLE Constant
( constant_ID  NUMBER(30)   CONSTRAINT nn_const1 NOT NULL,
  const_value  VARCHAR2(50) CONSTRAINT nn_const2 NOT NULL,
CONSTRAINT pk_const PRIMARY KEY (constant_ID));
ALTER TABLE Constant ADD CONSTRAINT fk_const1
  FOREIGN KEY constant_ID
  REFERENCES Term ON DELETE CASCADE;
```

19.8 Populated tables for Figure E19.3

Term table

term_ID	termType
1	expression
2	expression
3	expression
4	variable
5	expression
6	expression
7	variable
8	constant
9	constant

Variable table

variable_ID	variable_name
4	X
7	Y

Constant table

constant_ID	const_value
8	2
9	3

Expression table

expression_ID	binary_operator	first_operand	second_operand
1	/	2	3
2	+	4	5
3	-	6	7
5	/	7	8
6	/	4	9

Figure A19.8 Populated database tables for Figure E19.3

19.9 We include tables for *Polyline* and *ObjectGroup* even though they have no attributes. As a matter of style we prefer to apply standard mapping rules unless there is a performance bottleneck; it is easier to define foreign keys if you do not elide tables. A real problem would contain more attributes than shown in the exercise. In general, most of these additional attributes would further describe classes and could be null.

Note that we show separate tables for *Ellipse* and *Rectangle* even though they have the same attributes, because they really are two different things. A full model would most likely have additional attributes that would distinguish *Ellipse* and *Rectangle*.

We show *document_ID* + *page_number* as a candidate key for the page table. This constraint is based on our understanding of desktop publishing and was not specified in the class model.

Document table

documentID	pageWidth	pageHeight	leftMargin	rightMargin

Page table

pageID	pageNumber (ck1)	documentID (ck1) (references Document)

DrawingObject table

drawingObjectID	lineThickness	drawingObjectType	pageID (references Page)	objectGroupID (references ObjectGroup)

Ellipse table

ellipseID (references DrawingObject)	boundingBoxID (ck1) (references BoundingBox)

Rectangle table

rectangleID (references DrawingObject)	boundingBoxID (ck1) (references BoundingBox)

Polyline table

polylineID (references DrawingObject)

Point table

pointID	x	y	polylineID (references Polyline)

Textline table

textlineID (references DrawingObject)	alignment	text	pointID (ck1) (references Point)	fontID (references Font)

Figure A19.9 Tables for Figure E19.4

ObjectGroup table

objectGroupID (references DrawingObject)

BoundingBox table

bounding BoxID	left Edge	top Edge	width	height

Font table

fontID	fontSize	fontFamily	isBold	isItalic	isUnderlined

Figure A19.9 (continued) Tables for Figure E19.4

19.10 The table is the same as that for Exercise 19.9 except for the following change.

Point table

pointID	x	y	polylineID (ck1) (references Polyline)	sequenceNumber (ck1)

Figure A19.10 Change to table from Exercise 19.9

Note that a relational DBMS stores the rows of a table in an arbitrary order, and not necessarily in the order specified by *sequenceNumber*. To retrieve points in the proper order an *ORDER BY* clause must be included in the SQL query, such as:

```
SELECT point_ID
FROM Point
WHERE polyline_ID = :aPolyline
ORDER BY sequence_Number;
```

19.11 The tables are the same as that for Exercise 19.9 except for the following changes and cutting the *pointID* field from the *Textline* table. It is arbitrary whether we bury the foreign key for the one-to-one association in *Textline* or *TextlinePoint*.

Point table

pointID	x	y	point Type

PolylinePoint table

polylinePointID (references Point)	polylineID (references Polyline)

TextlinePoint table

textlinePointID (references Point)	textlineID (references Textline)

Figure A19.11 Changes to tables from Exercise 19.9

The proposed revision adds a structural constraint. Instead of just stating that a point may or may not associate with a textline or polyline, we can be more specific. We can state that each point associates with exactly one polyline or textline. However the change clutters the model. The merits of the revision depends on the importance of the constraint.

Unfortunately current RDBMS do not support generalization. Thus it is difficult to enforce the exclusive ‘or’ nature of a generalization in RDBMS tables. There are basically two alternatives.

- Forget about trying to enforce the generalization . (Disadvantage: loses the constraint.)
- Enforce the generalization with application code. (Disadvantage: error prone—may be omitted due to oversight. Also it is time consuming.)

19.12 We assume a real coordinate system in assigning data types. Note that the SQL code does not enforce the exclusive-or aspect of generalization.

```
CREATE TABLE Document
( document_ID   NUMBER(30)     CONSTRAINT nn_doc1 NOT NULL,
  page_width    NUMBER(20,10) CONSTRAINT nn_doc2 NOT NULL,
  page_height   NUMBER(20,10) CONSTRAINT nn_doc3 NOT NULL,
  left_margin   NUMBER(20,10) CONSTRAINT nn_doc4 NOT NULL,
  right_margin  NUMBER(20,10) CONSTRAINT nn_doc5 NOT NULL,
CONSTRAINT pk_doc PRIMARY KEY (document_ID));

CREATE SEQUENCE seq_doc;

CREATE TABLE Page
( page_ID      NUMBER(30)    CONSTRAINT nn_page1 NOT NULL,
  page_number  NUMBER(10)    CONSTRAINT nn_page2 NOT NULL,
  document_ID  NUMBER(30)    CONSTRAINT nn_page3 NOT NULL,
CONSTRAINT pk_page PRIMARY KEY (page_ID),
CONSTRAINT uq_page1 UNIQUE (document_ID, page_number));

CREATE SEQUENCE seq_page;

ALTER TABLE Page ADD CONSTRAINT fk_page1
  FOREIGN KEY document_ID
  REFERENCES Document;

CREATE TABLE Drawing_Object
( drawing_object_ID  NUMBER(30) CONSTRAINT nn_do1 NOT NULL,
  line_thickness     NUMBER(10) CONSTRAINT nn_do2 NOT NULL,
  drawing_object_type VARCHAR2(20)
    CONSTRAINT nn_do3 NOT NULL,
  page_ID           NUMBER(30) CONSTRAINT nn_do4 NOT NULL,
  object_group_ID    NUMBER(30),
CONSTRAINT pk_do PRIMARY KEY (drawing_object_ID));
```

```
CREATE SEQUENCE seq_do;

CREATE INDEX index_do1 ON Drawing_Object (page_ID);

CREATE INDEX index_do2 ON Drawing_Object (object_group_ID);

ALTER TABLE Drawing_Object ADD CONSTRAINT fk_do1
  FOREIGN KEY page_ID
  REFERENCES Page;

ALTER TABLE Drawing_Object ADD CONSTRAINT fk_do2
  FOREIGN KEY object_group_ID
  REFERENCES Object_Group;

ALTER TABLE Drawing_Object ADD CONSTRAINT enum_do1
  CHECK (drawing_object_type IN (’Ellipse’, ’Rectangle’,
  ’Polyline’, ’Textline’, ’Object_Group’));


CREATE TABLE Ellipse
( ellipse_ID      NUMBER(30) CONSTRAINT nn_ell1 NOT NULL,
  bounding_box_ID NUMBER(30) CONSTRAINT nn_ell2 NOT NULL,
CONSTRAINT pk_ell PRIMARY KEY (ellipse_ID),
CONSTRAINT uq_ell1 UNIQUE (bounding_box_ID));

CREATE INDEX index_ell1 ON Ellipse (bounding_box_ID);

ALTER TABLE Ellipse ADD CONSTRAINT fk_ell1
  FOREIGN KEY ellipse_ID
  REFERENCES Drawing_Object ON DELETE CASCADE;

ALTER TABLE Ellipse ADD CONSTRAINT fk_ell2
  FOREIGN KEY bounding_box_ID
  REFERENCES Bounding_Box;


CREATE TABLE Rectangle
( rectangle_ID    NUMBER(30) CONSTRAINT nn_rect1 NOT NULL,
  bounding_box_ID NUMBER(30) CONSTRAINT nn_rect2 NOT NULL,
CONSTRAINT pk_rect PRIMARY KEY (rectangle_ID),
CONSTRAINT uq_rect1 UNIQUE (bounding_box_ID));

CREATE INDEX index_rect1 ON Rectangle (bounding_box_ID);

ALTER TABLE Rectangle ADD CONSTRAINT fk_rect1
  FOREIGN KEY rectangle_ID
  REFERENCES Drawing_Object ON DELETE CASCADE;

ALTER TABLE Rectangle ADD CONSTRAINT fk_rect2
  FOREIGN KEY bounding_box_ID
  REFERENCES Bounding_Box;


CREATE TABLE Polyline
( polyline_ID NUMBER(30) CONSTRAINT nn_polyline1 NOT NULL,
 CONSTRAINT pk_polyline PRIMARY KEY (polyline_ID));
```

```
ALTER TABLE Polyline ADD CONSTRAINT fk_polyline1
  FOREIGN KEY polyline_ID
  REFERENCES Drawing_Object ON DELETE CASCADE;

CREATE TABLE Point
( point_ID    NUMBER(30)    CONSTRAINT nn_point1 NOT NULL,
  x           NUMBER(20,10) CONSTRAINT nn_point2 NOT NULL,
  y           NUMBER(20,10) CONSTRAINT nn_point3 NOT NULL,
  polyline_ID NUMBER(30),
CONSTRAINT pk_point PRIMARY KEY (point_ID));

CREATE SEQUENCE seq_point;

CREATE INDEX index_point1 ON Point (polyline_ID);

ALTER TABLE Point ADD CONSTRAINT fk_point1
  FOREIGN KEY polyline_ID
  REFERENCES Polyline;

CREATE TABLE Textline
( textline_ID NUMBER(30)    CONSTRAINT nn_tline1 NOT NULL,
  alignment   VARCHAR2(10),
  text        VARCHAR2(255) CONSTRAINT nn_tline2 NOT NULL,
  point_ID    NUMBER(30)    CONSTRAINT nn_tline3 NOT NULL,
  font_ID     NUMBER(30)    CONSTRAINT nn_tline4 NOT NULL,
CONSTRAINT pk_tline PRIMARY KEY (textline_ID));

CREATE INDEX index_tline1 ON Textline (point_ID);

CREATE INDEX index_tline2 ON Textline (font_ID);

ALTER TABLE Textline ADD CONSTRAINT fk_tline1
  FOREIGN KEY textline_ID
  REFERENCES Drawing_Object ON DELETE CASCADE;

ALTER TABLE Textline ADD CONSTRAINT fk_tline2
  FOREIGN KEY point_ID
  REFERENCES Point;

ALTER TABLE Textline ADD CONSTRAINT fk_tline3
  FOREIGN KEY font_ID
  REFERENCES Font;

CREATE TABLE Object_Group
( object_group_ID NUMBER(30) CONSTRAINT nn_og1 NOT NULL,
 CONSTRAINT pk_og PRIMARY KEY (object_group_ID));

ALTER TABLE Object_Group ADD CONSTRAINT fk_og1
  FOREIGN KEY object_group_ID
  REFERENCES Drawing_Object ON DELETE CASCADE;
```

```
CREATE TABLE Bounding_Box
( bounding_box_ID NUMBER(30)  CONSTRAINT nn_bbox1 NOT NULL,
  left_edge      NUMBER(20,10) CONSTRAINT nn_bbox2 NOT NULL,
  top_edge       NUMBER(20,10) CONSTRAINT nn_bbox3 NOT NULL,
  width          NUMBER(20,10) CONSTRAINT nn_bbox4 NOT NULL,
  height         NUMBER(20,10) CONSTRAINT nn_bbox5 NOT NULL,
CONSTRAINT pk_bbox PRIMARY KEY (bounding_box_ID));
CREATE SEQUENCE seq_bbox;

CREATE TABLE Font
( font_ID       NUMBER(30)    CONSTRAINT nn_font1 NOT NULL,
  font_size     NUMBER(20,10) CONSTRAINT nn_font2 NOT NULL,
  font_family   VARCHAR2(30)  CONSTRAINT nn_font3 NOT NULL,
  is_bold       VARCHAR2(1)   CONSTRAINT nn_font4 NOT NULL,
  is_italic     VARCHAR2(1)   CONSTRAINT nn_font5 NOT NULL,
  is_underlined VARCHAR2(1)   CONSTRAINT nn_font6 NOT NULL,
CONSTRAINT pk_font PRIMARY KEY (font_ID));
CREATE SEQUENCE seq_font;
```

19.13 This exercise is similar to the edge-node problem from Exercise 19.1. *City* is analogous to *Node* and *Route* is analogous to *Edge*. We infer that a *Route* has 2 *Cities* from the problem statement. We could not deduce that from the SQL code alone.

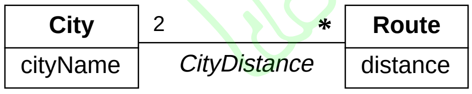


Figure A19.12 Class model for Figure E19.6

19.14 SQL code to determine distance between two cities for Figure E19.6. This code is similar to that for Exercise 19.5c that determines the edge for a pair of nodes.

```
SELECT distance
FROM Route R, City C1, City C2,
     City_Distance CD1, City_Distance CD2
WHERE C1.city_ID = CD1.city_ID AND
      CD1.route_ID = R.route_ID AND
      R.route_ID = CD2.route_ID AND
      CD2.city_ID = C2.city_ID  AND
      C1.city_name = :aCityName1  AND
      C2.city_name = :aCityName2;
```

19.15 The class model in Figure A19.13 corresponds to the SQL code in Figure E19.7. Note that this model is similar to Figure E19.1a. Figure A19.13 is a better model than Figure E19.1a; each pair of cities has a single value of distance as Figure A19.13 correctly states; however each pair of nodes corresponds to many edges and not one edge as Figure E19.1a shows.

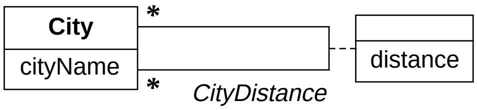


Figure A19.13 Class model for Figure E19.7

19.16 SQL code to determine distance between two cities for Figure E19.7. We don't know which name is *1* and which name is *2*, so the SQL code allows for either possibility.

```
SELECT distance
FROM City C1, City C2, City_Distance CD
WHERE C1.city_ID  = CD.city1_ID AND
      CD.city2_ID = C2.city_ID  AND
      ((C1.cityName = :aCityName1 AND
        C2.cityName = :aCityName2) OR
       (C1.cityName = :aCityName2 AND
        C2.cityName = :aCityName1));
```

19.17 We make the following observations about Figure A19.12 and Figure A19.13.

- Figure A19.12 has an additional table. Figure A19.12 could store multiple routes between the same cities with different distances. Given the lack of explanation about route in the problem statement (is it a series of roads with different distances or is it the distance by air?) this may or may not be a drawback.
- Figure A19.13 is awkward because of the symmetry between city1 and city2. Either data must be stored twice with waste of storage, update time, and possible consistency problems, or special application logic must enforce an arbitrary constraint.

We need to know more about the requirements to choose between the models.

19.18 The city-route problem is essentially an undirected graph. Two cities relate to a route and there is no natural way to distinguish the cities. We normally discourage symmetrical models for undirected graphs (such as Figure A19.13) because they are confusing, lead to possible redundancy, and complicate search and update code.

In contrast, the *Edge-Node* problem is essentially a directed graph. In a directed graph there is a *from* node and a *to* node.

As specified, a given pair of cities has only a single value of distance. In contrast, two nodes may have any number of edges between them. Thus Figure A19.13 is correct (though discouraged). The class model in Figure E19.1a is wrong.

19.19 Figure A19.14 shows our tables.

Meet table

meetID	date	location

RawScore table

trialID (references Trial)	**judgeID** (references Judge)	value

Event table

eventID	startingTime	meetID (references Meet)	stationID (references Station)

Judge table

judgeID	name

Station_Judge table

stationID (references Station)	**judgeID** (references Judge)

Station table

stationID	location	meetID (references Meet)

Trial table

trialID	netScore	eventID (references Event)	competitorID (references Competitor)

Competitor table

competitorID	name	age	address	telephoneNumber

Figure A19.14 Tables for scoring system problem

SQL commands to create an empty database.

```
CREATE TABLE Meet
( meet_ID       NUMBER(30)    CONSTRAINT nn_meet1 NOT NULL,
  meet_date     DATETIME,
  location      VARCHAR2(255),
CONSTRAINT pk_meet PRIMARY KEY (meet_ID));
CREATE SEQUENCE seq_meet;

CREATE TABLE RawScore
( trial_ID      NUMBER(30) CONSTRAINT nn_rs1 NOT NULL,
  judge_ID      NUMBER(30) CONSTRAINT nn_rs2 NOT NULL,
  raw_score    NUMBER(10),
CONSTRAINT pk_rs PRIMARY KEY (trial_ID, judge_ID));

CREATE INDEX index_rs1 ON RawScore (judge_ID);

ALTER TABLE RawScore ADD CONSTRAINT fk_rs1
  FOREIGN KEY trial_ID
  REFERENCES Trial;

ALTER TABLE RawScore ADD CONSTRAINT fk_rs2
  FOREIGN KEY judge_ID
  REFERENCES Judge;

CREATE TABLE Event
( event_ID      NUMBER(30) CONSTRAINT nn_event1 NOT NULL,
  starting_time DATETIME,
  meet_ID       NUMBER(30) CONSTRAINT nn_event2 NOT NULL,
  station_ID    NUMBER(30) CONSTRAINT nn_event3 NOT NULL,
CONSTRAINT pk_event PRIMARY KEY (event_ID));

CREATE SEQUENCE seq_event;

CREATE INDEX index_event1 ON Event (meet_ID);

CREATE INDEX index_event2 ON Event (station_ID);

ALTER TABLE Event ADD CONSTRAINT fk_event1
  FOREIGN KEY meet_ID
  REFERENCES Meet;

ALTER TABLE Event ADD CONSTRAINT fk_event2
  FOREIGN KEY station_ID
  REFERENCES Station;

CREATE TABLE Judge
( judge_ID    NUMBER(30)    CONSTRAINT nn_judge1 NOT NULL,
  judge_name  VARCHAR2(50),
CONSTRAINT pk_judge PRIMARY KEY (judge_ID));

CREATE SEQUENCE seq_judge;

CREATE TABLE Station
( station_ID NUMBER(30)   CONSTRAINT nn_station1 NOT NULL,
  location   VARCHAR2(255),
  meet_ID    NUMBER(30)   CONSTRAINT nn_station2 NOT NULL,
CONSTRAINT pk_station PRIMARY KEY (station_ID));

CREATE SEQUENCE seq_station;

CREATE INDEX index_station1 ON Station (meet_ID);

ALTER TABLE Station ADD CONSTRAINT fk_station1
  FOREIGN KEY meet_ID
  REFERENCES Meet;
```

```
CREATE TABLE Station_Judge
( station_ID    NUMBER(30) CONSTRAINT nn_sj1 NOT NULL,
  judge_ID      NUMBER(30) CONSTRAINT nn_sj2 NOT NULL,
CONSTRAINT pk_sj PRIMARY KEY (station_ID, judge_ID));

CREATE INDEX index_sj1 ON Station_Judge (judge_ID);

ALTER TABLE Station_Judge ADD CONSTRAINT fk_sj1
  FOREIGN KEY station_ID
  REFERENCES Station;

ALTER TABLE Station_Judge ADD CONSTRAINT fk_sj2
  FOREIGN KEY judge_ID
  REFERENCES Judge;

CREATE TABLE Trial
( trial_ID      NUMBER(30) CONSTRAINT nn_trial1 NOT NULL,
  net_score     NUMBER(10),
  event_ID      NUMBER(30) CONSTRAINT nn_trial2 NOT NULL,
  competitor_ID NUMBER(30) CONSTRAINT nn_trial3 NOT NULL,
CONSTRAINT pk_trial PRIMARY KEY (trial_ID));

CREATE SEQUENCE seq_trial;

CREATE INDEX index_trial1 ON Trial (event_ID);

CREATE INDEX index_trial2 ON Trial (competitor_ID);

ALTER TABLE Trial ADD CONSTRAINT fk_trial1
  FOREIGN KEY event_ID
  REFERENCES Event;

ALTER TABLE Trial ADD CONSTRAINT fk_trial2
  FOREIGN KEY competitor_ID
  REFERENCES Competitor;

CREATE TABLE Competitor
( competitor_ID NUMBER(30)   CONSTRAINT nn_comp1 NOT NULL,
  comp_name     VARCHAR2(50) CONSTRAINT nn_comp2 NOT NULL,
  age           NUMBER(3)    CONSTRAINT nn_comp3 NOT NULL,
  address       VARCHAR2(255),
  telephone_number VARCHAR2(20),
CONSTRAINT pk_comp PRIMARY KEY (competitor_ID));

CREATE SEQUENCE seq_comp;
```

19.20 Figure A19.15 is the same class model that we used in our answer in Chapter 3.

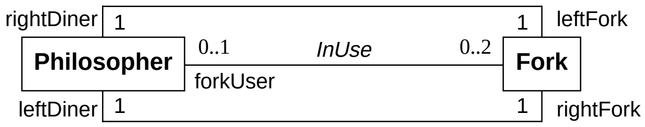
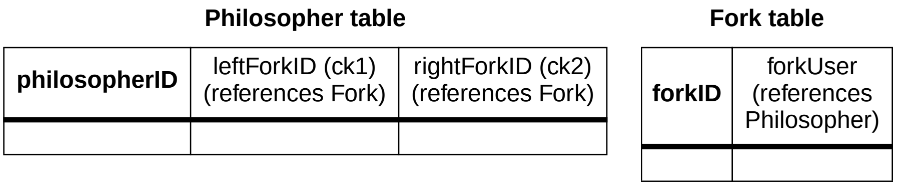


Figure A19.15 Class model for dining philosopher's problem

The mapping rules yield Figure A19.16. and the following SQL commands. (Alternatively, you could have buried the *leftDiner* and *rightDiner* in the *Fork* table.)



Philosopher table

philosopherID	leftForkID (ck1) (references Fork)	rightForkID (ck2) (references Fork)

Fork table

forkID	forkUser (references Philosopher)

Figure A19.16 Tables for dining philosopher's problem

```
CREATE TABLE Philosopher
( philosopher_ID NUMBER(30) CONSTRAINT nn_phil1 NOT NULL,
  left_fork_ID   NUMBER(30) CONSTRAINT nn_phil2 NOT NULL,
  right_fork_ID  NUMBER(30) CONSTRAINT nn_phil3 NOT NULL,
CONSTRAINT pk_phil PRIMARY KEY (philosopher_ID),
CONSTRAINT uq_phil1 UNIQUE (left_fork_ID),
CONSTRAINT uq_phil2 UNIQUE (right_fork_ID));

CREATE SEQUENCE seq_phil;

ALTER TABLE Philosopher ADD CONSTRAINT fk_phil1
  FOREIGN KEY left_fork_ID
  REFERENCES Fork;

ALTER TABLE Philosopher ADD CONSTRAINT fk_phil2
  FOREIGN KEY right_fork_ID
  REFERENCES Fork;


CREATE TABLE Fork
( fork_ID    NUMBER(30) CONSTRAINT nn_fork1 NOT NULL,
  fork_user  NUMBER(30),
CONSTRAINT pk_fork PRIMARY KEY (fork_ID));

CREATE SEQUENCE seq_fork;

CREATE INDEX index_fork1 ON Fork (fork_user);
```

```
ALTER TABLE Fork ADD CONSTRAINT fk_fork1
  FOREIGN KEY fork_user
  REFERENCES Philosopher;
```

Figure A19.17 shows database contents for the case where each philosopher has the left fork.

Philosopher table

philosopher_ID	left_fork_ID	right_fork_ID
1	15	11
2	11	12
3	12	13
4	13	14
5	14	15

Fork table

fork_ID	fork_user
11	2
12	3
13	4
14	5
15	1

Figure A19.17 Populated database table for dining philosopher's problem

19.21 Figure A19.18 shows our tables.

Here are SQL commands to create an empty database.

```
CREATE TABLE Customer
( customer_ID NUMBER(30)   CONSTRAINT nn_cust1 NOT NULL,
  cust_name   VARCHAR2(50) CONSTRAINT nn_cust2 NOT NULL,
CONSTRAINT pk_cust PRIMARY KEY (customer_ID));

CREATE SEQUENCE seq_cust;

CREATE INDEX index_cust1 ON Customer (cust_name);

CREATE TABLE Mailing_Address
( mailing_addr_ID NUMBER(30) CONSTRAINT nn_ma1 NOT NULL,
  address         VARCHAR2(255),
  phone_number    VARCHAR2(20),
CONSTRAINT pk_ma PRIMARY KEY (mailing_addr_ID));

CREATE SEQUENCE seq_ma;

CREATE TABLE Customer__Mailing_Address
( account_holder  NUMBER(30) CONSTRAINT nn_cma1 NOT NULL,
  mailing_addr_ID NUMBER(30) CONSTRAINT nn_cma2 NOT NULL,
CONSTRAINT pk_cma PRIMARY KEY
  (account_holder, mailing_addr_ID));
```

Customer table

customerID	name

MailingAddress table

mailingAddressID	address	phoneNumber

Customer_MailingAddress table

accountHolder (references Customer)	**mailingAddressID** (references MailingAddress)

Institution table

institutionID	name	address	phone Number

CreditCardAccount table

creditCard AccountID	maximum Credit	current Balance	mailingAddressID (references MailingAddress)	institutionID (ck1) (references Institution)	account Number (ck1)

Statement table

statementID	payment DueDate	finance Charge	minimum Payment	creditCardAccountID (ck1) (references CreditCardAccount)	statement Date (ck1)

Transaction table

transac tionID	transaction Date	explan ation	amount	transaction Type	statement ID (ck1)	transaction Number (ck1)

CashAdvance table

cashAdvanceID (references Transaction)

Interest table

interestID (references Transaction)

Purchase table

purchaseID (references Transaction)	merchantID (references Merchant)

Fee table

feeID (references Transaction)	fee Type

Adjustment table

adjustmentID (references Transaction)

Merchant table

merchantID	name

Figure A19.18 Tables for managing credit card accounts

```
CREATE INDEX index_cma1 ON Customer__Mailing_Address
  (mailing_addr_ID);

ALTER TABLE Customer__Mailing_Address
  ADD CONSTRAINT fk_cma1
  FOREIGN KEY account_holder
  REFERENCES Customer ON DELETE CASCADE;

ALTER TABLE Customer__Mailing_Address
  ADD CONSTRAINT fk_cma2
  FOREIGN KEY mailing_addr_ID
  REFERENCES Mailing_Address;

CREATE TABLE Institution
( institution_ID NUMBER(30)   CONSTRAINT nn_inst1 NOT NULL,
  inst_name      VARCHAR2(50) CONSTRAINT nn_inst2 NOT NULL,
  address        VARCHAR2(255),
  phone_number   VARCHAR2(20),
CONSTRAINT pk_inst PRIMARY KEY (institution_ID));

CREATE SEQUENCE seq_inst;

CREATE TABLE Credit_Card_Account
( ccard_account_ID NUMBER(30) CONSTRAINT nn_cca1 NOT NULL,
  maximum_credit   NUMBER(12,2),
  current_balance  NUMBER(12,2),
  mailing_addr_ID  NUMBER(30) CONSTRAINT nn_cca2 NOT NULL,
  institution_ID  NUMBER(30)   CONSTRAINT nn_cca3 NOT NULL,
  account_num     VARCHAR2(20) CONSTRAINT nn_cca4 NOT NULL,
CONSTRAINT pk_cca PRIMARY KEY (ccard_account_ID),
CONSTRAINT uq_cca1 UNIQUE (institution_ID, account_num));

CREATE SEQUENCE seq_cca;

CREATE INDEX index_cca1 ON Credit_Card_Account
  (mailing_addr_ID);

ALTER TABLE Credit_Card_Account ADD CONSTRAINT fk_cca1
  FOREIGN KEY mailing_addr_ID
  REFERENCES Mailing_Address;

ALTER TABLE Credit_Card_Account ADD CONSTRAINT fk_cca2
  FOREIGN KEY institution_ID
  REFERENCES Institution;

CREATE TABLE Statement
( statement_ID     NUMBER(30) CONSTRAINT nn_stat1 NOT NULL,
  payment_due_date DATETIME   CONSTRAINT nn_stat2 NOT NULL,
  finance_charge NUMBER(12,2) CONSTRAINT nn_stat3 NOT NULL,
  minimum_paymt  NUMBER(12,2) CONSTRAINT nn_stat4 NOT NULL,
  ccard_account_ID NUMBER(30) CONSTRAINT nn_stat5 NOT NULL,
  statement_date  DATETIME   CONSTRAINT nn_stat6 NOT NULL,
CONSTRAINT pk_stat PRIMARY KEY (statement_ID),
CONSTRAINT uq_stat1
  UNIQUE (ccard_account_ID, statement_date));

CREATE SEQUENCE seq_statemt;

ALTER TABLE Statement ADD CONSTRAINT fk_statemt1
  FOREIGN KEY ccard_account_ID
  REFERENCES Credit_Card_Account;


CREATE TABLE Transaction
( transact_ID   NUMBER(30)   CONSTRAINT nn_trans1 NOT NULL,
  transact_date DATETIME     CONSTRAINT nn_trans2 NOT NULL,
  explanation   VARCHAR2(255),
  amount       NUMBER(12,2) CONSTRAINT nn_trans3 NOT NULL,
  transact_type VARCHAR2(20) CONSTRAINT nn_trans4 NOT NULL,
  statement_ID  NUMBER(30)  CONSTRAINT nn_trans5 NOT NULL,
  transact_num  VARCHAR2(20) CONSTRAINT nn_trans6 NOT NULL,
CONSTRAINT pk_trans PRIMARY KEY (transact_ID),
CONSTRAINT uq_trans1 UNIQUE (statement_ID, transact_num));

CREATE SEQUENCE seq_trans;

ALTER TABLE Transaction ADD CONSTRAINT fk_trans1
  FOREIGN KEY statement_ID
  REFERENCES Statement;

ALTER TABLE Transaction ADD CONSTRAINT enum_trans1
  CHECK (transact_type IN (’Cash_Advance’, ’Interest’,
  ’Purchase’, ’Fee’, ’Adjustment’));


CREATE TABLE Cash_Advance
( cash_advance_ID NUMBER(30) CONSTRAINT nn_casha1 NOT NULL,
CONSTRAINT pk_casha PRIMARY KEY (cash_advance_ID));

ALTER TABLE Cash_Advance ADD CONSTRAINT fk_casha1
  FOREIGN KEY cash_advance_ID
  REFERENCES Transaction ON DELETE CASCADE;


CREATE TABLE Interest
( interest_ID  NUMBER(30) CONSTRAINT nn_interest1 NOT NULL,
CONSTRAINT pk_interest PRIMARY KEY (interest_ID));

ALTER TABLE Interest ADD CONSTRAINT fk_interest1
  FOREIGN KEY interest_ID
  REFERENCES Transaction ON DELETE CASCADE;


CREATE TABLE Adjustment
( adjustment_ID NUMBER(30) CONSTRAINT nn_adjust1 NOT NULL,
CONSTRAINT pk_adjust PRIMARY KEY (adjustment_ID));
```

```
ALTER TABLE Adjustment ADD CONSTRAINT fk_adjust1
  FOREIGN KEY adjustment_ID
  REFERENCES Transaction ON DELETE CASCADE;

CREATE TABLE Purchase
( purchase_ID  NUMBER(30) CONSTRAINT nn_purchase1 NOT NULL,
  merchant_ID  NUMBER(30) CONSTRAINT nn_purchase2 NOT NULL,
CONSTRAINT pk_purchase PRIMARY KEY (purchase_ID));

CREATE INDEX index_purchase1 ON Purchase (merchant_ID);

ALTER TABLE Purchase ADD CONSTRAINT fk_purchase1
  FOREIGN KEY purchase_ID
  REFERENCES Transaction ON DELETE CASCADE;

ALTER TABLE Purchase ADD CONSTRAINT fk_purchase2
  FOREIGN KEY merchant_ID
  REFERENCES Merchant;

CREATE TABLE Fee
( fee_ID        NUMBER(30)   CONSTRAINT nn_fee1 NOT NULL,
  fee_Type      VARCHAR2(20) CONSTRAINT nn_fee2 NOT NULL,
CONSTRAINT pk_fee PRIMARY KEY (fee_ID));

ALTER TABLE Fee ADD CONSTRAINT fk_fee1
  FOREIGN KEY fee_ID
  REFERENCES Transaction ON DELETE CASCADE;

CREATE TABLE Merchant
( merchant_ID   NUMBER(30)   CONSTRAINT nn_merch1 NOT NULL,
  merchant_name VARCHAR2(50) CONSTRAINT nn_merch2 NOT NULL,
CONSTRAINT pk_merch PRIMARY KEY (merchant_ID));

CREATE SEQUENCE seq_merch;

CREATE INDEX index_merch1 ON Merchant (merchant_name);
```

19.22 Here are the SQL queries.

- What transactions occurred for a credit card account within a time interval?

```
SELECT transact_ID
FROM Credit_Card_Account CCA, Statement S, Transaction T
WHERE CCA.ccard_account_ID = S.ccard_account_ID AND
      S.statement_ID = T.statement_ID AND
      T.transact_date >= :aStartDate AND
      T.transact_date <= :anEndDate;
```

- What volume of transactions were handled by an institution in the last year?

```
SELECT sum (T.amount)
FROM Institution I, Credit_Card_Account CCA, Statement S,
     Transaction T
```

```
WHERE I.institution_ID = CCA.institution_ID AND
      CCA.ccard_account_ID = S.ccard_account_ID AND
      S.statement_ID = T.statement_ID AND
      T.transact_date >= :aStartDate AND
      T.transact_date <= :anEndDate;
```

■ What customers patronized a merchant in the last year by any kind of credit card?

```
SELECT DISTINCT customer_ID
FROM Merchant M, Purchase P, Transaction T, Statement S,
     Credit_Card_Account CCA, Mailing_Address MA,
     Customer__Mailing_Address CMA, Customer C
WHERE M.merchant_ID = P.merchant_ID AND
      P.purchase_ID = T.transaction_ID AND
      T.statement_ID = S.statement_ID AND
      S.ccard_account_ID = CCA.ccard_account_ID AND
      CCA.mailing_addr_ID = MA.mailing_addr_ID AND
      MA.mailing_addr_ID = CMA.mailing_addr_ID AND
      CMA.customer_ID = C.customer_ID AND
      T.transact_date >= :aStartDate AND
      T.transact_date <= :anEndDate;
```

■ How many credit card accounts does a customer currently have?

```
SELECT COUNT(*)
FROM Customer C, Customer__Mailing_Address CMA,
     Mailing_Address MA, Credit_Card_Account CCA
WHERE C.customer_ID = CMA.customer_ID AND
      CMA.mailing_addr_ID = MA.mailing_addr_ID AND
      MA.mailing_addr_ID = CCA.mailing_addr_ID;
```

■ What is the total maximum credit for a customer?

```
SELECT sum (maximum_credit)
FROM Customer C, Customer__Mailing_Address CMA,
     Mailing_Address MA, Credit_Card_Account CCA
WHERE C.customer_ID = CMA.customer_ID AND
      CMA.mailing_addr_ID = MA.mailing_addr_ID AND
      MA.mailing_addr_ID = CCA.mailing_addr_ID;
```

20

Programming Style

20.1 The operations depend on the class of the node. For classes with an attribute that contributes to a total, the operation *total()* is appropriate. For geometrical classes, the operations *move(offset)* and *rotate(angle)* could be applied.

We did not say in the problem statement how to pass any arguments of the operations. This could be done via a global variable or by passing arguments in a data structure. We will assume the latter, and have modified *orderedVisit* accordingly. The following is pseudocode for one way for performing *orderedVisit*:

```
orderedVisit (node, method, arguments)
{
   if node is not NULL
   {
      orderedVisit (node.leftSubtree, method, arguments);
      apply method (arguments) to node;
      orderedVisit (node.rightSubtree, method, arguments);
   }
}
```

20.2 The first operation is a special case of the second operation. The distinction between the two types of accounts can be eliminated by treating a normal account as a reserve account with a reserve limit of zero. Attributes needed to track accounts include *balance*, *reserveBalance*, and *reserveLimit*.

20.3a. This is an example of poor programming style. The assumption that the arguments are legal and the functions called are well behaved will cause trouble during program test and integration.

The following statements will cause the program to crash if the argument to *strlen* is zero:

```
rootLength = strlen(rootName);
suffixLength = strlen(suffix);
```

The following statement will assign zero to *sheetName* if the program runs out of memory, causing a program crash during the call to *strcpy* later in the function:

```
sheetName = malloc(rootLength + suffixLength + 1);
```

The following statements will cause the program to crash if any of the arguments are zero:

```
sheetName = strcpy(sheetName, rootName);
sheetName = strcat(sheetName, suffix);
```

If *sheetType* is invalid the switch statement will fall through leaving *sheet* without an assigned value. Also, it is possible that the call to *vertSheetNew* or the call to *horizSheetNew* could return zero for some reason. Either condition would make it possible for the following statement to crash:

```
sheet->name = sheetName;
```

b. The revised function is given below. The function *reportError(message)* reports errors.

```
Sheet createSheet (sheetType, rootName, suffix)
SheetType sheetType;
char *rootName, *suffix;
{   char *malloc(), *strcpy(), *strcat(), *sheetName;
    int strlen(), rootLength, suffixLength;
    Sheet sheet, vertSheetNew(), horizSheetNew();
    if (rootName)
        rootLength = strlen(rootName);
    else
    {
        rootName = "";
        rootLength = 0;
    }
    if (suffix)
        suffixLength = strlen(suffix);
    else
    {
        suffix = "";
        suffixLength = 0;
    }
    sheetName = malloc(rootLength + suffixLength + 1);
    if (sheetName)
```

```
        {
           sheetName = strcpy(sheetName, rootName);
           sheetName = strcat(sheetName, suffix);
        }
        else
        {
           sheetName = "";
           reportError("Out of memory.");
        }
        switch (sheetType)
        {  case VERTICAL:
              sheet = vertSheetNew();
              break;
           case HORIZONTAL:
              sheet = horizSheetNew();
              break;
           default:
              sheet = vertSheetNew();
              break;
        }
        if (sheet)
           sheet->name = sheetName;
        else
           reportError("Failure to create sheet.");
        return sheet;
     }
```

The revised function will not crash under error conditions. If a sheet is not created, zero is returned, and the calling routine must properly handle the failure.