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.

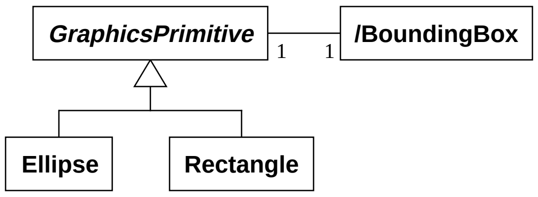

Figure A15.1 Revised class diagram for a bounding box

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.

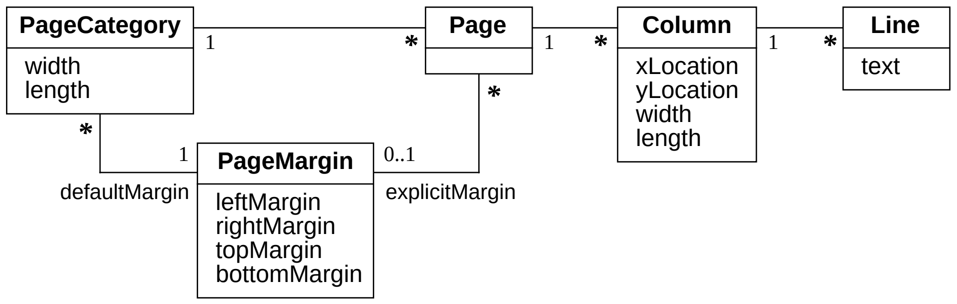

Figure A15.2 Portion of a newspaper model with a separate class for margins

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.

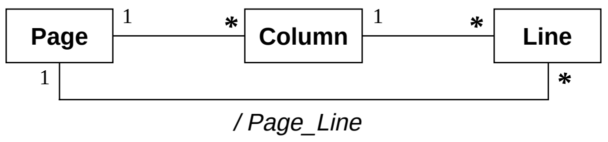

Figure A15.3 A revised newspaper model that can directly determine the page for a line

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-

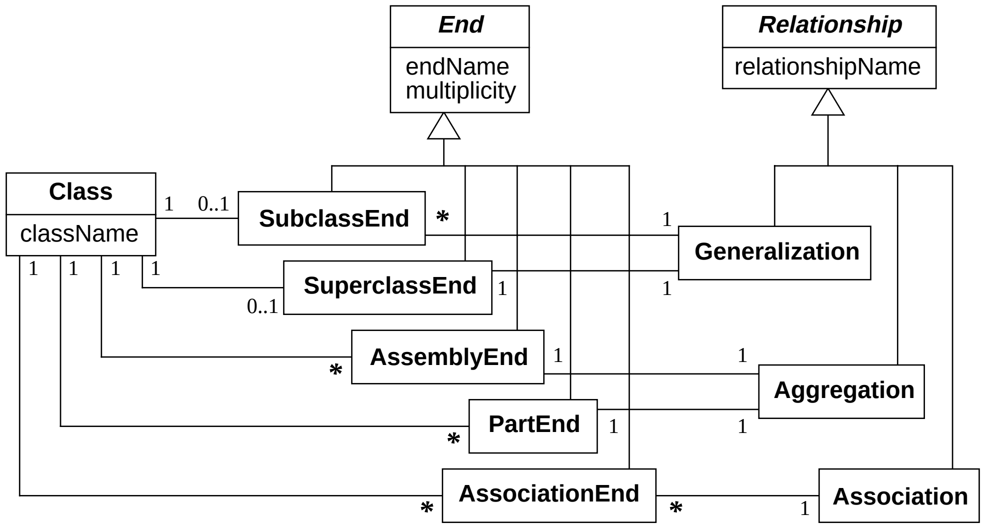

Figure A15.4 A revised model with associations to the subclasses

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.

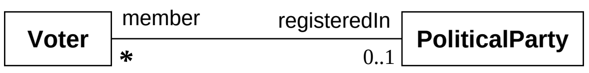

Figure A15.5 A revised model that reifies political party

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.

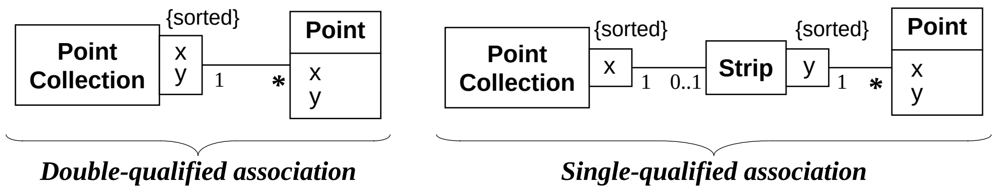

Figure A15.6 Models for sorted collections of points

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.