ADS for a Square

An **abstract data structure** is a collection of data structures, functions, and relationships which characterize a mathematical object. It is the same as an object and its methods in object-oriented programming. Abstract data structures are used to isolate the implementation of a mathematical structure from its purely mathematical definition.

The structure *square* has a collection of functions, predicates, variables, and axioms which define it as a *Square*. But it can be implemented in a dozen ways, as an array, a list, an ordered list, etc. Implementing an abstract data structure consists of defining the collection of mathematical properties and then choosing a particular implementation approach. An **abstraction barrier** is an implementation technique which allows an abstract object to be called by an application program, while keeping the implementation details completely separate.

CONCEPTUALIZATION

This phase is figuring out what the structure in question is. The exact characteristics of an abstract structure depend upon how that structure is to be used. For example, if the diagonal of a square is never used in an application, then a characteristic such as the length of the diagonal does not need to be included in the abstraction. It is often appropriate to include as many characteristics as possible in the implementation of an abstract structure, but most structures have thousands of possible characteristics. Thus, the abstraction technique makes modular extension of an abstraction easy.

In the conceptualization of a *square*, we identify what we think a square is.

- a visual object with four corner points and four sides on a plane
- the length of each side is variable
- definition: fours sides are of equal length and perpendicular
- alternative definition: interior angles are 90 degrees, and adjacent sides are equal

The idea is to identify the minimal attributes which completely define the object in question.

Note that a conceptualization often needs other concepts to make sense. We might assume, for example, that we need to actually see the square, so we identify a drawing capability to generate the picture of the square. This drawing capability is *not* part of the square. The role of the conceptualization phase is to sort out what is object and what is context. There are no right or wrong assumptions, only useful and clumsy ones.

The first attachment to this handout illustrates a type hierarchy of two-dimensional geometric figures. It shows that in an object-oriented approach, the square has multiple supertypes. Implementation of upbranching type hierarchies is particularly difficult. Designers of the language JAVA, for example, decided against including multiple inheritance. This is an example of the language restricting the type of modeling available. To include multiple inheritance in a model, a programmer would have to explicitly choose a language which includes it, since implementing multiple inheritance oneself, as a customized extension, is too difficult an undertaking.

There are always alternative ways to conceptualize the square, focussing on different sets of component objects.

Let's assume we have a way to draw a line, e.g.: a draw function,

```
DrawLine[point1,point2]
```

This function takes points as input, so that it somewhat predetermines how we think about the square itself: the square must contain points that permit it to be drawn. Drawing a square requires four lines to be drawn. Alternatively we may have a multiple point draw function:

```
DrawPolyline[point1,point2,point3,point4,point5]
```

We also need to assume a place to stand to observe the square. This is the idea of a coordinate system and an underlying space which can hold an object such as a square. E.g.: assume an integer Cartesian grid with origin at (0,0). This raises a central question: where is the origin with regard to the square?

The distinction between *coordinate-free* and *embedded-in-a-coordinate-system* is critical for abstraction. A bare square does not have a particular corner point which is its origin. An embedded square is in a space which has a special point called the origin of the space. Likewise, we can assume that the square is embedded in at least two dimensions (by definition), but a square can still exist in higher dimensions. Both **dimensionality** and **coordinate-system** are external to the square abstraction.

So *one conceptualization of a square* is:

> A compound object in 2 dimensions, consisting of four lines which intersect at four points, such that the line segments between the points are of equal length and the adjacent lines are perpendicular.

Thus we need to find a mathematical model which gives us a way to describe length in 2D and a notion of orthogonality.

Technique: It is almost always a good idea to construct a *canonical* representation, here the standard reference square. Since size is a parameter, it can be abstracted away (standardized) for the canonical square. Another way of thinking about this is that the canonical form is the private part of an object definition, while the parameter is the public part. Canonical forms are usually sorted, or arranged in order, for ease of processing. Global parameters are abstracted, and base values are set to either 1 or 0.

```
Object SQUARE consists of
     2D-points:       tl tr bl br       (tl=top-left, br=bottom-right)
     lines:           t l b r           (top, left, bottom, right)
     area:            a                 real number * real number
     size:            s                 real number

   Base:
     unit-square:     u
```

```
          Predicates:
            is-a-square[p1,p2,p3,p4]
            perpendicular[l1,l2]
            parallel[l1,l2]
            on[p,l]

          Functions:
            line-from-points[p1,p2]       => line
            points-from-line[l]           => p1, p2
            area-of[s]                    => square units
            hamming-distance[p1,p2]       => integer
```

The SQUARE uses other objects such as points and lines. These would have their own definitions, providing their own recognizers and accessors. For example:

```
      Object 2D-POINT consists of
            locations:  x1 x2                    real number
          Base:
            origin:  (0,0)
          Predicates:
            is-a-2D-point[p]
            is-the-origin[p]
          Functions:
            quadrant-of[p]                       =>  quadrant I, II, III, or IV
```

Technique: Define what you want, not what you do not what.

Note that the question of dimensionality (is the square planar?) is completely finessed by building in only with 2D points. Similarly, things like acute angles are simply disallowed by not providing for them.

Technique: Numbers are often a base type. Mathematical structures usually assume real numbers, but computers can handle at best only rational numbers. Choose the type of number that you use for modeling very carefully, using the minimal structure to achieve the purposes of the transformation.

MODEL

Technique: Choose a model that minimizes complexity.

The standard Cartesian coordinate system provides both an orthogonality and a distance concept for the square's definition.

Elect to put the origin at one of the square's corner points. [This is not the only good choice. Putting the origin in the center of the square makes rotation around the center easier. The point is that the choice of a simple model is deeply connected with the permitted transformations of that model.]

Elect to orient the square so that one side is along each coordinate axis. The canonical square would have sides of length 1, so that using orthogonal unit vectors, the four corner points have locations

```
    bl = (0,0)        br = (1,0)        tr = (1,1)        tl = (0,1).
```

I've elected to name the points also. Names are free, so some semantic reminders can be built into the names. I've used bottom-left, bottom-right, top-left, top-right.

Unit-vector model:

The Cartesian grid has orthogonality built-in as a primitive, but this does not tell you how to compute with it. From calculus, we might recall that the *unit vector* is the math structure underneath the concept of orthogonal axes in the Cartesian plane.

A 2D-vector is a set of two values specifying a distance and a direction. In polar coordinates, a point at the end of the vector is represented by (distance, angle). Alternatively, the location of a point can be expressed in Cartesian coordinates, as the composition of two orthogonal basis vectors (i, j), which are perpendicular by construction.

```
x  =  (1i,0j)
y  =  (0i,1j)
```

The definition of perpendicular is:

```
i·j = 0              (dot-product)
i·i = j·j = 1
```

Vectors are convenient to process using matrix algebra. Unit vectors provide a math model which makes conceptualization easy. Now we switch math models to make computation easy. The square is now a matrix (rows are corner points, columns are unit vectors):

```
               [0 0]                                          [0 0]
unit-square =  [1 0]          parameterized-square = size  ·  [1 0]
               [1 1]                                          [1 1]
               [0 1]                                          [0 1]
```

If we want to move the square (without rotation), for example so that point (0,0) is now at point (a,b), then it suffices to add the new location to the old for each point before scaling:

```
[0 0]                        [ a    b ]
[1 0] + [a b]      =         [a+1   b ]
[1 1]                        [a+1  b+1]
[0 1]                        [ a   b+1]
```

Vector calculus is not necessary here, since it is just one model of how a Cartesian plane works. We could just as well use high school geometry or algebra.

Size:

The canonical square has sides of unit 1. The size parameter can be factored out of the canonical form as a multiplier of the entire unit square:

```
size · {(0,0),(1,0),(1,1),(0,1)}
```

Thus, the square is the set of points:

```
{(0,0),(size,0),(size,size),(0,size)}
```

We say that a function is *listable* when it can be applied to or abstracted from the elements of a nested list. The example here is multiplying a nested list which represents the square by a scalar which represents the size parameter.

IMPLEMENTATION

Now we must implement the functions and predicates in the conceptualization. If the math model is selected carefully, it will provide the operators for the implementation (but it will not provide an efficient selection of data structure).

Example: `perpendicular[l1,l2]`

Our lines are defined by two points each. One method would be to use the formula for the slope of lines (in terms of two points defining the line) and the relationship that perpendicular lines have slopes of the form

```
m1 * m2 = -1
```

This is too complicated since it tests lines in any orientation, for any angle of intersection. Since all lines in the square are either perpendicular or parallel, we could just test for parallel lines, with

```
perpendicular[l1,l2] = not[parallel[l1,l2]]
```

Parallel lines are easier to determine. Two sides of a square are parallel when their slopes are equal, `m1 = m2`.

However, the easiest approach is recall that we have only squares in our model. So all lines are sides of a square. Sides are perpendicular if the have a corner point in common. Sides are parallel if they share no corner points.
In pseudo-code:

```
perpendicular[(p1,p2),(p3,p4)] =def=
      p1=p3 or p1=p4 or p2=p3 or p2=p4
```

If the points in the line data structure are always ordered, then this test becomes even easier:

```
      p1=p4 or p2=p3
```

TRANSFORMATION:

The selection of a model and an implementation depends greatly on how an object is used. Let's assume that we want to do three things with the square:

```
Translate[sq,distance]
Rotate[sq,angle]
Scale[sq,size]
```

If the representation of the square is the unit-square, with a parameterized scaling factor (i.e. size)

```
sq = size · {(0,0),(1,0),(1,1),(0,1)}
```

then these transformations are easy to model:

```
Translate[sq,d]  = d + sq
Scale[sq,s]      = s * sq
Rotate[sq,theta] =
  origin-at-center = Translate[sq,(-1/2,-1/2)];
  radius = SquareRoot[x^2 + y^2];            ;any point is ok by symmetry
  for all points in origin-at-center:
    new-x = radius * cos[theta]
    new-y = radius * sin[theta]
```

Rotation becomes complex because

1) we adopted Cartesian coordinates, when polar coordinates make rotation simpler, and
2) we can rotate around any arbitrary point.

The default above is to rotate around the origin point, but the intuitive default would be to rotate around the center of the square. Thus the origin needs to be translated before the rotation.

TRANSFORMATION DEFINING NEW PROPERTIES

Rotation of the square lets us observe some of its symmetry properties. For example, rotation by 90 degrees generates a unit-square with the same set of corner points, in a different order. But since the points are a Set, order doesn't matter. That is, rotation by 90 degrees is equivalent just to renaming the corners. Since the choice of names is free, rotation by 90 degrees is the same as no change at all to the square. An implementation could efficiently use:

```
Rotate[sq,90] = sq
```

This observation is the beginning of exploring the group structure of the square. We have moved from graphics to algebra to vectors to group theory, different mathematical models of the conceptualization of a square.

CODE DECISIONS

Without concern for optimization, we can see that almost any transformation of the square involves adding or multiplying the set of points which define the particular square. After the pseudo-code for all functions and predicates has been constructed, we will have a very good idea of the types and frequencies of mathematical operations (just count the number of times a data structure is accessed, multiplied, etc.). The implementation data structure should make these frequent operations computationally easy. But also fundamental to an abstract data type, the interface to the implementation must be mathematical, not computational. That is, the mathematical functionality must be partitioned from the implementation choice of data structure.

I'd put each square into an array consisting of the name and the eight rational numbers defining corner points:

```
make-square[name,sq] =
  sq-name := make-array[9];
  sq-name[0] := name;
  sq-name[1] := blx-of-sq;              ;blx is bottom-left-x value
  ...
  sq-name[8] := tly-of-sq;
```

An example function:

```
Translate[sq,(dx,dy)] =
  for i= 1 to 8, do
    when odd[i]  sq-name[i] := sq-name[i] + dx;      ;x values in array
    when even[i] sq-name[i] := sq-name[i] + dy;      ;y values in array
```

The next level of implementation optimization is to examine the encoding of the data structure to determine if it is efficient for the operations performed on it. For example, above Translation is better expressed in Cartesian coordinates, while Rotation is better expressed in polar coordinates. The programmer needs to know what the usage of the data structure will be. If, for instance, 95% of the operations on the square will be rotations, then the canonical square is best formulated in polar coordinates.

Another example from above is the choice of the Perpendicular algorithm. The encoding can encourage calculation of line slopes by providing them in ready form, or it can encourage comparison of points by providing the points directly.

AN OPTIMIZED EXAMPLE

The attached three pages contain a description of the development of a minimal data structure for a cube (the principles apply to a "cube" of any dimension, including a 2D square). The mathematical model is *unit vectors*. All properties are defined as operations on unit vectors, using only multiplication in the ternary domain `{0,_,1}`. Special multiplication operators are defined for this purpose. The ADS keeps a tight association with the visual properties of a cube.