The Polar Game in Dylan: A Case Study in Object-oriented Design

Paul R. Potts

June 2013

This article is adapted and revised from parts of several of blog posts that I originally published in my now-defunct blog "Praise, Curse, and Recurse." In those posts, I discussed learning Objective-C, take a dive into Objective-C's origins, and attempt to implement the logic from an old, simple Macintosh game called Polar in three languages: Objective-C, Dylan, and Haskell. In this article I describe the Dylan implementation. The original posts will eventually be part of my archive.

So, this is day 5 in my "undisclosed location" and I haven't gotten much done — I'm still engaged in a job search, and spent about eight hours working on a "take-home test" for an employer (with a few interruptions), I'm pursuing more leads, and I'm trying to socialize with my hosts occasionally, so there are some distractions. But I've got enough information to go on to start implementing something. What I want to implement is a small game. Many years ago there existed on old-school MacOS a small game called "Polar." It was a very simple game, written by a guy (Go Endo) who was probably a student at the time, but I was fond of it — fond enough to save it for 23 years, with the intention of studying its design and re-implementing it in the future. (I'm a bit of a digital pack-rat, and have saved a lot of old bits and bobs like this.) Years ago, I made notes of how to beat the first 3 levels (it was one of those "incredibly simple but maddeningly difficult" games), drew out the levels, made notes on how the objects behaved, etc. I haven't been able to run that game for a long time, but today I just got it working under the MacOS emulator SheepShaver. Here's what level 1 looks like (blown up a bit):

The penguin is your avatar. The rest of the objects are ice blocks, trees, hearts, bombs, mountains, and houses. The world is a sheet of ice. You can walk around on the ice. Some objects (trees, mountains, and houses) can't be moved, while bombs, hearts, and ice blocks move without friction — if you push them, they will keep going until they hit the edge of the world or another object. Trees don't block the movements of the penguin avatar — it can walk over them as if they were just painted on the ground — but trees will block the movement of other objects. If you *slide* an ice block, and, while sliding, it hits another object,

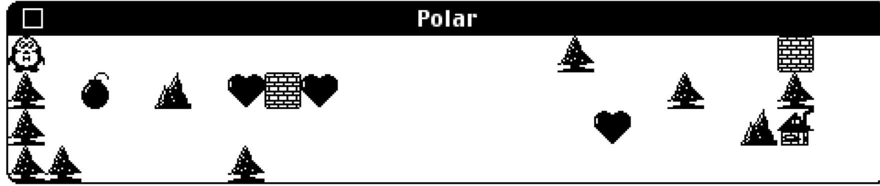

Figure 1: “Polar Level 1”

it will stop sliding. If you *push* an ice block directly against another object, the ice block is crushed, and disappears. If you slide a bomb towards a mountain, it will blow up the mountain — both the bomb and the mountain will disappear. Any other object will stop a sliding bomb and it won’t blow up.

The goal of the game is to slide all the hearts on the board into the house. Cute, huh? But because the ice is frictionless to everything except the penguin avatar, it’s incredibly easy to get objects stuck against walls or corners where you can no longer move them the way you need to, because there are no longer open spaces available for you to get the penguin avatar *into*, and to slide the objects *onto* (remember, you can’t *pull* anything, only *push* things by walking into them). So, you have to carefully plan out your moves. If you get stuck, there’s an option to start the level over. You might wind up having to use that option a lot.

I should mention that the original game had a copyright notice (1990), and was shareware ($2.00). I can’t remember if I ever sent the author $2.00. I’m not sure how he would feel about me taking apart and trying to re-implement his game, or whether he’d try to assert that copyright prevented me from doing so, but I’ll assume he’s a nice guy and wouldn’t care as long as I don’t charge for it, and go ahead, on the theory that easier to ask forgiveness than permission. I was not able to find him online — maybe “Go Endo” was a pseudonym?

Anyway, let’s try to reverse-engineer the way the original game boards are stored. They are in MacOS resources of type ‘STGE’ (stage). Using ResEdit, I was able to see the raw data for ‘STGE’ resource ID -16000:

```
0x0000 0x0000 0x0003 0x0001
0x0000 0x0000 0x0000 0x0000
0x0000 0x0000 0x0000 0x0000
0x0000 0x0000 0x0000 0x0000
0x0000 0x0000 0x0001 0x0000
0x0000 0x0000 0x0000 0x0000
0x0004 0x0000 0x0000 0x0001
0x0000 0x0006 0x0000 0x0002
0x0000 0x0005 0x0004 0x0005
0x0000 0x0000 0x0000 0x0000
0x0000 0x0000 0x0000 0x0000
```

```
0x0000 0x0001 0x0000 0x0000
0x0001 0x0000 0x0000 0x0001
0x0000 0x0000 0x0000 0x0000
0x0000 0x0000 0x0000 0x0000
0x0000 0x0000 0x0000 0x0000
0x0000 0x0000 0x0000 0x0005
0x0000 0x0000 0x0000 0x0002
0x0003 0x0000 0x0000 0x0001
0x0001 0x0000 0x0000 0x0000
0x0000 0x0001 0x0000 0x0000
0x0000 0x0000 0x0000 0x0000
0x0000 0x0000 0x0000 0x0000
0x0000 0x0000 0x0000 0x0000
0x0000 0x0000 0x0000
```

There are 99 16-bit values. My first guess is that this corresponds to the 24x4 grid (96 board positions) plus 3 extras for some kind of of header or footer data (maybe the total number of hearts is indicated, for example). There are 7 unique values, so it seems likely that these values correspond, somehow, to our eight different tile types, with zero representing a blank space. But the counts of each type don't *quite* match up. The first board has 8 trees, 1 bomb, 2 hearts, 2 ice blocks, 2 mountains, 3 hearts, 1 house, and 1 penguin (there is always 1 penguin), while this 'STGE' resource has: 9 ones, 2 twos, 2 threes, 2 fours, 3 fives, and 1 six. The counts are very close, though, so this just *has* to represent level 1. The 5 almost certainly represents a heart, but I'm not clearly seeing the layout. The first vertical column goes penguin, tree, tree, tree. I don't quite see a pattern that looks like that, but 'STGE' resources -1599 and -15996 give me a hint that the extra data is *before* the game board data: those boards contain 0x0007 and 0x0008 as their third values. Those don't appear anywhere else, so they probably don't indicate tiles. So let's try rearranging resource -16000 without the first 6 bytes, remove redundant zeroes for clarity, and look at the values aligned by groups of 24 instead of 4:

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 4 0 0
1 0 6 0 2 0 5 4 5 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 2 3 0 0
1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Ah, now we can clearly see the board. The left column is actually all trees — when the board first appears, the penguin is hiding a tree. So, there are actually nine trees in the resource. The encoding looks like this: empty space = 0, tree = 1, mountain = 2, home = 3, ice block = 4, heart = 5, and bomb = 6. The penguin isn't represented in this board data, but his starting position is probably represented by the first two values in the resource, **0x0000 0x0000**, most likely encoded as row index, column index to correspond to row-major indexing. The next value, **0x0003**, indicates that there are 3 hearts to count down as the game is solved (although this can be gleaned from the board data, the developer

probably found it simpler to specify that number separately).

Back in 1991 I came up with a C++ class design (actually, it doesn't quite look like C++; I think it was written using THINK C's object-oriented extensions, which are sort of lost in the mists of time to me — what is that **indirect** keyword? What did **#pragma options(virtual)** do? I don't remember for sure). But after revisiting that design, and coming up with a design in Objective-C, I turned my attention to how that design could look in Dylan. So I'll pick up the excerpts from my blog there.

I've been a little stymied as to how to express the design best in Objective-C. If the game was complicated to play, I wouldn't feel bad about having a program that looked complex. But it's really an elegantly simple game, and so I feel like the implementation should reflect that. My Objective-C implementation has been feeling more and more bloated and pointlessly complex, although it works, so my thought was to get the design back down to one that takes full advantage of Dylan's object-oriented programming features, largely borrowed from CLOS, and then try to port that back to Objective-C, adding whatever is needed to fake the dispatch features that Dylan gives me that Objective-C doesn't have.

Several days pass…

With some assistance from the folks on the Dylan Hackers mailing list, I got enough clues to press on and get my Dylan implementation of the Polar game working, at least up through the end of the first board. I haven't verified that every possible tile interaction works yet, but it's a start. This interests me as an object-oriented design problem. My design requires "double dispatch," based on the types of two different interacting objects.

If I wrote it in C, the tile classes pretty much wouldn't exist; they'd exist only as flag enumerations in an array of tiles, and the code would consist mostly of **switch** or **if-else** logic that did the "double dispatch" in an explicit order, without relying on the compiler very much. Objective-C, again mostly C with a thin layer for representing classes, doesn't really offer features that make these tile classes worthwhile, so I will likely just keep the model of the game board — the *model* in the model/view/controller — and treat the tiles like I would in plain old C. But in Dylan they have an interesting life in terms of how they can be used to organize the code — using **generic functions** — so that I can write less "code to find code." I can avoid explicitly writing code that just looks at the run-time identity of objects (sometimes called *introspection*), and chooses what lines of code should implement the interaction between objects. In other words, I can focus on the object-oriented design.

Here are the tile classes in Dylan:

```
define abstract class <tile> ( <object> ) end;
define abstract class <blocking> ( <tile> ) end;
define abstract class <walkable> ( <tile> ) end;
define abstract class <movable> ( <blocking> ) end;
define abstract class <fixed> ( <blocking> ) end;
define class <bomb> ( <movable> ) end;
define class <heart> ( <movable> ) end;
define class <ice-block> ( <movable> ) end;
define class <house> ( <fixed> ) end;
define class <mountain> ( <fixed> ) end;
define class <edge> ( <fixed> ) end;
define class <tree> ( <blocking>, <walkable> ) end;
define class <empty> ( <walkable> ) end;
```

These tile classes have no state — in Dylan, no *slots* — and are used in my program solely for their types. **<edge>** does not actually appear on the board, but is used internally when the penguin or another moving object attempts to interact with the edge of the board. We treat this just like another blocking object, as if the board was surrounded by immovable, inert objects.

Diagramatically, the classes look like so: (sorry for the image artifacts; I had difficulty finding my original OmniGraffle files from 2013, so I had to improvise):

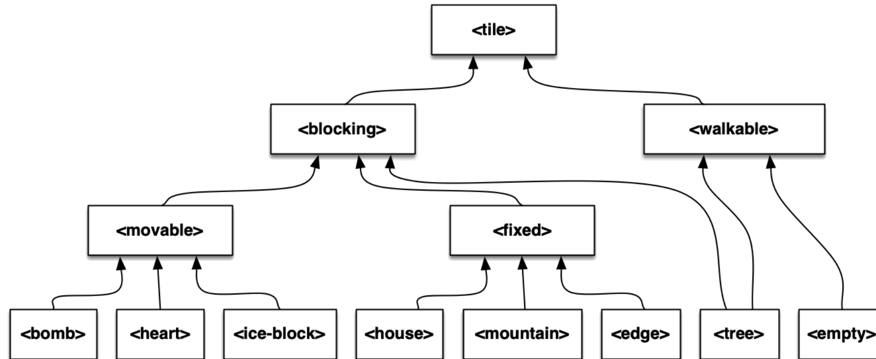

Figure 2: “Polar Classes V2”

There did not seem to be one absolute best way to represent these classes. I want to organize their abstract base classes by behavior, but their behavior does not break down with complete consistency — for example, tiles with trees are “blocking” with respect to sliding objects, except for the penguin. The ice block is “blocking” except for the case where the penguin pushes it and it is not adjacent to an empty tile — then it is crushed. Bombs and hearts seem to have the same interactions with mountains and houses whether they traverse an empty tile by sliding first across one or more empty tiles, while ice blocks

behave differently — if they slide first and then collide with a blocking object, they are not destroyed, they just stop. So the groupings of the concrete classes isn't going to be able to coherently divide up all their possible behaviors.

The scheme I settled on for object interactions involves three layers, in the form of three generic functions. The first, **pushTile**, represents interactions of the player's "avatar," the penguin, with tiles. Then, I can write methods that specialize on the three abstract subclasses of **<tile>** that, together, cover all the concrete subclasses.

```
define generic pushTile( model :: <model>, dir :: <dir>,
    target-pos :: <pos-or-false>, target-tile :: <tile> );

define method pushTile( model :: <model>, dir :: <dir>,
    target-pos :: <pos>, target-tile :: <walkable> )
    => ( result :: <boolean> )
    model.penguin-pos := target-pos;
    #t;
end;

define method pushTile( model :: <model>, dir :: <dir>,
    target-pos :: <pos>, target-tile :: <movable> )
    => ( result :: <boolean> )
    let next-pos :: <pos-or-false> =
        getAdjacentPos( target-pos, dir );
    let next-tile = getTileAtPos ( model, next-pos );
    collide( model, dir, target-pos, target-tile,
        next-pos, next-tile );
    #f;
end;

define method pushTile( model :: <model>, dir :: <dir>,
    target-pos :: <pos-or-false>, target-tile :: <fixed> )
    => ( result :: <boolean> )
    #f;
end;
```

Dylan doesn't strictly require that I define the generic function before defining methods for it; if I just start writing methods with the same name, it will assume that I mean them to be associated with a generic function. But defining the generic function first has a benefit — the compiler will tell me whether my methods make sense, in that their parameters are all strictly the same type or a more specific subclass of the types mentioned in the **define generic** statement. Note that **<pos-or-false>** is a type union of a simple **<pos>** class with **singleton(#f)**. The generic uses that type union, but the methods are more specific: they require an actual **<pos>** instance and will not accept **#f**.

The first method handles the case where the penguin is pushing a **<walkable>**

tile, and returns false to indicate that the penguin position can be updated. The pos must not be **#f**. The second method handles pushing any **<movable>** tiles. And the third handles the **<fixed>** tiles. Between the three methods, you might notice that they cover all the leaf classes (all the instantiable classes) in the graph above, in 3 separate groups with no overlapping. You could shade in the leaf nodes covered by the three different methods with three different colors, going from the abstract classes mentioned downward, and all the leaves would all be colored and none would be colored more than once:

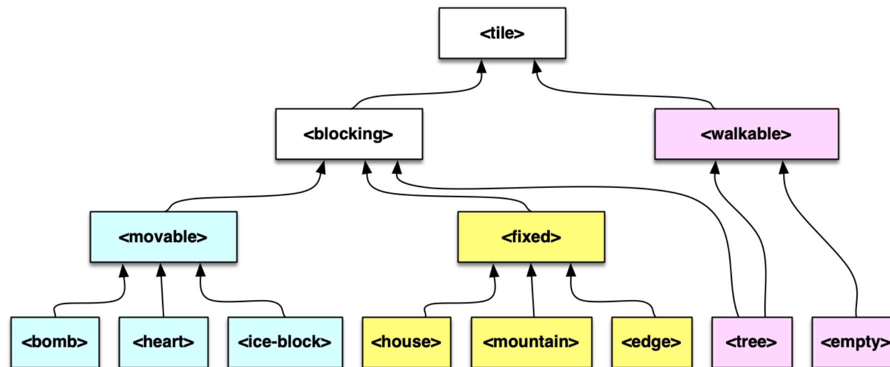

Figure 3: "Polar Dispatch for pushTile"

So on the tile parameter, the coverage of the concrete classes is complete and the dispatch algorithm should not have any difficulty. Combined with the position parameter, though, the situation is slightly trickier. At runtime, a caller could call **pushTile** with **#f** for **pos** and **<empty>** or **<bomb>** for **tile** and the dispatcher would, correctly, throw up its hands at this point and say that there was no applicable method. I could have defined a more general method to handle this case, but I didn't — there shouldn't ever be an empty or bomb tile without a corresponding valid position, since they are real tiles on the board, and I want the runtime to help me catch that case if it ever happens. Similarly, I could have defined a method that handled **<blocking>** or **<file>** as part of this generic function, but the whole point is that I don't know what to do with those more general classes here.

So, you may notice that the middle **pushTile** method calls **collide** with a second tile and position, adjacent to the first in a specified direction. That generic function **collide** and its methods look like this:

```
define generic collide( model :: <model>, dir :: <dir>,
    tile-1-pos :: <pos>, tile-1 :: <movable>,
    tile-2-pos :: <pos-or-false>, tile-2 :: <blocking-or-empty> );

define method collide( model :: <model>, dir :: <dir>,
    movable-pos :: <pos>, movable-tile :: <movable>,
```

```
        next-pos :: <pos>, next-tile :: <empty> )
        slide ( model, dir, movable-pos, movable-tile,
                next-pos, next-tile );
    end;

    define method collide( model :: <model>, dir :: <dir>,
        ice-block-pos :: <pos>, ice-block-tile :: <ice-block>,
        icebreaking-pos :: <pos-or-false>,
        ice-breaking-tile :: <blocking> )
        setTileAtPos( model, ice-block-pos, $the-empty );
    end;

    define method collide( model :: <model>, dir :: <dir>,
        heart-pos :: <pos>, heart-tile :: <heart>,
        house-pos :: <pos>, house-tile :: <house> )
        setTileAtPos( model, heart-pos, $the-empty );
        decrementHeartCount( model );
    end;

    define method collide( model :: <model>, dir :: <dir>,
        bomb-pos :: <pos>, bomb-tile :: <bomb>,
        mountain-pos :: <pos>, mountain-tile :: <mountain> )
        setTileAtPos( model, bomb-pos, $the-empty );
        setTileAtPos( model, mountain-pos, $the-empty );
    end;

    define method collide( model :: <model>, dir :: <dir>,
        movable-pos :: <pos>, movable-tile :: <movable>,
        blocking-pos :: <pos-or-false>, blocking-tile :: <blocking> )
    end;
```

You might notice that before long you hit yet another method call you haven't seen before — **slide**. This is, as you might guess, yet another generic function. (Doesn't this program every get around to *doing* anything?) In fact it does, but this is the often-paradoxical-seeming logic of object-oriented design — individual methods that seem too small and simple to get anything done can actually get a lot done together, especially when aided by a smart dispatcher that eliminates most of the need to write "code to find code."

The type-union **<blocking-or-empty>** allows us to specify, for our generic function, as tight a class as possible out of two otherwise disjoint sections of our class diagram. We don't have to loosen the type specification needlessly by using , which would allow as a valid class for this parameter. Meanwhile, we can loosen **tile-2-pos** so that we make our intention to allow **#f** explicit here.

The **collide** methods break down as follows. The first one handles any movable tile that is moving onto an empty tile, by calling a **slide** method to be defined

later. The second one is a special case to handle the crushable **<ice-block>** class — if it is pushed into the world edge, or any other object, it is destroyed (replaced with **$the-empty** singleton class instance). The third and fourth methods handle specific interactions between hearts and houses, which gets us closer to completing the current game board, and bombs and mountains, which destroys the mountain (which is sometimes necessary, but sometimes can lead to a situation where the board can no longer be completed). And finally, to handle the cases where the penguin pushes a heart against a mountain, or a bomb against the edge of the world, we have a less specific method that dispatches on **<movable>** and **<blocking>**. This prevents the runtime from generating an error, but also gives us a place where we could, in the future, add code to generate some kind of feedback to the user, like a special sound to indicate that this move didn't accomplish anything.

The breakdown of instantiable tile classes here is much more complex, especially given that we are dispatching on **two** class parameters drawn from the same hierarchy. We could try coloring them using two copies of the diagram, one highlighting the first class parameter we dispatch on, and one highlighting the second:

Err, that's pretty, but is it helpful? With the blocks that are filled with a color gradient, I'm trying to show a specially-handled interaction, in this case a specific method of the **collide** generic function. When a **<movable>** (pale blue) **<bomb>** (blue and green) collides with a **<fixed>** (yellow) **<mountain>** (yellow and green), the green color indicates that their interaction is handled specially. There's a similar specific method for a collision between **<heart>** and **<house>**, indicated by the gray color. This color scheme is also supposed to illustrate that **<ice-block>** and **<blocking>** have a special interaction. Unlike with the first generic function, in this one there is significant overlap between the classes handled by the different methods, due to the way we dispatch on *two* classes.

This is where the Dylan dispatch mechanism really has to shine, and work the way I want it to. There is an ordering that makes sense from my point of view, and that is one in which the *most specific* matching method will be called. However, as you can see, quantifying "most specific" may be slightly complex when dispatching on more than one class parameter, especially when throwing in type-unions. Fortunately this code is now working, but while I was developing it I became familiar with a warning message in Open Dylan that says something like "the method dispatch handling this set of classes is determined by arbitrary and capricious rules" — indicating that the dispatch logic is still considered a work in progress. I was concerned that the current version of the Open Dylan compiler wasn't quite solid enough to make this design work, but, in fact, it did just fine. My backup plan was to try to dispatch entirely on type-unions made up of different sets of singletons, but that would have resulted in longer code that obscured the meaning of the abstract classes.

I won't go to the trouble to make a similar diagram of dispatch for the **slide**

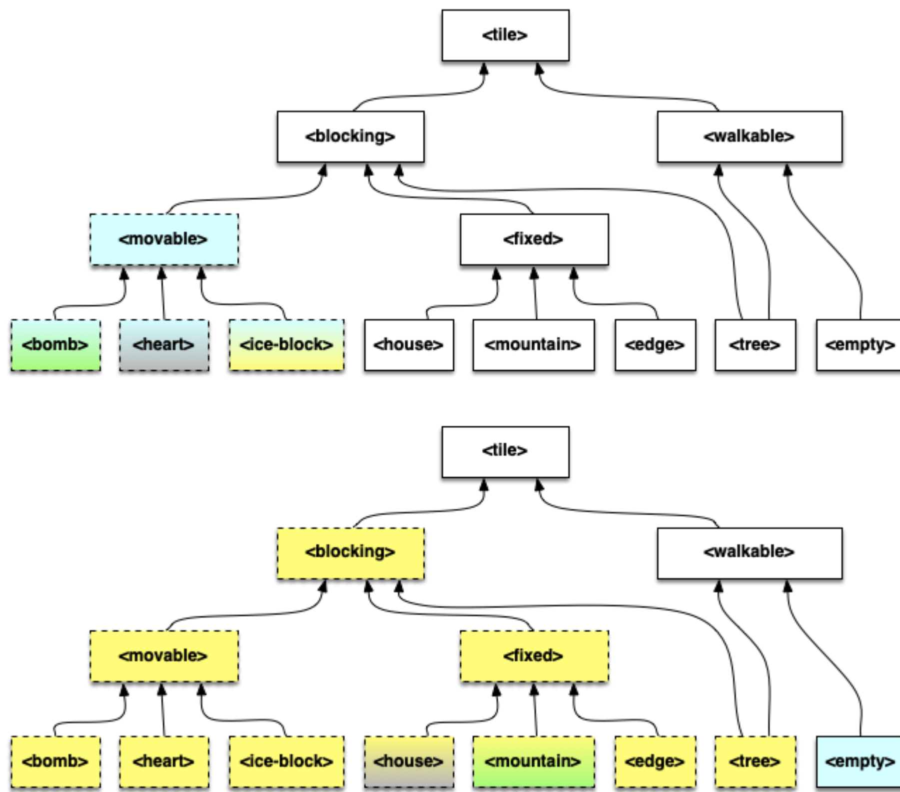

Figure 4: “Polar Dispatch for collide”

method, as it is relatively simple, but the **slide** code looks like this, and I hope by now you might be able to follow the way these methods work:

```
define generic slide( model :: <model>, dir :: <dir>,
    movable-pos :: <pos>, movable-tile :: <movable>,
    next-pos :: <pos-or-false>,
    next-tile :: <blocking-or-empty> );

define method slide( model :: <model>, dir :: <dir>,
    movable-pos :: <pos>, movable-tile :: <movable>,
    next-pos :: <pos>, next-tile :: <movable> )
    let next-next-pos ::  =
        getAdjacentPos( next-pos, dir );
    let next-next-tile = getTileAtPos( model, next-next-pos );
    setTileAtPos( model, next-pos, movable-tile );
    setTileAtPos( model, movable-pos, $the-empty );
    slide( model, dir, next-pos, movable-tile ),
           next-next-pos, next-next-tile );
end;

define method slide( model :: <model>, dir :: <dir>,
    movable-pos :: <pos>, movable-tile :: <movable>,
    next-pos :: <pos-or-false>, next-tile :: <blocking> )
    collide( model, dir, movable-pos, movable-tile,
             next-pos, next-tile );
end;

define method slide( model :: <model>, dir :: <dir>,
    ice-block-pos :: <pos>, ice-block-tile :: <ice-block>,
    next-pos :: <pos-or-false>, next-tile :: <blocking> )
end;
```

Aaaand that's pretty much the whole of the logic for handling interaction between the penguin and the various tiles.

Note that in the first **slide** method, the code calls **slide** recursively. It looks kind of like we have no termination condition! Except note that the method isn't really calling itself using the same tile objects that it was called with. There's a slightly tricky part where we want to bind up the next tile beyond the two tiles we were dispatched on, then perform two **set** operations to move the currently sliding tile, then dispatch on the starting tile at its moved position. To figure all that out, I had to draw some bits of the game board with circles and arrows (but not a paragraph on the back of each one to be used as evidence against me). (If you don't get that reference, either you're too young or I'm too old!) When we come to a termination condition for our slide, we'll actually call a different method of the same generic function — most likely the third one, where a sliding object encounters a blocking object. That condition can include hitting the edge

of the board. And fortunately, we already have logic for that in our **collide** generic function! So sliding hearts and bombs are handled just the same as if they were pushed instead of reaching the end of a slide movement.

This is not the whole program, obviously, but these are the key methods for encoding the collisions between tiles. If you'd like to play with the whole toy program, which is still lacking a user interface, you can find it on GitHub. I hope it might illustrate why I still love Dylan, because of the way it provides a different paradigm for object-oriented programming, allowing me to spend more time thinking about how I want my objects to interact, and less time writing tedious boilerplate and "code to find code." Happy programming!

As always, this content is available for your use under a Creative Commons Attribution-NonCommercial 4.0 International License.