Marine Biology Simulation Case Study

Chapter 4

Specialized Fish

After I demonstrated the dynamic population version of the simulation to the marine biologists to make sure it met their needs, we talked about what modification should be made next. They wanted me to create several new kinds of fish with specialized patterns of movement to see what effect that would have on the simulation.

Problem Specification

The marine biologists decided that they would like to start by adding two new kinds of fish to the simulation: fish that dart forward whenever possible and slow-moving fish. These new types of fish would share all the attributes already defined for the `Fish` class except that their movement behavior would be different. In particular, the biologists decided that:

- A darter fish darts two cells forward if both the first and second cells in front of it are empty. If the first cell is empty but the second cell is not, then the darter fish moves forward only one space. If the first cell is not empty, then the darter reverses its direction but does not change its location. Like objects of the `Fish` class, darters never move in the same timestep as breeding.
- A slow fish moves so slowly that, even when it does not breed, it only has a 1 in 5 chance of moving out of its current cell into an adjacent cell in any given timestep in the simulation. Like objects of the `Fish` class, slow fish never move in the same timestep as breeding and never move backward.

Design Issues

Although normal fish, darters, and slow fish exhibit different behavior when moving, they also share many similarities. For example, the `id`, `color`, `location`, and `direction` accessor methods have nothing to do with the particular type of fish being modeled. At the abstract level of the `act` method, deciding when to breed, when to move, and when to die, the different kinds of fish have the same behavior. The particulars of how they breed and die are almost identical; the only difference is what type of fish they generate when breeding. The primary difference among these three kinds of fish, though, is how they move.

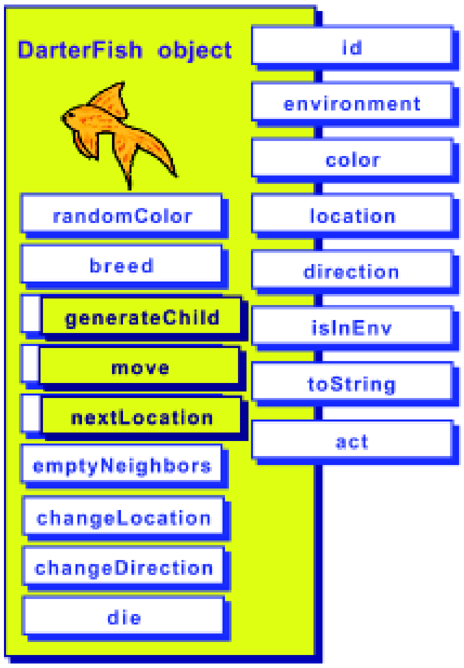
Rather than repeat the code for all the methods with the same behavior, I wanted to use *inheritance* to implement the specialized kinds of fish. My idea was to create new `DarterFish` and `SlowFish` subclasses that would *extend* the `Fish` class. This means that an object of the `DarterFish` (or `SlowFish`) class would inherit certain data and behavior defined in the `Fish` superclass, such as the accessor methods and the `act`, `breed`, and `die` methods. It could also *redefine* the behavior of some superclass methods by providing new implementations in the subclass. (This is also known as *overriding* the superclass method.) For example, the `DarterFish` class could redefine the `generateChild` method to create darter fish rather than normal fish, and redefine the `move` method (or its helper methods) to move in a different way.

One question I had was, if a darter or slow fish used the `act` method inherited from the `Fish` class, how would it know to use the redefined `generateChild` and `move` methods from the `DarterFish` or `SlowFish` class, instead of the methods from the `Fish` class? I decided to ask Jamie. The answer was *dynamic binding* (sometimes called *polymorphism*, according to Jamie). Here's how it works. The `Simulation` object asks a particular fish (darter, slow, or normal) to act. Let's say, for the sake of simplicity, that the object is a darter fish.

- The `DarterFish` class doesn't have an `act` method defined in it, so it inherits the generic `act` method from `Fish`. (We say that the call to `act` is *dynamically bound* to the `act` method in the `Fish` class.)
- The object that is acting, though, is still a `DarterFish` object. When the `act` method (inherited from `Fish`) calls `move`, it is actually the darter fish executing the `act` method that is invoking the `move` method on itself. The `DarterFish` class does have a `move` method defined in it, so that is the one that's executed. (The call to `move` is dynamically bound to the redefined `move` method in the `DarterFish` class.)
- Depending on how it is implemented, the redefined `move` method could call another internal method, like `location` or `emptyNeighbors`. We'll see that in `DarterFish` the redefined `move` method calls `nextLocation` (which it redefines) and `location`, `direction`, `changeLocation`, and `changeDirection` (which it does not redefine). This means that the call to `nextLocation` will be dynamically bound to the redefined method in the `DarterFish` class, but the calls to `location`, `direction`, `changeLocation`, and `changeDirection` will be dynamically bound to the inherited methods from the `Fish` class.

The picture below shows an object of the DarterFish class, with the redefined methods overlaying the methods they override. (Protected methods, which are meant to be used internally, are shown inside the box representing the DarterFish object.)

DarterFish



One of the assumptions in this explanation of dynamic binding is that the darter or slow fish has access privileges to the inherited methods (like `act` and `location`) and that inherited methods (like `act`) have access to redefined methods (like `move`). This would not be the case if these methods were private. For example, if the `move` method were private in `Fish` and `DarterFish`, then the `act` method in `Fish` would always use the `move` method from the `Fish` class, even for a darter fish, because that is the only `move` method to which it would have access. Similarly, if the `changeLocation` method were private in `Fish`, a redefined `move` method in a subclass would not be allowed to invoke it. The `protected` keyword allows inherited methods in superclasses to call methods that dynamically bind to methods in subclasses, and allows methods in subclasses to call inherited methods in superclasses. The `public` keyword would allow this also, but the `protected` keyword is an indication that the access is not meant to be open to all classes. Unfortunately, Java does not guarantee that objects of other classes, like the `Simulation` class, do not make use of `protected` methods. This meant that I needed to be very careful to check for myself that I used `protected` methods only in subclasses, as intended, and that I did not use them in client code.

Once I understood dynamic binding and how I should use protected methods in the marine biology simulation program, I felt comfortable implementing `DarterFish` and `SlowFish` as subclasses of the `Fish` class.

Darter Fish

Implementation of the `DarterFish` Class

The first thing I needed to do was create the empty `DarterFish` subclass, specifying that it extends the `Fish` class.

```
public class DarterFish extends Fish
{
}
```

A subclass inherits its superclass's data, can inherit or redefine its methods, and can define new data and methods. In the case of `DarterFish`, I knew I wanted to inherit most of the `Fish` methods but redefine the `move` method. According to the specification, a darter can only move forward. It moves two spaces forward if it can, and one space forward if it can't move two spaces. If it can't move at all, because the cell in front of it is not empty, then it reverses its direction without moving.

I decided that my first step would be to modify the `nextLocation` method, which defines how the fish chooses where to move. My redefined method finds the location in front of the darter (in other words, the neighbor of the current location in the same direction that the fish is facing) and the location in front of that (the neighbor of the one in front, in the same direction). The new `nextLocation` method then checks whether those spaces are empty. If neither location is empty, `nextLocation` returns the darter's current location because it was unable to move. (My logic for deciding when a darter is unable to move was incorrect, though, as I discovered later.)

The code below shows the first draft of my redefined `nextLocation` method, without debugging messages.

```
protected Location nextLocation()      // first draft!
                                       // (warning: buggy!)
{
    Environment env = environment();
    Location oneInFront = env.getNeighbor(location(), direction());
    Location twoInFront = env.getNeighbor(oneInFront, direction());
    if ( env.isEmpty(twoInFront) )
        return twoInFront;
    else if ( env.isEmpty(oneInFront) )
        return oneInFront;
    else
        return location();      // can't move, stay in
                                // current location
}
```

I also needed to write a new `move` method for `DarterFish`, so that if the darter did not change location, then it reversed its direction. Here is the new `move` method without debugging messages.

```
protected void move()
{
    // Find a location to move to.
    Location nextLoc = nextLocation();

    // If the next location is different, move there.
    if ( ! nextLoc.equals(location()) )
    {
        changeLocation(nextLoc);
    }
    else
    {
        // Otherwise, reverse direction.
        changeDirection(direction().reverse());
    }
}
```

The `move` method for `DarterFish` was simple to write because the logical structure is the same as the `move` method for `Fish`. The code is simpler, though, because when a darter moves, its direction does not change.

I also needed to write one or more constructors for the `DarterFish` class, because constructors are not inherited like other methods. Each class must explicitly define its own constructors. The original `Fish` class has three constructors: one that specifies the environment and initial location, one that also specifies the initial direction, and a third that specifies environment, location, direction, and a color. For testing purposes I decided to make all darter fish yellow. This was fine with the biologists; they had primarily been using color to make the simulation more interesting, not to represent meaningful information. At first I thought that because all darters would be the same color, I didn't need to provide the third constructor. Then I decided to go ahead and provide it anyway, in case the biologists wanted to define darters with other colors later.

Below is the code for the first constructor. The only thing it has to do is to call the appropriate superclass constructor (using the `super` keyword) to initialize the attributes inherited from `Fish`, such as the location, direction, and color. The expression to get a random direction is the same as that found in the two-parameter `Fish` constructor. The code for the other constructors is practically the same, so it is not shown here.

```
public DarterFish(Environment env, Location loc)
{
    // Construct and initialize the attributes inherited from Fish.
    super(env, loc, env.randomDirection(), Color.yellow);
}
```

Finally, I redefined the `generateChild` method to construct a new `DarterFish` rather than a new `Fish` object, as shown below (without the debugging message). Everything else about the method is the same. With the redefined `generateChild` method, I did not need to copy or redefine the rest of fish breeding behavior but could inherit it from the `Fish` class.

```
protected void generateChild(Location loc)
{
    // Create new fish, which adds itself to the environment.
    DarterFish child = new DarterFish(environment(), loc,
                           environment().randomDirection(), color());
}
```

Analysis Question Set 1:

1. What is the logic error in Pat's first draft of the `nextLocation` method?
2. A darter can only swim east and west or north and south. How might we change the darter so that it usually continues east and west, or north and south, but occasionally switches?

Testing the `DarterFish` Class

The new code for the `DarterFish` class seemed pretty simple, so I thought I would test it by just running it a number of times and seeing if the behavior seemed right. It did, and I was about to move on to the `SlowFish` class when I noticed something odd. One of the darters had hopped right over another fish! I ran the program a few more times and saw the same behavior again.

I turned on debugging at the beginning of the `step` method in the `Simulation` class (and restored it before returning) to help trace fish activity throughout the simulation. To help clarify when and where darters were moving and when they were blocked, I added some debugging statements to the `nextLocation` method in `DarterFish`. I ran the program and analyzed my results more carefully and realized that there were two different situations that could cause these results, one of which was an error. One situation was that the darter could have moved forward through an empty cell to get to the second cell, and then another fish could have slipped from the side into the first empty cell. This would be acceptable behavior, according to the program specification.

The second situation was that the darter could have hopped over a fish to get to an empty cell beyond it, because I had written `nextLocation` incorrectly. The code I had written did not check that **both** the cell immediately in front of the darter and the cell beyond that were empty. The corrected code appears below.

```
protected Location nextLocation()     // corrected code!
{
    Environment env = environment();
    Location oneInFront = env.getNeighbor(location(), direction());
    Location twoInFront = env.getNeighbor(oneInFront, direction());
    if ( env.isEmpty(oneInFront) )
    {
        if ( env.isEmpty(twoInFront) )
            return twoInFront;
        else
            return oneInFront;
    }

    // Only get here if there isn't a valid location to move to.
    return location();
}
```

Because I had almost missed the error in `nextLocation`, I decided to go back to testing more thoroughly, as the original programmer had done. I first developed black box test cases to test the `DarterFish` class, basing them on the test cases for the `Fish` class. Only the new or modified test cases are shown below.

- A file with a single darter fish should run with no errors. (The behavior of the fish will depend on its starting location and direction.)
- A file with two or more darter fish, or with one darter and one non-darter fish, in the same location should generate an error.
- A file with either normal or darter fish in every location in the environment (but only one in every location) should run with no errors. Whether they may move or not depends on whether breeding and dying have been implemented (see the appropriate test case from either Chapter 2 or Chapter 3). Any darters that do not move should reverse direction.
- A darter that does not breed and that has two empty neighboring locations in front of it should always move forward two spaces. A darter that does not breed and that has only one empty neighboring location in front of it should always move into that location. A darter that does not breed and does not change location should reverse its direction. This leads to a visual pattern that is easy to spot — darter fish appear to pace back and forth in the bounded environment.
- All darter fish should be yellow. (This is actually based on an implementation decision, not on the original problem specification.)

I then considered the code in the new `move` and `nextLocation` methods to see if I needed to develop additional test cases. It seemed that the black box cases listed above would cover the new code. I decided to start my testing by rerunning my previous tests with normal fish to verify that the results were the same and that my modifications had not broken working code. This kind of testing is known as *regression testing*.

As I had with my previous test runs, I seeded the random number generator to get predictable results (see Exercise 1 in Exercise Set 5 of Chapter 2). For the regression tests it was important to use the same seed, and there was no reason to change the seed for the new tests. I had already turned on debugging to find the bug in `nextLocation`. Finally, I made a copy of `fish.dat` and changed the six fish to six darter fish. I thought that if I used the same seed and the same initial configuration of fish, then I would see the same behavior for breeding and dying, although the movement would be different. Then I ran the program. This time the darters moved exactly where I expected them to move in each timestep.

To test breeding and dying, which are probabilistic, I needed to keep statistics for a number of timesteps, as I had done for the normal fish. My results are below.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total after 10**	**Total after 20**
Number of darter actions	6	9	11	16	16	19	23	31	42	45	218	733
Number of breed attempts	1	1	2	1	2	2	4	6	7	4	30	114
Number of deaths	1	1	3	2	1	2	3	6	11	9	39	139

After ten timesteps, the percentage of darters in all timesteps that had attempted to breed was 13.8% (30 attempts at breeding in 218 calls to the `act` method); the percentage that had died was 17.9%. This corresponded reasonably well to the 1 in 7 chance of breeding (14.3%) and the 1 in 5 chance of dying (20%) specified in the problem description. I continued the test up to 20 timesteps. This time the percentages were 15.6% for breeding and 19% for dying.

What surprised me, though, was that the numbers recorded for breeding and dying darter fish were different from the earlier tests with normal fish, even though the probabilities remained the same, the initial configuration was the same (except for the name of the class), and I had used the same seed. I realized that the difference was that the original breeding and dying fish use random numbers for movement as well as for breeding and dying, but darters do not. Consequently, the random numbers used for breeding and dying are different for the two populations of fish.

Exercise Set 1:

1. Draw two diagrams illustrating each of the situations Pat discovered in which a darter could hop over, or appear to hop over, another fish.
2. Run the marine biology simulation with the `darter.dat` and the `darterAndNormalFish.dat` initial configuration files to see how the behavior is different. (The difference is more obvious if you use the original `Fish` class rather than the one from the breeding and dying chapter or if the probabilities of breeding and dying are both set to zero. Or you can temporarily comment out the lines of code in the `Fish act` method that deal with breeding and dying.) *[Reminder: In the distributed version of the case study, the class containing the main method is `MBSGUI`. You can edit `MBSGUI.java` and follow the directions in the comments to make darter fish appear different from normal fish or to include darter fish as an option when creating a new environment using the graphical user interface.]*
3. If you're running a user interface that has a "Save" function, run the simulation with the `fish.dat` configuration file and save a copy of the results after 5 timesteps. Use the same seed you used in Chapter 3 (see Exercise 2 in Exercise Set 1). Give the file a descriptive name, like `chap4after5steps.dat`. Run the program for 10 timesteps and save the results in another file with a descriptive name. Compare the files you saved in Chapter 3 with the files you just saved. Are the fish movements the same? Why or why not? Run the program with the `darter.dat` configuration file and save the results in a file for future regression testing. (Be sure to give the file a name that will allow you to identify it later.)
4. If you have added constructors to the `Fish` class in addition to the three original constructors from Chapter 2, analyze which of these constructors should be added to the `DarterFish` class as well. Add them.
5. Redefine the `toString` method in `DarterFish` to clarify that this is a darter. This makes it easier to keep track of darters and normal fish in the debugging output. Turn on debugging in the `step` method in `Simulation`, as you did in Chapter 3, and run the simulation again. This will let you observe the changed behavior at a greater level of detail.
6. Choose a different seed for the random number generator and rerun your tests. What effect does this have on the behavior of the simulation?
7. The darters always move east and west or always move north and south. Create a subclass of the `DarterFish` class, called `TurningDarter`, that behaves like `DarterFish` except that there is a probability of 0.1 that a turning darter turns right or left (each with equal probability) before it tries to move forward. To use the `Random` class, you will need to import `java.util.Random`.

Slow Fish

Implementation of the `SlowFish` Class

The behavior of a slow fish is very similar to the behavior of a normal fish, so again I knew I wanted to inherit most of the `Fish` methods but redefine how (and when) it moved. According to the specification, a slow fish moves so slowly that it only has a 1 in 5 chance of moving out of its current cell into an adjacent cell in any given timestep in the simulation. When it does move, however, it moves just like any other fish of the `Fish` class.

To test whether the fish should move, I knew I would randomly pick a number and compare it to the probability of moving. The first design decision I had to make, though, was whether to put that test in the `move` method or in the `nextLocation` method. If I put it in the `move` method, then 4 out of 5 times it would do nothing. If I put it in the `nextLocation` method, then 4 out of 5 times it would return the current location. Since it seemed like the test could go in either place, I decided to put it in the lower-level, more specific method, `nextLocation`.

I decided to store the probability of moving, the mathematical value 1/5 represented as a `double`, in an instance variable, just as I had with the probabilities of breeding and dying. Then I redefined the `nextLocation` method to pick a `double` randomly in the range of 0.0 to 1.0 (using the `nextDouble` method in `java.util.Random`), and compare it to the instance variable representing the probability of moving. If the randomly chosen number is less than the probability, the slow fish chooses a new location in the usual way, otherwise it returns the current location. To choose a new location, the slow fish calls `super.nextLocation()`, which executes the `nextLocation` method in the `Fish` superclass. (Without the `super` keyword, the call to `nextLocation` in the first `return` statement would be a recursive call to the `nextLocation` method in `SlowFish`. The `super` keyword forces the call to use the inherited `nextLocation` method, which in this case is defined in `Fish`.)

The code below shows the new instance variable and the redefined `nextLocation` method without debugging messages.

```
// Instance Variables: Encapsulated data for EACH slow fish
private double probOfMoving;    // defines likelihood in each
                                // timestep

protected Location nextLocation()
{
    // There's only a small chance that a slow fish will actually
    // move in any given timestep, defined by probOfMoving.
    Random randNumGen = RandNumGenerator.getInstance();
    if ( randNumGen.nextDouble() < probOfMoving )
        return super.nextLocation();
    else
       return location();
}
```

The `SlowFish` class also needed new constructors, not only because constructors aren't inherited like other methods, but also because I needed to initialize the `probOfMoving` instance variable. Each `SlowFish` constructor calls the four-parameter superclass constructor (using the `super` keyword) to initialize the instance variables inherited from `Fish`. To make slow fish easier to spot when testing, I decided to make them red. The code below shows the two-parameter `SlowFish` constructor; the other constructors are similar.

```
public SlowFish(Environment env, Location loc)
{
    // Construct and initialize the attributes inherited from Fish.
    super(env, loc, env.randomDirection(), Color.red);

    // Define the likelihood that a slow fish will move in any given
    // timestep.  For now this is the same value for all slow fish.
    probOfMoving = 1.0/5.0;      // 1 in 5 chance in each timestep
}
```

Finally, I redefined the `generateChild` method to construct a new `SlowFish` rather than a new `Fish`, just as I had for `DarterFish`.

Analysis Question Set 2:

1. What do you think are the advantages or disadvantages of putting the test for whether to move in the `nextLocation` method instead of the `move` method? Would the advantages or disadvantages be different if the problem specification had said that slow fish randomly change direction when they do not move beyond their cell?

2. Pat did not create an accessor method for the `probOfMoving` instance variable. What problem would this cause if you were to create a subclass of `SlowFish` that redefined the `nextLocation` method again? Pat also did not create accessor methods for the `probOfBreeding` and `probOfDying` instance variables in Chapter 3. Under what conditions would this lack of accessor methods cause problems?

3. If you were to create accessor methods for these instance variables, would you make them `public` or `protected`?

Testing the `SlowFish` Class

As usual, I developed the black box test cases for the `SlowFish` class based on the test cases for the `Fish` class. Only the new or modified test cases are shown below.

- A file with a single slow fish should run with no errors. (The behavior of the fish will depend on its starting location.)
- A file with two or more slow fish, or with one slow and one other fish, in the same location should generate an error.
- A file with a fish in every location in the environment (but only one in every location) should run with no errors, regardless of the types of the fish. Whether they may move or not depends on whether breeding and dying have been implemented (see the appropriate test case from either Chapter 2 or Chapter 3).
- A slow fish that does not breed and that has one or more empty neighboring locations in front of it or to its sides should move to one of its neighbors approximately 20% of the time. In other words, in each timestep approximately one fifth of the slow fish that don't breed should move. When a slow fish does move, it should have an equal probability of moving to each of its valid empty neighbors.

I then considered the code in the `SlowFish nextLocation` method to see if I needed to develop additional test cases. It seemed that my final black box case would cover the new code. I would still need to include all the test cases for normal fish when testing slow fish, though, because of the call to `super.nextLocation()`. As usual, I decided to start with regression testing (rerunning my previous tests) to verify that the results were the same and that my modifications had not broken working code.

To help clarify why various fish failed to move, I added a debugging statement to the `nextLocation` method in `SlowFish` to notify me when fish appeared not to move because they were moving too slowly. Finally, I developed a new initial configuration file that contained seven normal fish and seven slow fish. Then I ran the program with the seeded random number generator. My results are below.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**Total after 10**	**Total after 20**
Number of slow fish actions	6	5	5	9	11	17	15	16	15	20	119	523
Number of breed attempts	0	0	1	1	2	1	2	1	4	4	16	76
... that were successful	0	0	1	1	2	0	1	1	4	4	14	64
Number of calls to `move`	6	5	4	8	9	17	14	15	11	16	105	459
... that attempted to move beyond cell	2	0	0	1	0	2	2	1	5	4	17	94
... but were blocked	0	0	0	0	0	0	0	0	1	1	2	22
Number of deaths	1	0	0	1	1	2	2	2	5	3	17	96

After ten timesteps, the percentage that had attempted to breed was 13.4% (16 attempts at breeding in 119 calls to the `act` method). Only 14 of the 16 attempts were successful; two fish did not breed because there were no empty neighboring locations. This led to 105 move attempts. Of these, 16.2% attempted to move out of their current cell (17 out of 105), which was a little lower than I expected, given the 1 in 5 chance of trying to move. The number of fish that actually moved was somewhat lower than the number that attempted to move; twice there were fish that were blocked from moving by other fish in neighboring locations. The percentage of slow fish that died in the first ten timesteps was 14.3%, which was also low.

I continued the test up to 20 timesteps. This time the numbers (shown in the table above) were much closer to what I expected.

Analysis Question Set 3:

1. In Chapter 3, Pat added breeding and dying behavior by modifying the `Fish` class. Another alternative would have been to create a subclass of `Fish` with breeding and dying behavior, leaving `Fish` unchanged. What are the advantages and disadvantages of the two alternatives? What would be the impact on `DarterFish` and `SlowFish`?

2. A method is *deterministic* if, given the inputs to it, you can tell exactly what its result will be. A method is *probabilistic* if, given the inputs, there are various probabilities of different results. Of the `nextLocation` methods in `Fish`, `DarterFish`, and `SlowFish`, which are deterministic? Which are probabilistic?

Exercise Set 2:

1. If you have added constructors to the `Fish` class in addition to the three mentioned above, analyze which of these constructors should be added to the `SlowFish` class as well. Add them.
2. Run the marine biology simulation with the `slowAndNormalFish.dat` and `3species.dat` initial configuration files to see how the behavior has changed. (Again, you may find it easier to see the differences between types of movement without breeding and dying behavior.)
3. Redefine the `toString` method in `SlowFish` to clarify that this is a slow fish. This makes it easier to keep track of slow and normal fish in the debugging output. Turn on debugging in the `step` method in `Simulation`, as you did in Chapter 3, and run the simulation again. This will let you observe the changed behavior at a greater level of detail.
4. Choose a different seed for the random number generator and rerun your tests. What effect does this have on the behavior of the simulation?
5. Using Pat's table of test run results, which may or may not match your own results, calculate the following for 20 timesteps.
   - What percentage of slow fish attempted to breed in each timestep? (We're interested in the average over all 20 timesteps, not in the actual percentage for any one timestep.) Does this percentage correspond with the specified 1 in 7 chance of breeding?
   - Consider the number of times the `act` method was called for slow fish over those 20 timesteps, the number of fish that tried to breed, and the number that bred successfully. Given these values, is the number of calls to the `move` method what you would expect? In other words, is `move` being called the correct number of times?
   - How many times was the `nextLocation` method in `SlowFish` called over the 20 timesteps? (Under what conditions is `nextLocation` called?)
   - How many times was the `nextLocation` method in `Fish` called over the 20 timesteps? (Under what conditions is the `Fish nextLocation` method called?)
   - Of the slow fish that did not breed, what percentage moved too slowly to attempt to leave their cells? What percentage attempted to move beyond their own cell (either successfully or unsuccessfully)? What percentages would you expect, given the problem specification? Do the actual results correspond to the expected results?
   - On average, what percentage died? What percentage would you expect, given the problem specification? Does the actual result correspond to the expected result?
   - Compared to the test results for 10 timesteps, are the actual results over 20 timesteps closer to the expected results as Pat claimed?

6. A slow fish moves in each timestep, even when it doesn't move far enough to leave its current cell. As it moves slowly in its own cell, it may change direction. Modify the `SlowFish` class so that even when it doesn't move outside its cell it may still turn right or left (or continue in its current direction).

7. Implement breeding and dying behavior by creating a subclass of the original `Fish` class. (See Question 1 in Analysis Question Set 3 above.)

8. Define a new `CircleFish` subclass of `Fish` that constantly swims in a circle (as much as is possible in a rectangular grid). In each timestep, the circle fish moves to the location forward and to the right, on a diagonal from its current location, if possible. It also changes its direction by turning 90 degrees to the right. If the fish cannot move as described above, it stays in its current location, but still turns 90 degrees to the right. Make the constructor give all circle fish the same color, so they can easily be distinguished from other fish.

9. Refine the `CircleFish` class to make the movement look more like a circle. In the first timestep, a circle fish moves forward one location (if possible), without turning. During the next timestep, it moves to the location forward and to the right (if possible), as described in Exercise 8. If the fish cannot move, it stays in its current location, but turns 90 degrees to the right. After the fish has moved and turned, or just turned without moving, its next movement will be to move forward one location. The fish continually alternates these moves (except when it is unable to move and only turns).

Quick Reference for Specialized Fish Subclasses

This quick reference lists the constructors and methods associated with the specialized fish classes, `DarterFish` and `SlowFish`, introduced in this chapter. Public methods are in regular type. *Private and protected methods are in italics.* (Complete class documentation for the marine biology simulation classes can be found in the `Documentation` folder.)

DarterFish Class (extends Fish)

```
public DarterFish(Environment env, Location loc)
public DarterFish(Environment env, Location loc, Direction dir)
public DarterFish(Environment env, Location loc, Direction dir, Color col)

protected void generateChild(Location loc)
protected void move()
protected Location nextLocation()
```

SlowFish Class (extends Fish)

```
public SlowFish(Environment env, Location loc)
public SlowFish(Environment env, Location loc, Direction dir)
public SlowFish(Environment env, Location loc, Direction dir, Color col)

protected void generateChild(Location loc)
protected Location nextLocation()
```