



AP[®] Computer Science

**Marine Biology
Teacher's Manual**

Supplement to the Marine Biology Case Study

AP

Advanced Placement Program[®]

This Teacher's Manual is intended for use by AP[®] teachers for course and exam preparation in the classroom; permission for any other use must be sought from the Advanced Placement Program[®]. Teachers may reproduce it, in whole or in part, in limited quantities, for face-to-face teaching purposes *but may not mass distribute the materials, electronically or otherwise*. This Teacher's Manual and any copies made of it may not be resold, and the copyright notices must be retained as they appear here. This permission does not apply to any third-party copyrights contained herein.

The program code for this manual is protected as provided by the GNU public license. A more complete statement is available in the Computer Science section of the AP website.

This booklet was produced by Educational Testing Service (ETS), which develops and administers the examinations of the Advanced Placement Program for the College Board. The College Board and Educational Testing Service (ETS) are dedicated to the principle of equal opportunity, and their programs, services, and employment policies are guided by that principle.

The College Board is a national nonprofit membership association dedicated to preparing, inspiring, and connecting students to college and opportunity. Founded in 1900, the association is composed of more than 3,800 schools, colleges, universities, and other educational organizations. Each year, the College Board serves over three million students and their parents, 22,000 high schools, and 5,000 colleges, through major programs and services in college admission, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT[®], the PSAT/NMSQT[®], the Advanced Placement Program[®] (AP[®]), and Pacesetter[®]. The College Board is committed to the principles of equity and excellence, and that commitment is embodied in all of its programs, services, activities, and concerns.

Copyright © 2000 by College Entrance Examination Board and Educational Testing Service. All rights reserved. College Board, Advanced Placement Program, AP, College Board Online, and the acorn logo are registered trademarks of the College Entrance Examination Board.

**Advanced Placement Program®
Computer Science**

Marine Biology Teacher's Manual

Supplement to the Marine Biology Case Study

*The AP® Program wishes to acknowledge and to thank
Cary Matsuoka of Lynbrook High School in San Jose, California.*

AP Computer Science Marine Biology Teacher's Manual

Contents

Introduction to Case Studies	5
Integration Into Your Course Curriculum	6
Accessing Files	6
Organization of the Teacher's Manual	6
Marine Biology Case Study Part 1	7
Random Walkers	7
Suggestions for integration into course curriculum	7
Teaching suggestions and strategies	7
Answers to exercises	8
Marine Biology Case Study Part 2	14
Experimenting with the program	14
Suggestions for integration into course curriculum	14
Teaching suggestions and strategies	14
Explanations of code	15
Answers to exercises	15
Learning how the program stores fish	19
Suggestions for integration into course curriculum	19
Teaching suggestions and strategies	19
Explanations of code	20
Answers to exercises	21
Learning how fish interact with the environment	21
Teaching suggestions and strategies	21
Answers to exercises	22
Understanding the rest of the program	26
Teaching suggestions and strategies	26
Answers to exercises	27
Appendix A	29
Object Diagrams	30
Appendix B	39
Sample Multiple-Choice Questions	39
Sample Free-Response Questions	44
Appendix C	47
Solutions for Sample Test Questions	47
Sample Multiple-Choice Questions	47
Sample Free-Response Questions	50
College Board Regional Offices	52
1999-2000 AP Computer Science Development Committee	back cover

Introduction to Case Studies

Case studies are a teaching tool used in many disciplines. They give students an opportunity to learn from others with more experience and skill. Just as lawyers, physicians, and therapists learn from the prior experience of others in their fields, so can programmers learn from more experienced programmers. Case studies have been a part of the AP Computer Science curriculum since the 1994-95 academic year. Some good reasons to use a case study in AP Computer Science include:

- Working with a program of significant length
- Thinking through issues of good program design
- Learning from an expert programmer

A lot can be learned from reading programs written by another person. Difficult topics such as program design and construction of algorithms are often best introduced through studying existing programs, including their design, implementation, and validation. For example, think about when you first studied sorting algorithms. You could start from scratch and devise your own sorting algorithms, or you could learn from the work of others who have devised a selection sort or quicksort. This is also true when learning about object-oriented design, a topic recently introduced to the AP Computer Science community. Through the Marine Biology Case Study, the strategies, vocabulary, and techniques of object-oriented design will be emphasized. The case study document, code, and Teacher's Guide should provide some valuable and interesting curricula for your course.

Integration Into Your Course Curriculum

AP Computer Science teachers often face a shortage of time and find themselves squeezing the case study into an already packed timeline. For that reason, the Teacher's Guide has been organized into sections. Within each section, you will find suggestions on how to integrate the case study into topics you already teach during the school year. For example, you might introduce random number generation or apvectors using Part 1 of the case study. Assessment of student understanding can be accomplished using the exercises included in the main case study document.

Accessing Files

The case study files can be downloaded from the AP section of College Board Online® under computer science (www.collegeboard.org/ap/computer-science). Here you will find the case study files and the code to set up some of the study questions.

Organization of the Teacher's Manual

This Teacher's Manual is organized into sections. Each section will typically consist of the following parts:

- Page reference in the case study document
- Suggestions for integration into course curriculum
- Teaching suggestions and strategies
- Explanations of code (as needed)
- Answers to study questions (when the answer includes adding lines to code, the new lines are in boldface)

You are encouraged to integrate the case study into your course throughout the school year. Students will learn and retain concepts more effectively if the case study content is spread out in this way. The Teacher's Manual is not intended to be prescriptive about how and when you teach the case study. Rather, you should feel free to use the suggestions described here as you see fit.

Marine Biology Case Study Part 1

Random Walkers

Text reference: pages 7-18

Suggestions for integration into course curriculum

This section of the case study could be used to teach random number generation. You might coordinate this material with your own presentation and with material in your textbook. The RandGen class will be a very useful class for you to use in the rest of your course. (We recommend that C++ teachers teach “classes” in their “courses.”)

The development of the AquaFish class on pages 13-16 provides a fine example for studying simple classes. After you introduce objects and classes in your AP Computer Science course, the AquaFish class will provide another example to check for student understanding of classes. The exercises will give you a chance to see if students can modify an existing class, an expectation described in the course syllabus for AP Computer Science.

The development of the AquaFish class also provides a context to introduce object diagrams. An object diagram is a graphic organizer that shows classes, their public and private data members, and the interaction between classes in a larger program such as this case study. The object diagrams referred to in this Teacher’s Manual are included in Appendix A. They can also be downloaded from the Computer Science section of the AP website (www.collegeboard.org/ap). The object diagram for the AquaFish class is included on page 30. You are encouraged to use the object diagram for the AquaFish class as you teach this material.

Teaching suggestions and strategies

- Have students do exercises 1 and 2 on page 8 in small groups of 4-5 students. Let them run the simulation using a coin and a piece of paper, or maybe even walking back and forth in the room.
- As students read through the text, they should work with the RandGen class as they use the client program samples and exercises on pages 11-12. The RandGen class files are included in the files for Part I. Please note that students are not required to know or study the implementation code for the RandGen class.
- The two programs used in this section, sixflips.cpp and onedwalk.cpp are available for study and revision from the College Board website.

Answers to exercises

page 8-1 The fish could end up in the starting position, or some even distance away from the starting point, e.g., +2, +4, +6, or -2, -4, -6. Given the even number of flips (6), it is impossible for the fish to end up an odd distance away from the starting position.

page 8-2 Given that a coin flip is a truly random event, the number of heads (move right) should equal the number of tails (move left), thus canceling each other. The fish will most likely end up at the starting position, often 2 positions to the right or left, and very rarely 6 positions to the right or left.

page 9 **Stop and help.** The output of program `sixflips.cpp` looks random, although you should run it a few times to see the “randomness” of the results. Each run will most likely give a different output than the previous run. If it does not, the random number generator is being constructed with a seed that causes the same sequence of numbers to be generated every time you run the program. Remove the integer parameter passed to the `RandGen` constructor to obtain different results with each run.

page 10 **Stop and help** (top of page). If an integer argument is supplied to the `RandGen` constructor, each run will produce the same output.

Stop and help (bottom of page). The final position value will be -2, 0, or +2, although +2 rarely showed up when the author ran the program. These are the only possibilities if the number of steps chosen is two.

page 11-1

```
// diceroll.cpp

#include <iostream.h>
#include "randgen.h"

int main()
{
    int k;
    RandGen r;

    for (k = 0; k < 10; k++)
        cout << r.RandInt(1, 6) << endl;
    return 0;
}
```

page 11-2 The first if statement will evaluate true approximately 33% of the time, so red will appear about 33% of the time.

The else-if statement will be evaluated the other 66% of the time. The probability of matching the value 1 is 1/3. Therefore the chances of getting a “white” value is $2/3 * 1/3$, or 22% of the time.

The last else will happen 2/3 of the time the else-if is evaluated. The chances of getting a “blue” value is $2/3 * 2/3$, or 44% of the time.

page 12-3

```
flip = (int)(r * 2);
```

`r` will be a random number $0 \leq r < 1.0$, so $0 \leq r * 2 < 2.0$. When a double is cast to an int it is truncated, so if $0 \leq r * 2 < 1.0$, then $(int)(r * 2) = 0$, and if $1.0 \leq r * 2 < 2.0$, then $(int)(r * 2) = 1$.

page 12-4

```
// onedwalk.cpp

#include <iostream.h>
#include "randgen.h"

int main()
{
    RandGen randomVals;
    int numSteps, step, flip;
    int position = 0;
    int max = 0;
    int min = 0;

    cout << "Number of steps? ";
    cin >> numSteps;

    for (step = 0; step < numSteps; step++)
    {
        flip = randomVals.RandInt(2);
        if (flip == 0)
            position++;
        else
            position--;

        if (position > max)
            max = position;

        if (position < min)
            min = position;
    }

    cout << "Final position = " << position << endl;
    cout << "Maximum position = " << max << endl;
    cout << "Minimum position = " << min << endl;

    return 0;
}
```

page 18-1

The use of a RandGen object is encapsulated inside an AquaFish object. The #include "randgen.h" call is taken care of at the top of the aquafish.cpp file.

page 18-2

A compiler will compile each file (aquafish.cpp, randgen.cpp, aquamain.cpp) into object code. The linker links all the object codes together into an executable program.

page 18-3

```
AquaFish::AquaFish(int tankSize, int initPosition)
: myPosition(initPosition),
  myTankSize(tankSize),
  myBumpCount(0),
  myDebugging(true)
{
}
```

page 18-4

```
AquaFish::AquaFish(int tankSize, int initPosition, bool debug)
: myPosition(initPosition),
  myTankSize(tankSize),
  myBumpCount(0),
  myDebugging(debug)
{
}
```

page 18-5

When the debugging responsibility is left to the AquaFish class, it makes life less complicated for the client programmer, i.e., you don't have to worry about setting a Boolean value. However, if you wanted to switch the debugging off, the programmer would have to recode the debugging value inside of the AquaFish class and recompile.

Providing another constructor that allows the client programmer to set the Boolean value is very useful. This gives the client programmer a bit more control of the behavior of an AquaFish object. It also eliminates the need to recode the AquaFish class if it is necessary to change the debugging value. If the client programmer does not wish to set the debugging value, it can be omitted and either of the other two constructors will be invoked. There really isn't a downside to this approach.

page 18-6

```
// aquafish.h

#ifndef _AQUAFISH_H
#define _AQUAFISH_H

class AquaFish
{
public:
    //... already existing member functions
    int MaxPos() const;    // Return max position
    int MinPos() const;    // Return min position

private:
    //... original data members
    int myMaxPos;
    int myMinPos;
};

#endif

// aquafish.cpp

#include <iostream.h>
#include "aquafish.h"
#include "randgen.h"

AquaFish::AquaFish(int tankSize)
    : myPosition(tankSize/2),
      myTankSize(tankSize),
      myBumpCount(0),
      myDebugging(true),
      myMaxPos(tankSize/2),
      myMinPos(tankSize/2)
{
}
```

```

void AquaFish::Swim()
{
    RandGen randomVals;
    int flip;

    if (myPosition == myTankSize - 1)
    {
        myPosition--;
    }
    else if (myPosition == 0)
    {
        myPosition++;
    }
    else
    {
        flip = randomVals.RandInt(2);

        if (flip == 0)
        {
            myPosition++;
        }
        else
        {
            myPosition--;
        }
    }

    if (myDebugging)
    {
        cout << "*** Position = " << myPosition << endl;
    }

    if (myPosition == 0 || myPosition == myTankSize - 1)
    {
        myBumpCount++;
    }

    if (myPosition > myMaxPos)
        myMaxPos = myPosition;
    if (myPosition < myMinPos)
        myMinPos = myPosition;
}

int AquaFish::MaxPos() const
{
    return myMaxPos;
}

int AquaFish::MinPos() const
{
    return myMinPos;
}

int AquaFish::BumpCount() const
{
    return myBumpCount;
}

```

page 18-7

The bug that was introduced could be in one of three files: aquamain.cpp, aquafish.h, or aquafish.cpp. A syntax error will be caught by the compiler and will be easy to spot and correct in the appropriate file. If it's a logic error, such as initializing a value with a 1 instead of a 0, that type of error will be more difficult to catch and fix because it could be in any one of three files.

page 18-8

```
// aquafish.h

#ifndef _AQUAFISH_H
#define _AQUAFISH_H

#include "apvector.h"

class AquaFish
{
public:
    AquaFish(int tankSize);
    void Swim(); // Swim one foot.
    int BumpCount() const; // Return the bump count.
    int NumVisits(int index) const; // Return number of visits
    // at that index

private:
    int myPosition;
    int myBumpCount;
    int myTankSize;
    bool myDebugging;
    apvector<int> myVisits;
};

#endif

-----

// aquafish.cpp

#include <iostream.h>
#include "aquafish.h"
#include "randgen.h"

AquaFish::AquaFish(int tankSize)
    : myPosition(tankSize/2),
      myTankSize(tankSize),
      myBumpCount(0),
      myDebugging(true),
      myVisits(tankSize,0)
{
}
```

```

void AquaFish::Swim()
{
    RandGen randomVals;
    int flip;

    if (myPosition == myTankSize - 1)
    {
        myPosition--;
    }
    else if (myPosition == 0)
    {
        myPosition++;
    }
    else
    {
        flip = randomVals.RandInt(2);

        if (flip == 0)
        {
            myPosition++;
        }
        else
        {
            myPosition--;
        }
    }

    if (myDebugging)
    {
        cout << "*** Position = " << myPosition << endl;
    }

    if (myPosition == 0 || myPosition == myTankSize - 1)
    {
        myBumpCount++;
    }
    myVisits[myPosition]++;
}

int AquaFish::NumVisits(int index) const
// precondition: 0 <= index < myTankSize.
// postcondition: returns number of times that the fish
//                visited position index.
{
    return myVisits[index];
}

```

Marine Biology Case Study Part 2

Experimenting with the program

Text reference: pages 19-28

Suggestions for integration into course curriculum

One of the important concepts to teach in a computer science course is abstraction — using a program or code segment without knowing how it works. Abstraction in the context of using a program means that the user of the program needs only to follow the rules about using the program. For example, the rules about using a web browser are pretty simple: click on the hyperlinks, type in a URL, or type in keywords in a search engine. Using a class in C++ involves knowing how to instantiate an object using a constructor and how to use all the member functions to access or manipulate the object. At any level, abstraction tries to hide the details of implementation from the user.

This section of the case study provides an opportunity to experience the Marine Biology program as an abstraction. We want to treat the program as a black box and simply test it. The idea of a black box is that you supply input to the box, push some buttons on the box (call some routines), and examine the output from the box without seeing the internals of the box. At this point in our study of the program, we want to approach it as a black box, test it, supply different input values, and see how it performs. Learning about the behavior of the program will provide the overview before we dig into the code behind the scenes.

As you teach this section, have students study the program as a black box. Don't dig into the details yet. Let them play with the program, change the data file that supplies the starting fish locations, and learn by observing the behavior of the program. Later sections will dig into the details of the code.

Teaching suggestions and strategies

- Remember to emphasize a “black box” type of approach to the initial studying of the program.
- Students will be asked in the text to experiment with different scenarios. Different sizes of environments and different quantities of fish are controlled by the content of the fish.dat file. For example, the following file would create a 5 x 7 environment with 2 fish:

```
5 7  
0 1  
1 5
```

The first two numbers give the dimensions of the environment. Each successive pair of numbers gives the location of each fish.

You might need to review the documentation in `environ.h` that explains the format of the data file, `fish.dat`. It might be a good idea to discuss how to set up a certain sized environment with fish in specific locations.

Explanations of code

A sample `fish.dat` file is provided with the code. You may need to move it to a different location depending on where your compiler looks for data files.

Answers to exercises

page 20-1 `simulate.h`

page 20-2 The first line in main

```
ifstream input("fish.dat");
```

would be changed by replacing "fish.dat" with a different file name (including the full path if necessary).

Your students may also find it helpful to change the code in the main function in the `fishsim.cpp` file so that the user is prompted for the fish data file name, rather than always using "fish.dat". This makes it easier to experiment with several different fish data files. Simply replace the one line of code

```
ifstream input("fish.dat");
```

with the following four lines of code.

```
apstring fishfile;
cout << "Enter file name for fish: ";
cin >> fishfile;
ifstream input(fishfile.c_str());
```

page 20-3 The modified code is shown below for the for loop in `fishsim.cpp`

```
for (step = 0; step < numSteps; step++)
{
    sim.Step(env);
    display.Show(env);
    if (step % 5 == 0)
    {
        cout << " step " << step << " (press return)";
        getline(cin,s);
    }
}
```

page 20-4 Some common strategies include running the program using different data or sample input, reading the documentation supplied with the program, and making minor changes to the code to see what effect it has on the execution of the program.

page 22 **Stop and help.** Possible hypotheses: the fish move one square at a time; fish movement is random; the possible moves are up, down, left, or right; fish don't seem to die off; the fish don't seem to be reproducing; the fish do not move outside the boundaries of the environment; fish are not able to share a location.

page 22 **Stop and predict.** Some possible answers include designing data files to test different scenarios such as a small environment (3 x 3) with one fish, a small environment with two fish, checking the life and death behavior of a small population of fish in a small environment, a small environment (3 x 3) filled with 8 fish, etc. All of these scenarios will help confirm or challenge different hypotheses about the program.

page 24 **Stop and predict.** A fish could move to a location that prevents a subsequent fish from moving. The order of fish movement will have an effect on the movement of each fish.

page 24 **Stop and help** (first one). The fish seems to move left and right in a random pattern, much like the fish in a five foot tank from part one. The fish in the 1 x 5 environment might not behave the same depending on its behavior as it runs into the walls. A fish in the marine biology case study has the potential of 4 or possibly 8 moves (including diagonals) and we do not yet know its behavior as it runs into walls.

page 24 **Stop and help** (second one). It seems that the direction selected is random and limited to one of four possible directions: up, down, left, or right. When it runs into the boundary of the environment, it moves along or away from the boundary. The fish does not seem to "escape" outside the environment.

page 28-1 When the program was run multiple times, the fish moved from the starting configuration on the left to the configuration shown on the right:

<u>Starting</u>	<u>Final</u>
A _	B A
B C	C _

Given the consistency of the output, the pattern of movement is not random. Two possible patterns of movement could produce this result: row by row, left to right, or column by column, top to bottom.

page 28-2 Making an assumption that the movement is row by row, left to right, the resulting configuration should look like this:

A	C	E	
D	B	H	Running the program confirmed this result.
F	G	_	

page 28-3 The program took a long time to run, and a single 'A' appeared briefly in the upper left-hand corner. It seems that the program spent time setting up this large environment and printing out blank characters. In one experiment on a 2000 x 2000 environment the computer had to be reset. The maximum size will vary depending on the resources (memory and processing speed) of the computer.

- page 28-4 An error message was printed: "error, attempt to create fish at non-empty: (1,1)". If a position in the environment already contains a fish, it apparently is an error to try to create another fish at the same location.
- page 28-5 The sequence of fish positions in fish.dat does not matter. The two suggested fish.dat files produced the same pattern of movement. Different letters occupied the positions in each trial, but the pattern of movement was consistent.
- page 28-6 This program produces a new data file "newfish.dat" so that the original fish.dat file is unchanged.

```
// makeFish.cpp

// This program asks the user for the row size, col size, and
// the number of fish. It then prompts the user for the
// positions of the fish and writes the data values to the
// file newfish.dat in the format specified in environ.h

#include <iostream.h>
#include <fstream.h>

int main()
{
    int rowSize, colSize, howMany, count;
    int row, col;
    ofstream outfile ("newfish.dat");

    cout << "Enter row size -> ";
    cin >> rowSize;
    cout << "Enter col size -> ";
    cin >> colSize;
    cout << "Enter number of fish -> ";
    cin >> howMany;

    outfile << rowSize << "    " << colSize << endl;

    for (count = 1; count <= howMany; count++)
    {
        cout << "fish " << count << ", enter row position -> ";
        cin >> row;
        cout << "fish " << count << ", enter col position -> ";
        cin >> col;
        outfile << row << "    " << col << endl;
    }

    return 0;
}
```

Another exercise is to generate a fish.dat file of randomly generated fish using the RandGen class. The sample program below can be used to generate this file. All the code files except display.cpp must be compiled for this program because of the interdependencies.

```
// randfish.cpp

// This program asks the user to specify the number of
// rows and columns for an environment and the number of
// fish to be placed in this environment. It then
// writes a file with the number of rows and columns specified
// on the first line and one line for each randomly generated
// fish. The resulting file is in the format specified
// in environ.h
//
// In order to avoid duplication, an apvector of positions
// generated is maintained and checked each time a new
// position is generated, so that all positions are unique.

#include <iostream.h>
#include <fstream.h>

#include "apstring.h"
#include "apvector.h"

#include "position.h"
#include "randgen.h"

int main()
{
    int numRows, numCols, numFish;
    int fishCount;
    int k;
    Position pos;
    apstring filename;
    ofstream out;
    RandGen rand;

    cout << "Enter the number of rows: ";
    cin >> numRows;
    cout << "Enter the number of cols: ";
    cin >> numCols;
    cout << "Enter the number of fish to be generated: ";
    cin >> numFish;

    cout >> "Enter the name of the output file: ";
    cin >> filename;
    out.open(filename.c_str());

    apvector<Position> positionsFilled(numFish);

    out << numRows << " " << numCols << endl;
```

```

fishCount = 0;
while (fishCount < numFish)
{
    pos = Position(rand.RandInt(numRows), rand.RandInt(numCols));

    k = 0;
    while ((k < fishCount) && ! (pos == positionsFilled[k]))
        k++;

    if (k == fishCount)
    {
        positionsFilled[fishCount] = pos;
        fishCount++;
    }
}

for (k = 0; k < fishCount; k++)
    out << positionsFilled[k].Row() << " "
        << positionsFilled[k].Col() << endl;

out.close();

return 0;
}

```

Learning how the program stores fish

Text reference: pages 29-36

Suggestions for integration into course curriculum

This section provides a context to teach about alternative strategies of data storage. After you have covered the `apvector` and `apmatrix` classes in your course, you could use the case study to explore different ways of storing data as described on page 31 of the case study.

Teaching suggestions and strategies

In conjunction with their reading on page 30, students should skim through all eight header files to start understanding what the various types of objects can do. We are now moving from experimenting with the black box to opening it up and understanding how it works. This section focuses on how fish are stored in the environment.

Here is an interesting discussion to have with your students. What would be the advantages and disadvantages of the two data storage strategies presented in this section? One limitation of a container class such as an `apmatrix` is the restriction of storing one type of object, i.e., an `apmatrix` of fish. You might have students think about the problem of implementing a larger ocean. Suppose you wanted to store fish, land (islands), and polluted areas. The `apmatrix` could store the fish, but not the other two objects, land and polluted spots. Instead of an `apmatrix` of fish, the environment could be

implemented using multiple `apvectors` to store fish, land, and polluted areas. Of course the original implementation could have multiple matrices to accomplish the same result, but at a much larger cost of memory.

With an object-oriented approach to programming comes the complexity of interaction of classes. An object diagram that shows the interaction of classes for the `Environment` constructor is included in Appendix A. You will notice that some of the rectangles are contained completely inside of a class object, indicating that it is a private member function. Public member functions extend outside the class object box, indicating that other programs can use these public member functions. You might want to make transparencies of these diagrams, or display the web page during a presentation to help explain the detail of constructing an `Environment` object. The object diagrams can be found in the AP Computer Science section of the College Board website.

Explanations of code

At the top of some of the header files, you will see some class declarations that are uncommon. For example, in the `Display` class, just before the code begins, you see these lines:

```
// class declarations for those classes only used
// as references (passed by reference/const reference)

class Environment; // says that Environment is a class,
                  // details will follow later
```

This is similar to a forward declaration in Pascal; it signals that the identifier name `Environment` is, in fact, a class. This allows the `Display` class to use a reference to an `Environment` object as a parameter later in the header file. If the `Environment` object were a data member, the `environ.h` file would need to be included.

```
class Display
{
    public:
        Display(); // constructor
        void Show(const Environment & env); // now this will compile
        //... rest of file ...
```

You will notice that three of the classes, namely Fish, Neighborhood, and Position, include a function called ToString. When debugging, this allows easier presentation of important information about the current state of the object. Returning a string allows for a “stringized” representation of one object or many objects together as one string. This makes for some cool debugging tools in the program.

Answers to exercises

- page 32 **Stop and consider.** Using an explicit strategy, some of the operations that would be faster are counting, printing, etc. With the same explicit strategy, some of the operations that would be less efficient would be inserting a new value if the list was sorted, searching, and examining the neighbors of a specific position.
- page 35 **Stop and help.** Control of the order of processing of fish is currently in the AllFish member function of the Environment class. One way of changing the order of processing is to change the AllFish code. A second way is to change the Step function in the Simulation class so that it sorts the array of fish (fishList), causing fish to be moved in a different order.
- page 36-1 A function would be needed to sort the array of fish into the desired order based on their position. Once the fish have been sorted, the Step function would then move each fish.
- page 36-2 Advantage of having the Environment AllFish function manage the order of movement: The AllFish function already examines each location in the container class (apmatrix). By simply changing the nesting of the loop (row vs. column) and direction (ascending vs. descending), the order of processing can be easily modified.
- Disadvantage of having the Environment AllFish function manage the order of movement: The list of fish returned by AllFish is used for two different purposes. Simulate::Step uses the list to determine the order for processing fish; Display::Show uses the list to display all the fish. If AllFish sorts the list before returning it, then that order might be appropriate for one purpose but not the other. For example, if AllFish returns the list in bottom-up, right-to-left order for use in Simulate::Step, then a text-based version of Display::Show will need to sort the list a second time in order to display the fish.

Learning how fish interact with the environment

Text reference: pages 36-42

Teaching suggestions and strategies

This section examines the problem of moving fish and keeping track of their positions in the environment. The program has a fish storing its own position, and the environment also keeps track of fish positions indirectly by storing fish objects in the apmatrix. This is the most complicated section to understand as we learn how three classes (Environment, Fish, and Neighborhood) interact to move fish.

In this section, students will begin making major modifications to the code in various classes. Make sure that students save an original copy of the code as they make changes to the classes. Later on in the case study they will want to compare the output of the original code with modified versions. We suggest putting the original and modified code in separate directories rather than renaming them.

The series of exercises on page 42 explores three different alternatives to selecting an empty position to which a fish can move. One of the potential modifications the biologists requested was considering different movement possibilities. After students have done the exercises on page 42, you might have students expand the movement options to all eight directions to include diagonals. Before you let them loose on this assignment, some small or large group discussion about the best approach would be valuable. Topics such as terminology (northeast, northwest, etc.) and the process of selecting a direction are potential starters.

Answers to exercises

- page 37 **Stop and consider.** Exploring the code down to a detailed level might lead to an understanding of the interaction of the objects involved in moving a fish. Conversely, stepping back and thinking about the big picture might lead to understanding of the issues that led to such a complex answer. Understanding the issues of fish movement will also help when it comes time to modify the program to support a larger ocean.
- page 40 **Stop and help.** A summary of the fish movement process: 1) the fish collects empty neighboring positions; if there is an empty nearby position, it continues; 2) it stores its current position before moving to a new position; 3) it selects a random new position from the list of empty neighboring positions; 4) it then tells the environment to update its new position.
- page 42-1

```
bool operator != (const Position & lhs, const Position & rhs)
// postcondition: returns true iff lhs != rhs
{
    return ! lhs.Equals(rhs);
}
```
- page 42-2 There is a slight advantage to moving the random neighbor selection process from the Fish class to the Neighborhood class as described. In either scenario, the Move function must fill the Neighborhood object (nbrs) with potential empty positions, test for (nbrs.size > 0), then randomly select a position. However, moving the random number generation code to the Neighborhood class better encapsulates the sense of an unordered collection and makes the code in the Move function easier to read. On the other hand, it would be more difficult to step through all the positions in a neighborhood if the Select function returns a randomly chosen position. For example, if a neighborhood contains four positions, one could not easily step through all the positions simply by calling Select four times, because two or more of these calls might return the same position.

page 42-3

```
Position Neighborhood::Select() const
// precondition: 0 < Size()
// postcondition: returns a random Position in Neighborhood
{
    RandGen randomVals;

    return myList[randomVals.RandInt(0, Size() - 1)];
}

void Fish::Move(Environment & env)
// precondition: Fish stored in env at Location()
// postcondition: Fish has moved to a new location
//                 in env (if possible)
{
    Neighborhood nbrs = EmptyNeighbors(env, myPos);
    DebugPrint(3, nbrs.ToString());

    if (nbrs.Size() > 0)
    {
        // there were some empty neighbors, so randomly choose one
        Position oldPos = myPos;
        myPos = nbrs.Select(); // uses new member function
        DebugPrint(1, "Fish at " + oldPos.ToString()
                  + " moves to "
                  + myPos.ToString());
        env.Update(oldPos, *this);
    }
    else
    {
        DebugPrint(1, "Fish " + ToString() + " can't move.");
    }
}
```

page 42-4

```
Neighborhood Environment::EmptyNeighbors(const Position & pos)
// postcondition: returns a Neighborhood with empty positions
//                 around pos in N, S, E, W directions
{
    Neighborhood nbrs;

    if (IsEmpty(pos.North()))
        nbrs.Add(pos.North());
    if (IsEmpty(pos.South()))
        nbrs.Add(pos.South());
    if (IsEmpty(pos.East()))
        nbrs.Add(pos.East());
    if (IsEmpty(pos.West()))
        nbrs.Add(pos.West());
    return nbrs;
}
```

```

void Fish::Move(Environment & env)
// precondition: Fish stored in env at Location()
// postcondition: Fish has moved to a new location
//                in env (if possible)
{
    RandGen randomVals();
    Neighborhood nbrs = env.EmptyNeighbors(myPos);
    DebugPrint(3, nbrs.ToString());

    if (nbrs.Size() > 0)
    {
        // there were some empty neighbors, so randomly choose one
        Position oldPos = myPos;
        myPos =
            nbrs.Select(randomVals.RandInt(0, nbrs.Size() - 1));
        DebugPrint(1, "Fish at" + oldPos.ToString()
                    + " moves to "
                    + myPos.ToString());
        env.Update(oldPos, *this);
    }
    else
    {
        DebugPrint(1, "Fish " + ToString() + " can't move.");
    }
}

```

page 42-5

To support this constructor, the `AddIfEmpty` function was moved from the `Fish` to the `Neighborhood` class.

```

void Neighborhood::AddIfEmpty(const Environment & env,
                             const Position & pos)
// postcondition: pos is added to nbrs if pos in env and empty
{
    if (env.IsEmpty(pos))
    {
        Add(pos);
    }
}

```

Here's the new `Neighborhood` constructor that queries the environment around `Position pos` for empty positions.

```

Neighborhood::Neighborhood(const Environment & env,
                           const Position & pos)
    : myList(4),
      myCount(0)
// postcondition: Size() == # positions in Neighborhood
{
    AddIfEmpty(env, pos.North());
    AddIfEmpty(env, pos.South());
    AddIfEmpty(env, pos.East());
    AddIfEmpty(env, pos.West());
}

```

And finally, the revised Fish member function, Move.

```
void Fish::Move(Environment & env)
// precondition: Fish stored in env at Location()
// postcondition: Fish has moved to a new location
//                in env (if possible)
{
    RandGen randomVals;
    Neighborhood nbrs(env, myPos);
    DebugPrint(3, nbrs.ToString());

    if (nbrs.Size() > 0)
    {
        // there were some empty neighbors, so randomly choose one
        Position oldPos = myPos;
        myPos =
            nbrs.Select(randomVals.RandInt(0, nbrs.Size() - 1));
        DebugPrint(1, "Fish at" + oldPos.ToString()
                    + " moves to "
                    + myPos.ToString());
        env.Update(oldPos, *this);
    }
    else
    {
        DebugPrint(1, "Fish " + ToString() + " can't move.");
    }
}
```

page 42-6

Each change required a moderate amount of rewriting the code. Moving the random selection process to the Neighborhood class (question 3), or moving the filling of the Neighborhood object to the Environment class (question 4) simply shifts code and responsibility without any clear advantage or disadvantage.

Thinking about the entire problem, it seems to make the most sense to have the environment control the determination of any empty neighbor spot to move a fish. A Fish object could make that determination by checking locations in the environment. The Neighborhood class seems to be a helper class to organize the process. Yet from a code standpoint, there is an advantage to moving the determination of an empty position into the Neighborhood class. It consolidates the code about empty positions and potential moves into one class, so that, if the program needed to support more than four directions (N,S,E,W), only one class needs to be modified, the Neighborhood class. The program in its current form, or using either of the first two modifications, would require changes to two classes to support a different quantity of directions.

page 42-7

The original design:

Neighborhood selects the random position, page 42-2

A fish is told to move by the Simulation object
The fish gets a list of empty neighbors from the environment

A fish is told to move by the Simulation object
The fish gets a list of empty neighbors from the environment

If there is an empty neighbor, the fish gets a random number and gets a position from the Neighborhood object to move to

If there is an empty neighbor, the fish asks the Neighborhood object to return a random position stored in Neighborhood

The fish informs the Environment of its new position, which is updated

The fish informs the Environment of its new position, which is updated

The only difference between the original design and the modification suggested in question 42-2 is the responsibility for selecting the random position to move to. In the original design, the fish takes responsibility for selecting a random number, which in turn is used to get a position from the Neighborhood object. In the modified version, the Neighborhood object takes responsibility for selecting a random position within itself and returns that position to the Fish object.

page 42-8

Answers may vary, but most people would probably respond with continuing to explore the code, rather than design the code. Designing code is hard work and often leads to several cycles of design and coding. One of the advantages of trying to design code is you will probably end up understanding the problem to a greater depth than just by reading someone's else code. It might lead to an alternative solution because you are starting from scratch with the design. The drawback is this takes more time. But eventually you will encounter a situation where original design work is required and prior experience will be helpful.

Understanding the rest of the program

Text reference: pages 43-48

Teaching suggestions and strategies

The author of the program included debugging statements to help track key events in the program. Given that fish moves involve a random choice of a direction, it is difficult to verify if the program is working correctly. By changing the LEVEL_OF_DEBUG_DETAIL constant value in the utils.cpp file, different debugging messages are printed out by the program. The chart is on page 45. A good teaching strategy might involve working with a small environment (say 3 x 3) and a few fish and observe debugging messages for different levels. If you have access to a computer LCD panel or projector, I would do this as a class discussion and run the program in front of the entire class.

Answers to exercises

page 44 **Stop and help.** The following configuration would not be printed correctly if the AllFish function returned fish ordered column-by-column.

```
_ B _  
A _ _      fishList would be stored as: A(1,0), B(0,1), C(2,2)  
_ _ C
```

The nested loops in the Show function of the Display class process fish row-by-row, left to right, while checking the position of the fish in fishList in sequential order. The 'A' fish would be printed first since it's the first fish in fishList, but then fish 'B' would never be printed because the nested for loops have already passed over row 0.

```
page 48-1      void Display::Show(const Environment & env)  
                 // postcondition: state of env written as text to cout  
                 {  
                     const int WIDTH = 1;     // for each fish  
  
                     int rows = env.NumRows();  
                     int cols = env.NumCols();  
                     int r, c;  
                     Fish tempFish;  
  
                     for (r = 0; r < rows; r++)  
                     {  
                         for (c = 0; c < cols; c++)  
                         {  
                             tempFish = env.FishAt (Position(r, c));  
                             if (tempFish.IsUndefined())  
                             {  
                                 cout << setw(WIDTH) << ' ';  
                             }  
                             else // this is a position with a fish  
                             {  
                                 cout << setw(WIDTH) << tempFish.ShowMe();  
                             }  
                         }     // finished processing all columns in a row  
  
                     cout << endl;  
  
                     }     // finished processing all rows in the grid  
                 }  
                 }
```

page 48-2 Somewhere in the overall program, a function would have to rearrange the fish into some random order. The function could be in the Environment class, the Simulation class, or in the utils.cpp file. The function would need to utilize a random number generator to rearrange the fish in some pseudorandom order. The Step function would then move each fish in the array, which would be in a random order based on position. This problem is tricky because the random sequence would include duplicate positions, and the code would have to avoid them in arranging the fish positions.

For students interested in exploring this shuffling algorithm, you might assign a program to

- fill an `apvector` with integers,
- randomly rearrange the integers in the vector.

Doing this as a separate exercise, outside the complexities of this case study, allows the student to concentrate on this interesting algorithm.

page 48-3 After making the modifications to support moving fish in random order, change the `LEVEL_OF_DEBUG_DETAIL` constant to 1 in the `utils.cpp` file. This will cause fish id and positions to be printed out with both the before and after positions. Several simulations should be run for multiple steps using different `fish.dat` files to confirm the movement of fish in random order.

page 48-4 You could create a Fish data member, `myLastPosition`, which can be used to determine the direction the fish has just moved. You need this in order to keep track of what moving forward means. Use `EmptyNeighbors` to fill the array as follows. If the forward position is empty, store that location in the array twice. Store each available location to the fish's left or right in the array once. If the array is not empty, randomly select a position and move. Otherwise, try to move backward. If that space is empty, the fish does not move.

Another variation, not involving probability, would be to have the fish always attempt to move forward first, then to one side or the other, and finally to return to its former position. To do this, first test if the next position in the same direction is empty. If that position is empty, the fish moves there. Otherwise it looks for positions to its right or left and selects one. If none of those is empty, it tries looking behind itself (which is really `myLastPosition`) and moves backward. Of course, if all four positions are occupied, the fish does not move.

page 48-5 The Simulation class manages the tasks of stepping or running the simulation. The Step function takes care of filling the array of fish and calling Move for each fish. The Run member function simply performs multiple calls of Step. It packages the stepping of a program and keeps the main client program (`fishsim.cpp`) nice and clean. It may also allow for a layer of abstraction to be maintained between the client program and the lower level classes (Environment and Fish) as the program is modified.

page 48-6 If the main client program starts calling member functions that are rather detailed and low level such as `AllFish` and `Move`, it's probably a good idea to create an intermediary class such as Simulation. If the strategies were to change in the Environment or Fish class and stepping through the environment required different member function calls, the Simulation class would hide those changes from the client program.

Appendix A

Object Diagrams

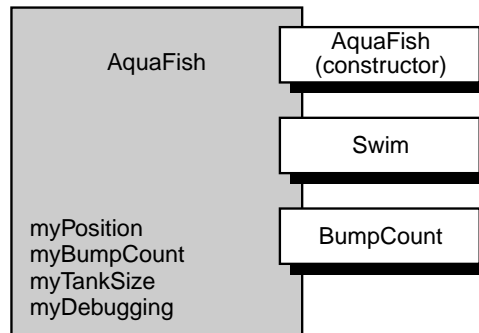
This appendix contains diagrams that illustrate object interactions in the AP Computer Science Marine Biology Case Study.

In these object diagrams, each class is represented by a rectangle. Most of these diagrams show only the public and private member functions of the class, which appear in smaller rectangles. The **public member functions**, which represent the class interface, overlap the edge of the rectangle. The **private member functions** are encapsulated inside the class rectangle. The AquaFish object diagram shows public member functions and private data members; there are no private member functions in this class. The **private data members** are encapsulated inside the class rectangle, but do not appear in smaller rectangles.

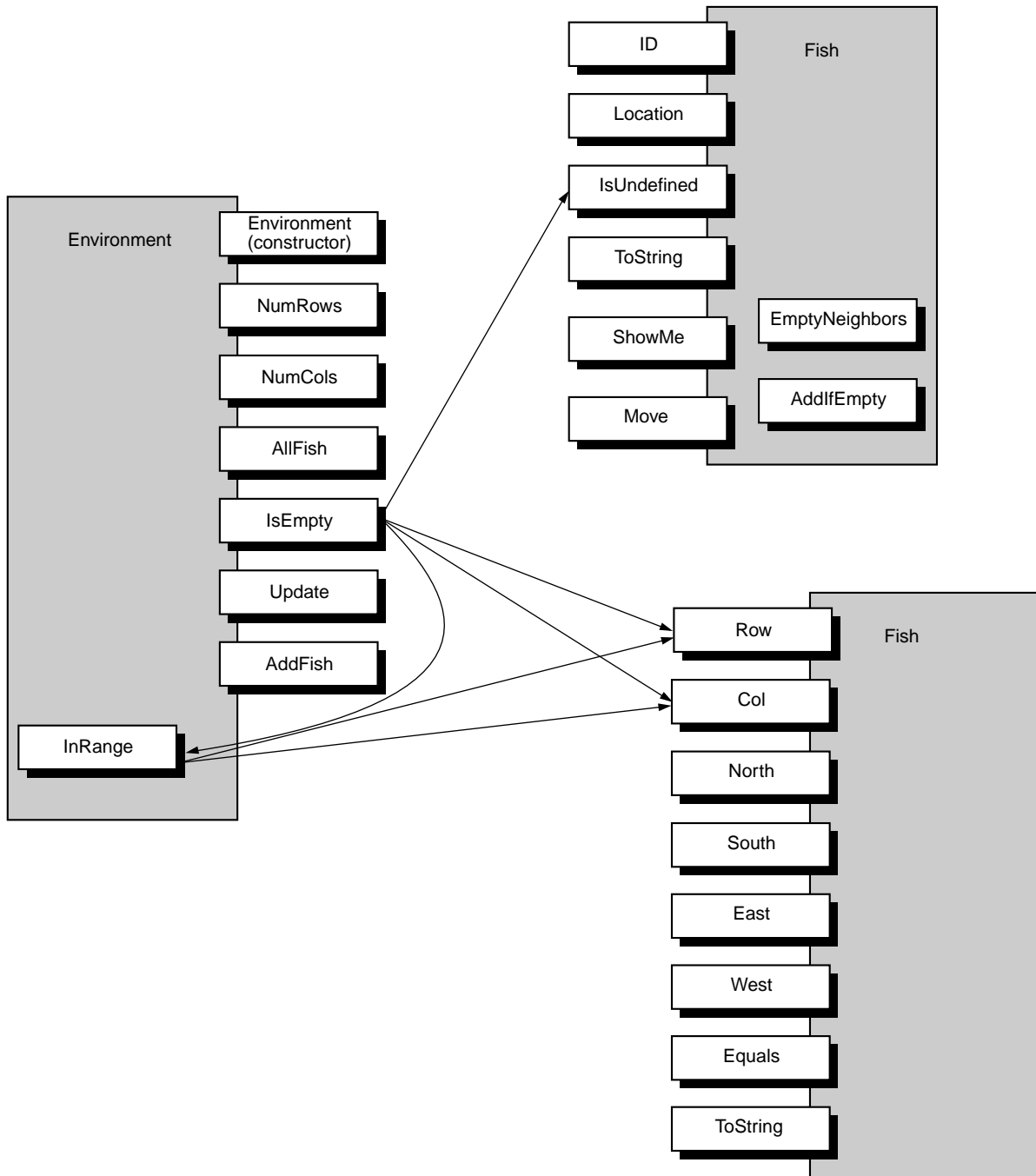
The object diagrams listed below appear on the following pages:

- The AquaFish Class
- Environment Construction
 - Environment::IsEmpty
- Top-Level Interactions of the Simulation
 - Simulation::Step
 - Fish::Move
 - Fish::EmptyNeighbors
 - Environment::Update
 - Display::Show

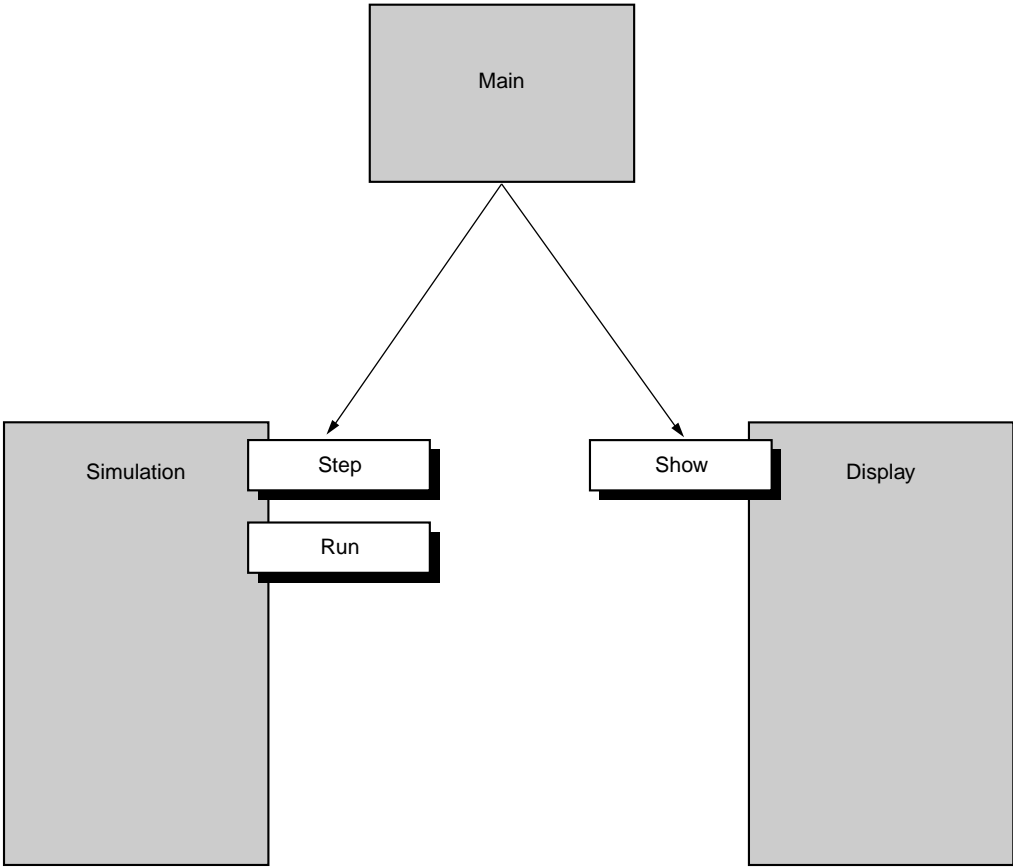
AquaFish Class



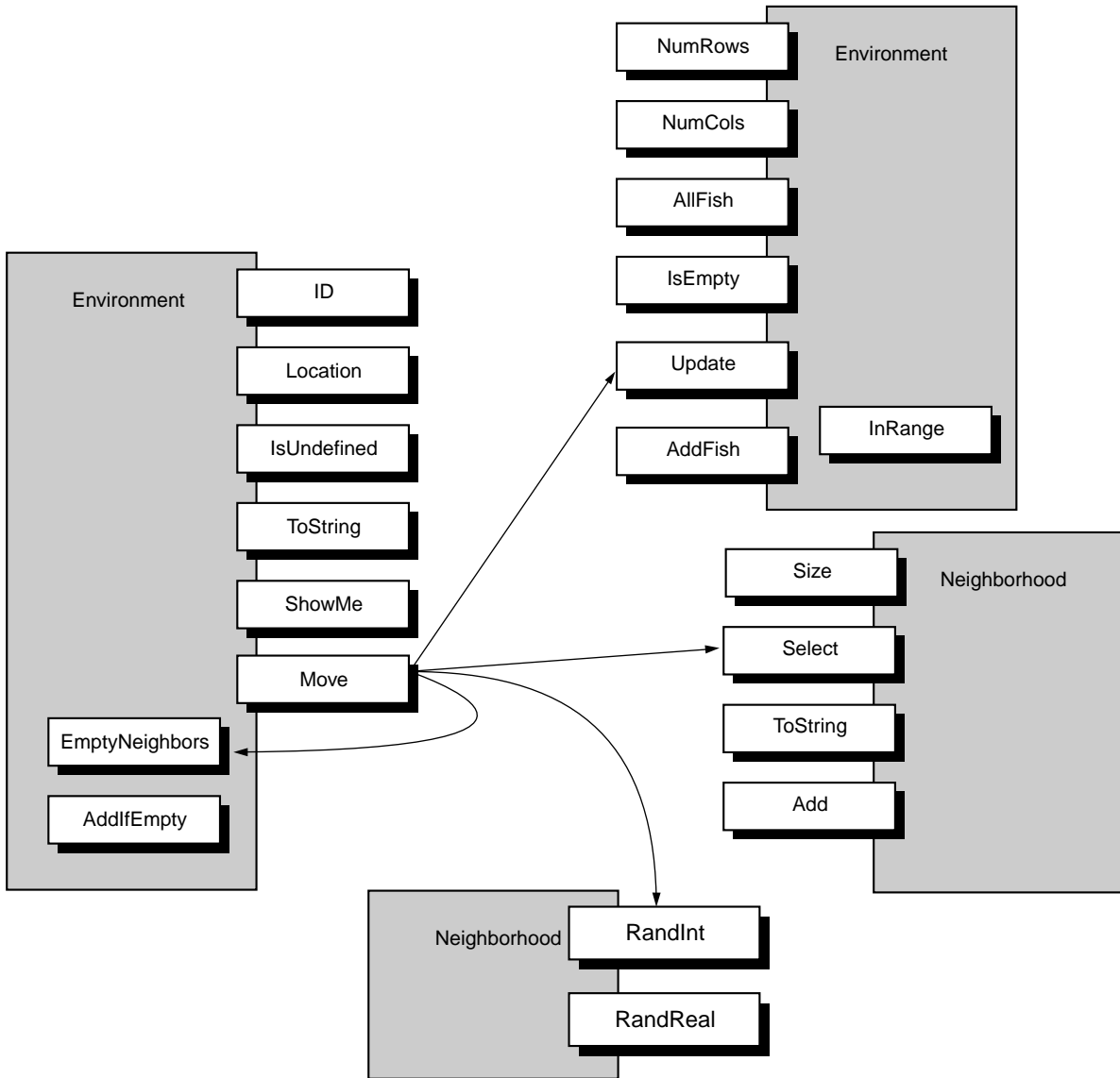
Focus on Environment IsEmpty



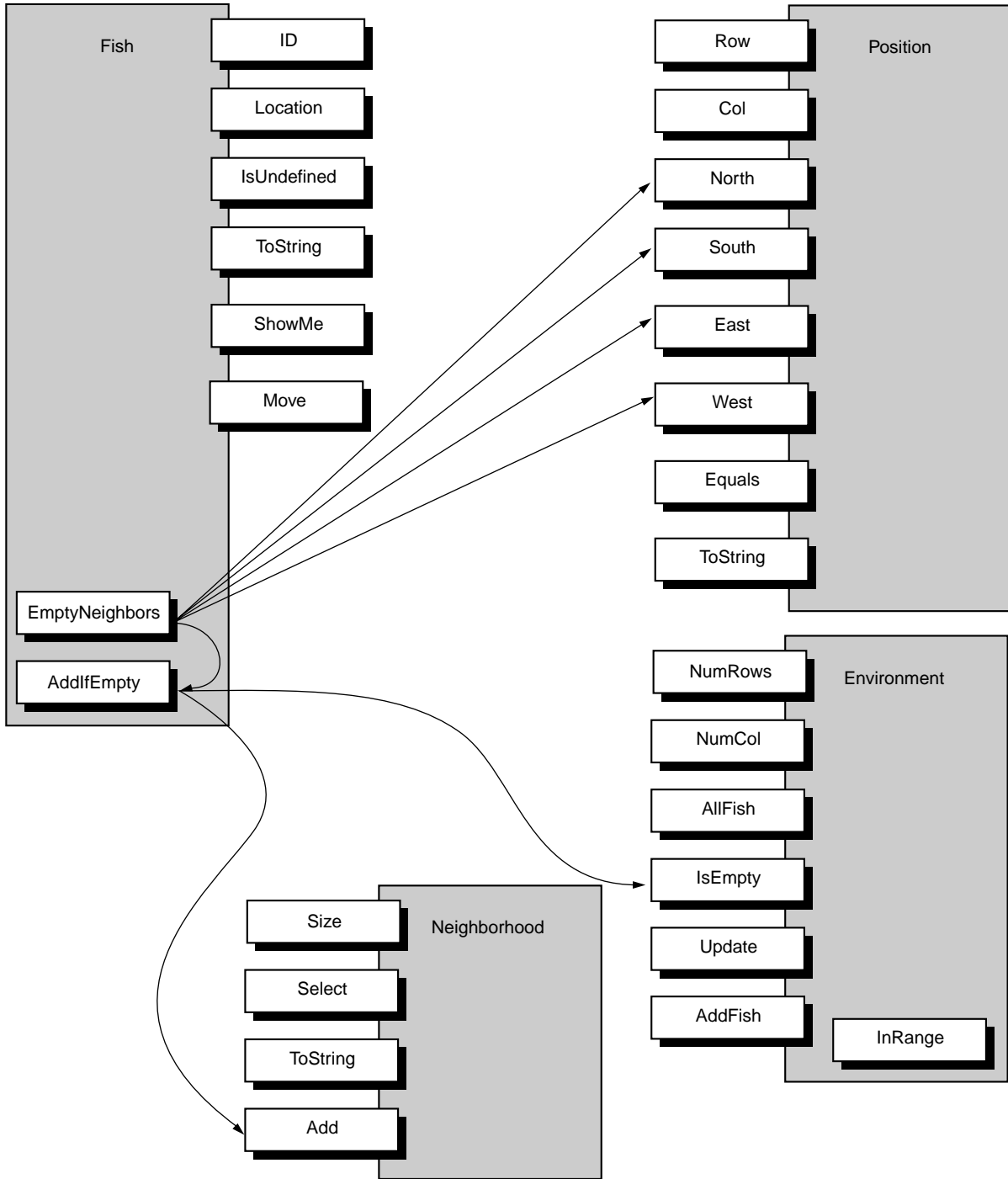
Top-Level Interactions



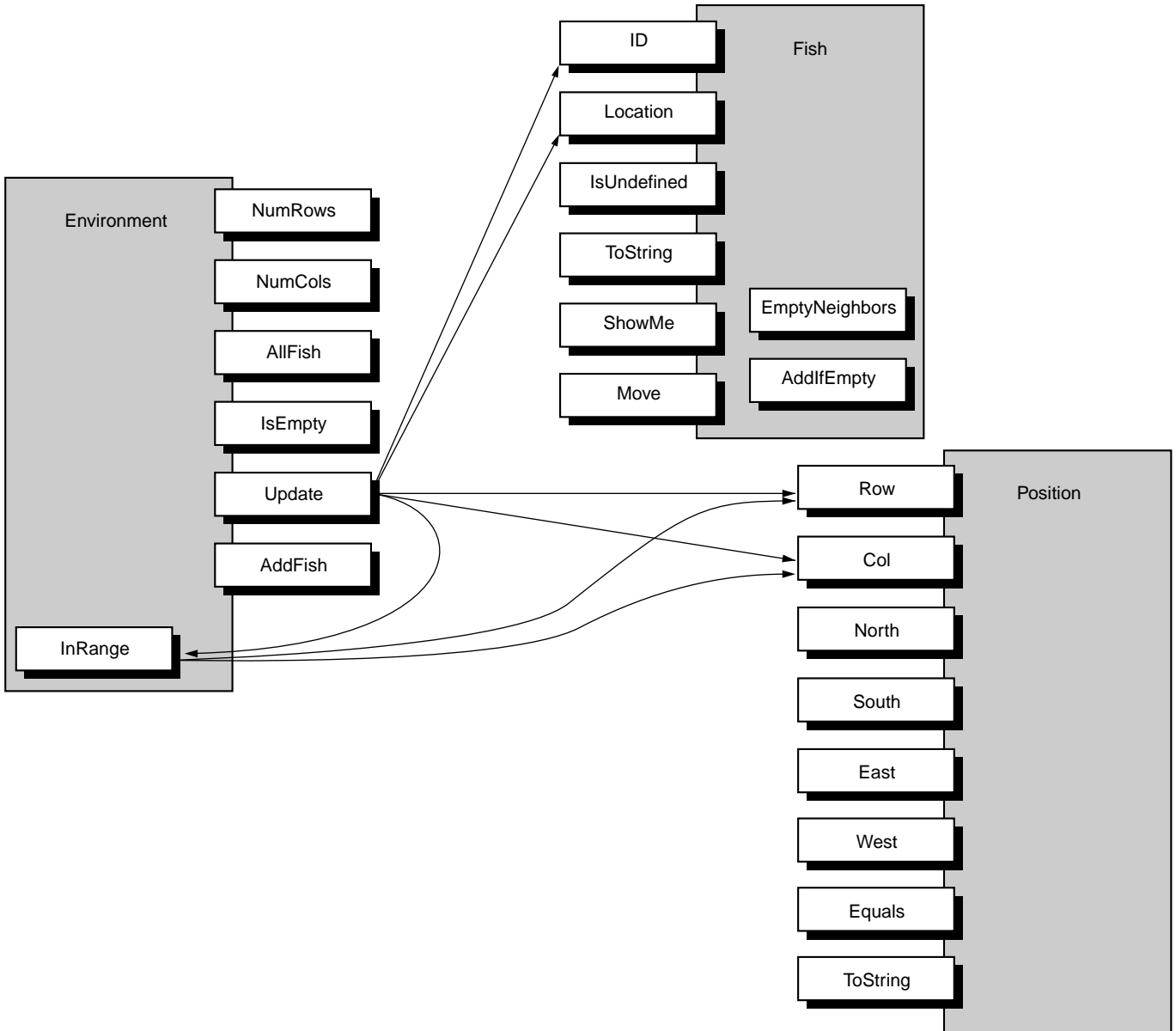
Focus on Fish::Move



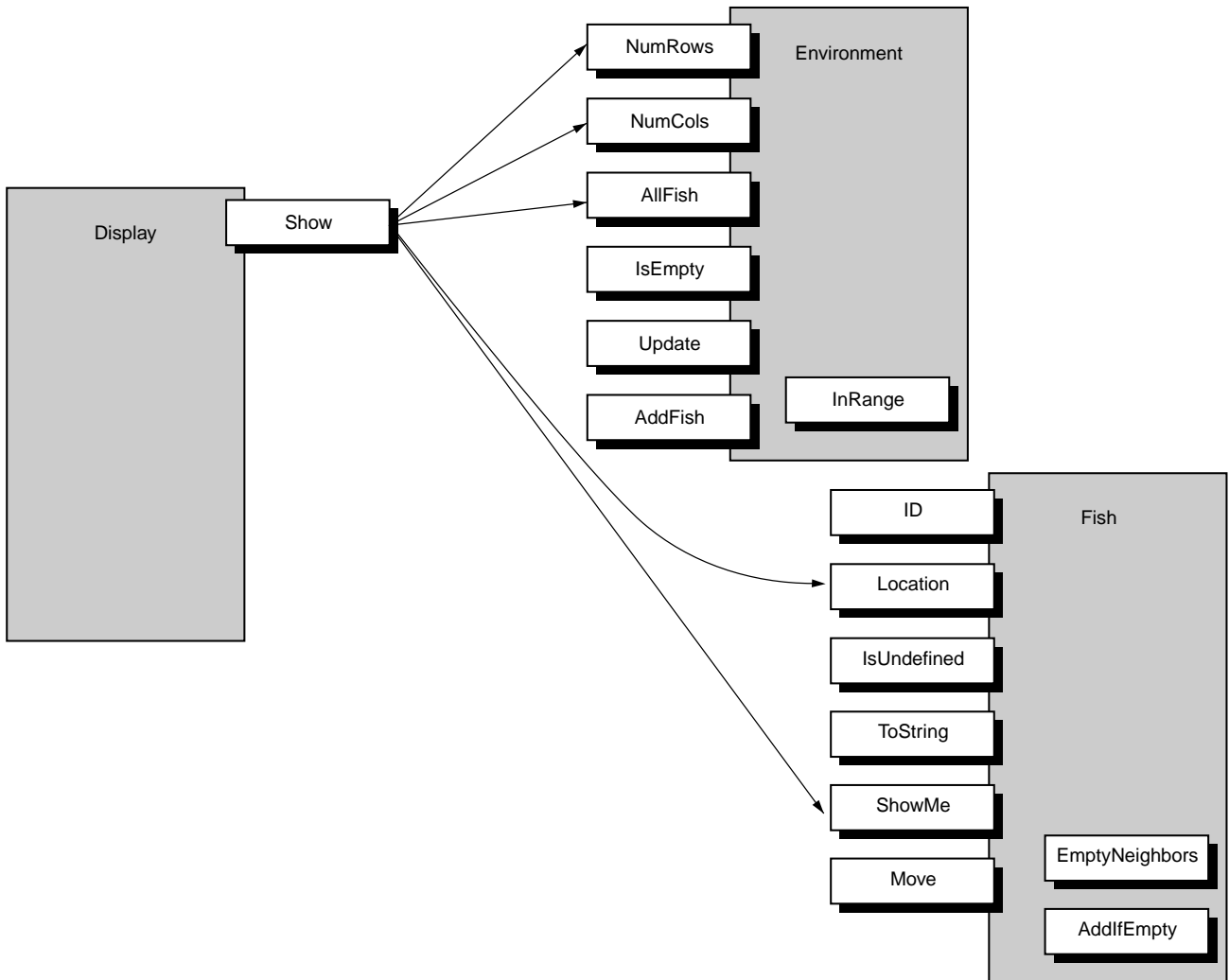
Focus on Fish::EmptyNeighbors



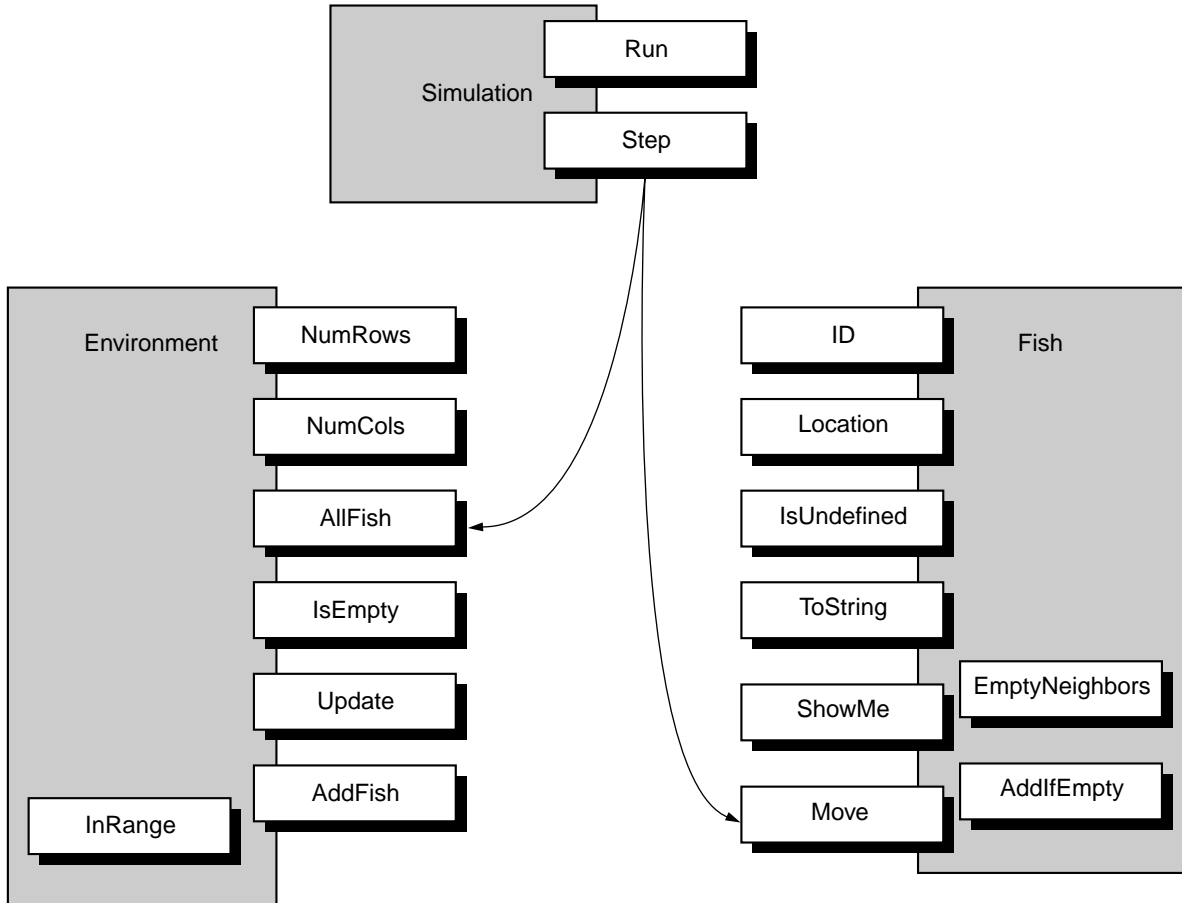
Focus on Environment::Update



Focus on Display::Show



Focus on Simulation::Step



Appendix B

Sample Multiple-Choice Questions

1. Consider a decision to change the implementation of the Position class to store a position as an apstring instead of a pair of integers. Consider the following groups of classes.

- I. Fish, Neighborhood
- II. Environment, Display
- III. Simulation

Which of these classes will require modification?

- (A) I only
- (B) II only
- (C) I and II only
- (D) I, II, and III
- (E) none

2. In the AquaFish class, it seems very inefficient to check for myPosition both at the beginning and at the end of the Swim function. A proposal has been made for combining the checks at the end with the separate beginning checks.

Consider the following revised version of the Swim function in the AquaFish class with the proposed changes highlighted in bold.

```
void AquaFish::Swim()
{
    RandGen randomVals;
    int flip;

    if (myPosition == myTankSize - 1)
    {
        myPosition --;
        myBumpCount++;
    }
    else if (myPosition == 0)
    {
        myPosition ++;
        myBumpCount++;
    }
}
```

```

else
{
    flip = randomVals.RandInt(2);
    if (flip == 0)
    {
        myPosition ++;
    }
    else
    {
        myPosition --;
    }
}
if (myDebugging)
{
    cout << "****Position = " << myPosition << endl;
}

// eliminate the following lines:
// if (myPosition == 0 || myPosition == myTankSize - 1)
// {
//     myBumpCount++;
// }
}

```

How will the proposed changes affect the execution of the simulation?

- (A) The simulation will produce the same results as the original in all cases.
- (B) The simulation will produce the same results as the original only when myTankSize is an even number.
- (C) The simulation will produce the same results as the original only when myTankSize is an odd number.
- (D) The simulation will produce the same results as the original only when the fish hits either end of the tank with its last move.
- (E) The simulation will never produce the same results as the original in any situation.

3. Consider the following configuration of fish.

A B C
D - -

If scanning is done in row order (i.e., top-to-bottom, with each row left-to-right) how many different configurations are possible after two complete steps of the simulation?

- (A) 1
- (B) 2
- (C) 3
- (D) 4
- (E) 5

4. Consider the following two alternatives for constructing a neighborhood of empty positions.

- I. In the current implementation, the responsibility for building a neighborhood of empty neighboring positions is in the Fish class.
- II. An alternative implementation would be to create a new Neighborhood constructor that takes a position as a parameter and constructs a neighborhood of the empty neighbors around that position.

Which of the following is FALSE?

- (A) If Implementation II were used, the EmptyNeighbors and AddIfEmpty member functions in the Fish class would no longer be necessary.
- (B) If Implementation II were used, the public member function Add in the Neighborhood class could become a private member function.
- (C) If Implementation II were used, the body of the if statement in the Move member function of the Fish class that randomly selects a position in the neighborhood would be the same as in Implementation I.
- (D) Modifying the program to be able to construct different kinds of neighborhoods (for example, a neighborhood of non-empty neighbors) in Implementation I would require the addition of new Neighborhood member functions.
- (E) Modifying the program to be able to construct different kinds of neighborhoods (for example, a neighborhood of non-empty neighbors) in Implementation II would require the addition of new Neighborhood member functions.

5. Consider the following program, which is intended to do ten trials of the AquaFish simulation for a specified number of steps and print the average (arithmetic mean) number of times a fish starting from the middle of the tank bumps the ends of the tank. This program does not work as intended. Some lines of the program have been numbered for reference.

```
int main()
{
    int tankSize, numSteps, trial, step;
    int tot = 0;

    cout << "Tank size? ";
    cin >> tankSize;
    cout << "Steps per simulation: ";
    cin >> numSteps;

Line 1:    AquaFish fish(tankSize);
Line 2:    for (trial = 0; trial < 10; trial++)
Line 3:    {
Line 4:        for (step = 0; step < numSteps; step++)
Line 5:        {
Line 6:            fish.Swim();
Line 7:        }
Line 8:        tot = tot + fish.BumpCount();
Line 9:    }

Line 10:   cout << "Average number of Bumps per " << numSteps;
Line 11:   cout << " step simulation is " << tot / 10.0 << endl;

    return 0;
}
```

Which of the following changes would make the program work correctly?

I. Add the following statement immediately before line 4.

```
fish.myPosition = tankSize / 2;
```

II. Remove line 8 and replace line 11 with the following statement.

```
cout << " step simulation is " fish.BumpCount() / 10.0 << endl;
```

III. Replace lines 4 - 8 with the line

```
tot = tot + Count(tankSize, numSteps);
```

where the function Count is defined as follows.

```
int Count(int tankSize, int numSteps)
{
    int step;
    AquaFish fish(tankSize);

    for (step = 0; step < numSteps; step++)
        fish.Swim();

    return fish.BumpCount();
}
```

- (A) I only
- (B) II only
- (C) III only
- (D) II and III only
- (E) I, II, and III

Sample Free-Response Question

Note: The intent of this sample question is to give the reader an idea of the type of question that might be asked in the context of the case study. This question deals with modifying existing code. Other types of questions might involve adding functionality to a class by adding a member function. The exercises in the case study contain examples of adding member functions to some of the classes.

This question involves reasoning about the code from the Marine Biology Case Study. A copy of the code is provided as part of this exam.

In the existing case study code, the responsibility for identifying an empty neighboring position is shared between the Fish and Neighborhood classes. Consider modifying the Neighborhood class so that it has all the responsibility for identifying empty positions for possible fish moves.

The two private functions, EmptyNeighbors and AddIfEmpty, will be removed from the Fish class. The code from the EmptyNeighbors function will be moved into a Neighborhood constructor, and the AddIfEmpty function will become a private member function of the Neighborhood class. A third change will involve revising the Select function so that it returns a random Position from a Neighborhood object. The following class declarations reflect these changes (highlighted in bold). Assume that the environ.h file has been included in the nbrhood.h file.

```
class Neighborhood
{
    public:

        //constructors
        Neighborhood();
        // postcondition: Size() == 0
        Neighborhood(const Environment & env, const Position & pos);
        // postcondition: this Neighborhood contains the empty positions
        // surrounding pos;
        // Size() == number of empty positions in
        // this Neighborhood

        int Size() const; // # Positions
        // postcondition: returns # Positions in the neighborhood

        Position Select() const; // access a Position
        // precondition: 0 < Size()
        // postcondition: returns a random Position from this Neighborhood
}
```

```

    apstring ToString() const;    // stringized representation
    // postcondition: returns a string version of all Positions
    //                in Neighborhood

// modifying functions

void Add(const Position & pos); // add pos to Neighborhood
    // precondition: there is room in the Neighborhood
    // postcondition: pos added to Neighborhood

private:

    void AddIfEmpty(const Environment & env, const Position & pos);
        // postcondition: pos is added to this Neighborhood
        //                if pos is in env and is empty

    apvector<Position> myList;
    int myCount;
};

class Fish
{
public:
    Fish();
    Fish(int id, const Position & pos);

    int Id() const;
    Position Location() const;
    apstring ToString() const;
    bool IsUndefined() const;

    char ShowMe() const; // makes me visible as a char

    void Move(Environment & e);

private:

    // helper functions have been removed

    int myId;
    Position myPos;
    bool amIDefined;
};

```

- (a) Write a constructor for the `Neighborhood` class that fills the `Neighborhood` object with empty neighbors around `Position pos` in the four directions described in the text (North, South, East, and West). The `Neighborhood` constructor is passed an `Environment` and a `Position`.

Complete the constructor for `Neighborhood` below.

```
Neighborhood::Neighborhood(const Environment & env, const Position & pos)
// postcondition: this Neighborhood contains the empty positions
//                surrounding pos;
//                Size() == number of empty positions in
//                this Neighborhood
```

- (b) Write a revised version of `Neighborhood` member function `Select`, as started below. `Select` should return a random `Position` from this `Neighborhood`.

In writing `Select`, you may assume that the `randgen.h` file has been included in the `nrhood.cpp` file.

Complete function `Select` below.

```
Position Neighborhood::Select() const
// precondition: 0 < Size()
// postcondition: returns a random Position from this Neighborhood
```

- (c) Write a revised version of the `Fish` member function `Move`, as started below. `Move` should define a `Neighborhood` object with the two parameter constructor that fills the `Neighborhood` with potential empty neighboring positions. If the `Neighborhood` contains a potential neighbor, the revised `Select` function should be called to select a `Position`, then the `Environment` should be updated to reflect the move. No debugging statements need to be included in your answer.

In writing function `Move`, you may call any public member function of the `Neighborhood` class, including the `Neighborhood` constructor of part (a) and `Select` of part (b). Assume that the `Neighborhood` constructor and `Select` work as specified, regardless of what you wrote in parts (b) and (c).

Complete function `Move` below.

```
void Fish::Move(Environment & env)
// precondition: Fish stored in env at Location()
// postcondition: Fish has moved to a new location in env
//                (if possible)
```

Appendix C

Solutions for Sample Test Questions

Multiple-Choice

1. (E) none

Rationale:

The implementation of the internal representation of a position within the Position class affects the implementation of the Position member functions but does not change the interface to the Position class from other classes in programs.

2. (D) The simulation will produce the same results as the original only when the fish hits either end of the tank with its last move.

Rationale:

It is not necessary to check for a bump at the beginning of a move, since the fish starts in the middle of the tank. By checking at the end of a move, every bump including one at the final position, will be counted. The proposed change loses a bump in the final position.

3. (D) 4

Rationale:

This problem simply requires some detailed analysis. Below the four possible sequences of moves are shown.

<u>starting position</u>	<u>after one move</u>	<u>after two moves</u>
A B C	A C -	A B C
D - -	D B -	D - -
	A C -	A - C
	D B -	D - B
	A - -	D A -
	D B C	B C -
	A - -	D A C
	D B C	B - -

Two variations on this “black-box testing” question would include the following:

Give a starting configuration and ask which of five given configurations could be reached after one (or two) steps of the simulation.

Give a starting configuration and a configuration reached after one (or two) steps of the simulation and ask which order of movement of fish could be used (e.g. top-to-bottom, left-to-right within each row, or left-to-right, bottom-to-top within each column, etc.)

4. (D) Modifying the program to be able to construct different kinds of neighborhoods (for example, a neighborhood of non-empty neighbors) in Implementation I would require the addition of new Neighborhood member functions.

Rationale:

Implementation I is more flexible than implementation II because the Fish class or some other class could use the Add function of the Neighborhood class to put different positions into a neighborhood, for example neighboring positions that contain a different species of fish in an extended version of the program.

Implementation II would encapsulate the idea of empty neighbors of a position entirely within the Neighborhood class, but thereby prevents the use of the Neighborhood class for other types of neighbors. This illustrates the conflict between encapsulation and flexibility inherent in object-based design.

5. (C) III only

Rationale:

The body of the outer loop should execute one trial of a simulation and record the results. Since each trial of a simulation involves the fish starting at the center position in the tank and a count of the bumps, these must be properly accounted for. The best way is to reset both to their proper initial values, $\text{tankSize}/2$ for the fish position and zero for the number of bumps.

Method I is an attempt to reset the fish position directly. It will not compile because a client program cannot directly access private data members for a class. There is no Fish member function that resets the position; only the constructor sets the value of myPosition. In addition, the result would still be in error because the count of bumps would not be reset to 0.

Method II is an attempt to correctly count the bumps. If the fish is not reset for each iteration of the outer loop, then BumpCount will continue to count the total of all bumps for the ten trials. The line `tot = tot + BumpCount()` is incorrect because it counts many bumps repeatedly. By using BumpCount() in the calculation of the average, the number of bumps is the total for all trials. Unfortunately, this does not reset the position of the fish for each trial. For each iteration of the outer loop after the first, the fish starts that trial at the position it reached in the previous trial, so these are not independent trials of the experiment described.

Method III is correct and demonstrates a good design principle for the problem described. To run a set of trials of a simulation, each independent with the same starting conditions, it is best to encapsulate the process of one trial in a function, which returns or records the results for that trial. Then the program can call that function for each trial, and accumulate the results. The same effect could be obtained from the original program by either moving line 1 to be inside the outer loop immediately before line 4, or by placing a new statement immediately before line 4:

```
fish = Fish(tankSize);
```

However, the encapsulation of one trial in a function is a better design because it is less prone to errors.

Free-Response (Neighborhood modification)

The fish.h file is changed by removing the two functions EmptyNeighbors and AddIfEmpty. The nbrhood.h file is changed as shown in the question.

Here is the implementation of the new Neighborhood private member function AddIfEmpty which may be used in answering the question. Note also that although the question does not refer to it, the Neighborhood member function Add could be made private.

```
void Neighborhood::AddIfEmpty(const Environment & env,
                              const Position & pos)
{
    if (env.IsEmpty(pos))
    {
        Add(pos);
    }
}
```

Solution for part (a).

```
Neighborhood::Neighborhood(const Environment & env,
                            const Position & pos)
    : myList(4),
      myCount(0)
// postcondition: Neighborhood contains all empty neighbors of pos in env
{
    AddIfEmpty(env, pos.North());
    AddIfEmpty(env, pos.South());
    AddIfEmpty(env, pos.East());
    AddIfEmpty(env, pos.West());
}
```

Solution to part (b).

```
Position Neighborhood::Select() const
// precondition: 0 <= index < Size()
// postcondition: returns a random Position in this Neighborhood
{
    RandGen randomVals;

    int index = randomVals.RandInt(myCount);
    DebugPrint(5, "Selecting neighborhood element # "
               + IntToString(index));
    return myList[index];
}
```

Alternate solution to part (b) that does not include the debugging

```
Position Neighborhood::Select() const
// precondition: 0 <= index < Size()
// postcondition: returns a random Position in this Neighborhood
{
    RandGen randomVals;

    return myList[randomVals.RandInt(myCount)];
}
```

Solution to part (c).

```
void Fish::Move(Environment & env)
// precondition: Fish stored in env at Location()
// postcondition: Fish has moved to a new location in env (if possible)
{
    Neighborhood nbrs(env, myPos);
    DebugPrint(3, nbrs.ToString());

    if (nbrs.Size() > 0)
    {
        Position oldPos = myPos;
        myPos = nbrs.Select();
        DebugPrint(1, "Fish at" + oldPos.ToString() + " moves to "
                  + myPos.ToString());
        env.Update(oldPos, *this); // *this means this fish
    }
    else
    {
        DebugPrint(1, "Fish " + ToString() + " can't move.");
    }
}
```

College Board Regional Offices

National Office

Lee Jones, Executive Director, AP Program
Philip Arbolino, AP Secondary School Services
Robert DiYanni, AP International Affairs and Program Publications
Michael Johaneck, AP Teacher Professional Development
Neela Nauth, AP Customer Service
Trevor Packer, AP Program Operations
Mondy Raibon, Pre-AP Initiatives
Frederick Wright, AP Equity and Access Initiatives
45 Columbus Avenue, New York, NY 10023-6992
(212) 713-8066
E-mail: ap@collegeboard.org

Middle States

Mary Alice McCullough/Michael Marsh
3440 Market Street, Suite 410, Philadelphia, PA 19104-3338
(215) 387-7600

Midwestern

Bob McDonough/Paula Herron/Ann Winship
1560 Sherman Avenue, Suite 1001, Evanston, IL 60201-4805
(847) 866-1700

New England

Fred Wetzell
470 Totten Pond Road, Waltham, MA 02451-1982
(781) 890-9150

Southern

Bill Heron/Letishia Jones
100 Crescent Centre Parkway, Suite 340, Tucker, GA 30084-7039
(770) 908-9737

Southwestern

Scott Kampmeier/Paul Sanders
4330 South MoPac Expressway, Suite 200, Austin, TX 78735-6734
(512) 891-8400

Dallas/Fort Worth Metroplex AP Office

Kay Wilson
Box 19666, 600 South West Street, Room 108, Arlington, TX 76019
(817) 272-7200

Western

Claire Pelton/Gail Chapman
2099 Gateway Place, Suite 480, San Jose, CA 95110-1048
(408) 452-1400

AP Director in Canada

George Ewonus
1708 Dolphin Avenue, Suite 406, Kelowna, BC, Canada V1Y 9S4
(250) 861-9050
E-mail: gewonus@ap.ca

Staff at U.S. College Board Regional Offices other than the National Office can be reached via e-mail using their first initial and last name@collegeboard.org



1999-2000

AP Computer Science Development Committee



Susan Rodger, <i>Chair</i>	Duke University Durham, North Carolina
Alyce Brady	Kalamazoo College Kalamazoo, Michigan
Don Kirkwood	North Salem High School Salem, Oregon
Joseph W. Kmoch	Washington High School Milwaukee, Wisconsin
Kathleen A. Larson	Kingston High School Kingston, New York
Mark A. Weiss	Florida International University Miami, Florida
<i>Chief Faculty Consultant:</i> Chris Nevison	Colgate University Hamilton, New York
<i>ETS Consultants:</i> Frances E. Hunt, Esther Tesar	
<i>College Board Consultant:</i> Gail L. Chapman	