

# Appendix B

## Source Code for Visible Classes

This appendix contains implementations of the visible core classes covered in Chapters 1 – 4: `Simulation`, `Fish`, `DarterFish`, and `SlowFish`. Information about `Environment`, which is black box, can be found in Appendix C.

### Simulation.java

```
/**
 * Marine Biology Simulation:
 * A Simulation object controls a simulation of fish
 * movement in an Environment.
 *
 * @version 1 July 2002
 */

public class Simulation
{
    // Instance Variables: Encapsulated data for each simulation object
    private Environment theEnv;
    private EnvDisplay theDisplay;

    /** Constructs a Simulation object for a particular environment.
     * @param env the environment on which the simulation will run
     * @param display an object that knows how to display the environment
     */
    public Simulation(Environment env, EnvDisplay display)
    {
        theEnv = env;
        theDisplay = display;

        // Display the initial state of the simulation.
        theDisplay.showEnv();
        Debug.println("—— Initial Configuration ——");
        Debug.println(theEnv.toString());
        Debug.println("—————");
    }

    /** Runs through a single step of this simulation. */
    public void step()
    {
        // Get all the fish in the environment and ask each
        // one to perform the actions it does in a timestep.
        Locatable[] theFishes = theEnv.allObjects();
        for ( int index = 0; index < theFishes.length; index++ )
        {
            ((Fish)theFishes[index]).act();
        }

        // Display the state of the simulation after this timestep.
        theDisplay.showEnv();
        Debug.println(theEnv.toString());
        Debug.println("—— End of Timestep ——");
    }
}
```

## Fish.java (includes breeding and dying modifications from Chapter 3)

```
import java.awt.Color;
import java.util.ArrayList;
import java.util.Random;

/**
 * Marine Biology Simulation:
 * A Fish object represents a fish in the Marine Biology
 * Simulation. Each fish has a unique ID, which remains constant
 * throughout its life. A fish also maintains information about its
 * location and direction in the environment.
 *
 * Modification History:
 * - Modified to support a dynamic population in the environment:
 *   fish can now breed and die.
 *
 * @version 1 July 2002
 */

public class Fish implements Locatable
{
    // Class Variable: Shared among ALL fish
    private static int nextAvailableID = 1;    // next avail unique identifier

    // Instance Variables: Encapsulated data for EACH fish
    private Environment theEnv;                // environment in which the fish lives
    private int myId;                          // unique ID for this fish
    private Location myLoc;                    // fish's location
    private Direction myDir;                  // fish's direction
    private Color myColor;                    // fish's color
    // THE FOLLOWING TWO INSTANCE VARIABLES ARE NEW IN CHAPTER 3 !!!
    private double probOfBreeding;            // defines likelihood in each timestep
    private double probOfDying;              // defines likelihood in each timestep

    // constructors and related helper methods

    /** Constructs a fish at the specified location in a given environment.
     * The Fish is assigned a random direction and random color.
     * (Precondition: parameters are non-null; loc is valid
     * for env.)
     * @param env    environment in which fish will live
     * @param loc    location of the new fish in env
     */
    public Fish(Environment env, Location loc)
    {
        initialize(env, loc, env.randomDirection(), randomColor());
    }
}
```

```

/** Constructs a fish at the specified location and direction in a
 * given environment. The Fish is assigned a random color.
 * (Precondition: parameters are non-null; loc is valid
 * for env.)
 * @param env environment in which fish will live
 * @param loc location of the new fish in env
 * @param dir direction the new fish is facing
 **/
public Fish(Environment env, Location loc, Direction dir)
{
    initialize(env, loc, dir, randomColor());
}

/** Constructs a fish of the specified color at the specified location
 * and direction.
 * (Precondition: parameters are non-null; loc is valid
 * for env.)
 * @param env environment in which fish will live
 * @param loc location of the new fish in env
 * @param dir direction the new fish is facing
 * @param col color of the new fish
 **/
public Fish(Environment env, Location loc, Direction dir, Color col)
{
    initialize(env, loc, dir, col);
}

/** Initializes the state of this fish.
 * (Precondition: parameters are non-null; loc is valid
 * for env.)
 * @param env environment in which this fish will live
 * @param loc location of this fish in env
 * @param dir direction this fish is facing
 * @param col color of this fish
 **/
private void initialize(Environment env, Location loc, Direction dir,
                        Color col)
{
    theEnv = env;
    myId = nextAvailableID;
    nextAvailableID++;
    myLoc = loc;
    myDir = dir;
    myColor = col;
    theEnv.add(this);

    // object is at location myLoc in environment

    // THE FOLLOWING INITIALIZATIONS ARE NEW IN CHAPTER 3 !!!
    // For now, every fish is equally likely to breed or die in any given
    // timestep, although this could be individualized for each fish.
    probOfBreeding = 1.0/7.0; // 1 in 7 chance in each timestep
    probOfDying = 1.0/5.0; // 1 in 5 chance in each timestep
}

```

```

/** Generates a random color.
 * @return      the new random color
 **/
protected Color randomColor()
{
    // There are 256 possibilities for the red, green, and blue attributes
    // of a color. Generate random values for each color attribute.
    Random randNumGen = RandNumGenerator.getInstance();
    return new Color(randNumGen.nextInt(256),    // amount of red
                    randNumGen.nextInt(256),    // amount of green
                    randNumGen.nextInt(256));    // amount of blue
}

```

**// accessor methods**

```

/** Returns this fish's ID.
 * @return      the unique ID for this fish
 **/

```

```

public int id()
{
    return myId;
}

```

```

/** Returns this fish's environment.
 * @return      the environment in which this fish lives
 **/

```

```

public Environment environment()
{
    return theEnv;
}

```

```

/** Returns this fish's color.
 * @return      the color of this fish
 **/

```

```

public Color color()
{
    return myColor;
}

```

```

/** Returns this fish's location.
 * @return      the location of this fish in the environment
 **/

```

```

public Location location()
{
    return myLoc;
}

```

```

/** Returns this fish's direction.
 * @return      the direction in which this fish is facing
 **/

```

```

public Direction direction()
{
    return myDir;
}

```

```

/** Checks whether this fish is in an environment.
 * @return true if the fish is in the environment
 *         (and at the correct location); false otherwise
 */
public boolean isInEnv()
{
    return environment().objectAt(location()) == this;
}

/** Returns a string representing key information about this fish.
 * @return a string indicating the fish's ID, location, and direction
 */
public String toString()
{
    return id() + location().toString() + direction().toString();
}

// modifier method

// THE FOLLOWING METHOD IS MODIFIED FOR CHAPTER 3 !!!
// (was originally a check for aliveness and a simple call to move)
/** Acts for one step in the simulation.
 */
public void act()
{
    // Make sure fish is alive and well in the environment - fish
    // that have been removed from the environment shouldn't act.
    if ( ! isInEnv() )
        return;

    // Try to breed.
    if ( ! breed() )
        // Did not breed, so try to move.
        move();

    // Determine whether this fish will die in this timestep.
    Random randNumGen = RandNumGenerator.getInstance();
    if ( randNumGen.nextDouble() < probOfDying )
        die();
}

```

```

// internal helper methods

// THE FOLLOWING METHOD IS NEW FOR CHAPTER 3 !!!
/** Attempts to breed into neighboring locations.
 * @return true if fish successfully breeds;
 *         false otherwise
 */
protected boolean breed()
{
    // Determine whether this fish will try to breed in this
    // timestep. If not, return immediately.
    Random randNumGen = RandNumGenerator.getInstance();
    if ( randNumGen.nextDouble() >= probOfBreeding )
        return false;

    // Get list of neighboring empty locations.
    ArrayList emptyNbrs = emptyNeighbors();
    Debug.print("Fish " + toString() + " attempting to breed. ");
    Debug.println("Has neighboring locations: " + emptyNbrs.toString());

    // If there is nowhere to breed, then we're done.
    if ( emptyNbrs.size() == 0 )
    {
        Debug.println(" Did not breed.");
        return false;
    }

    // Breed to all of the empty neighboring locations.
    for ( int index = 0; index < emptyNbrs.size(); index++ )
    {
        Location loc = (Location) emptyNbrs.get(index);
        generateChild(loc);
    }

    return true;
}

// THE FOLLOWING METHOD IS NEW FOR CHAPTER 3 !!!
/** Creates a new fish with the color of its parent.
 * @param loc location of the new fish
 */
protected void generateChild(Location loc)
{
    // Create new fish, which adds itself to the environment.
    Fish child = new Fish(environment(), loc,
        environment().randomDirection(), color());
    Debug.println(" New Fish created: " + child.toString());
}

```

```

/** Moves this fish in its environment.
**/
protected void move()
{
    // Find a location to move to.
    Debug.print("Fish " + toString() + " attempting to move. ");
    Location nextLoc = nextLocation();

    // If the next location is different, move there.
    if ( ! nextLoc.equals(location() )
    {
        // Move to new location.
        Location oldLoc = location();
        changeLocation(nextLoc);

        // Update direction in case fish had to turn to move.
        Direction newDir = environment().getDirection(oldLoc, nextLoc);
        changeDirection(newDir);
        Debug.println(" Moves to " + location() + direction());
    }
    else
        Debug.println(" Does not move.");
}

/** Finds this fish's next location.
* A fish may move to any empty adjacent locations except the one
* behind it (fish do not move backwards). If this fish cannot
* move, nextLocation returns its current location.
* @return the next location for this fish
**/
protected Location nextLocation()
{
    // Get list of neighboring empty locations.
    ArrayList emptyNbrs = emptyNeighbors();

    // Remove the location behind, since fish do not move backwards.
    Direction oppositeDir = direction().reverse();
    Location locationBehind = environment().getNeighbor(location(),
                                                         oppositeDir);
    emptyNbrs.remove(locationBehind);
    Debug.print("Possible new locations are: " + emptyNbrs.toString());

    // If there are no valid empty neighboring locations, then we're done.
    if ( emptyNbrs.size() == 0 )
        return location();

    // Return a randomly chosen neighboring empty location.
    Random randNumGen = RandNumGenerator.getInstance();
    int randNum = randNumGen.nextInt(emptyNbrs.size());
    return (Location) emptyNbrs.get(randNum);
}

```

```

/** Finds empty locations adjacent to this fish.
 * @return    an ArrayList containing neighboring empty locations
 **/
protected ArrayList emptyNeighbors()
{
    // Get all the neighbors of this fish, empty or not.
    ArrayList nbrs = environment().neighborsOf(location());

    // Figure out which neighbors are empty and add those to a new list.
    ArrayList emptyNbrs = new ArrayList();
    for ( int index = 0; index < nbrs.size(); index++ )
    {
        Location loc = (Location) nbrs.get(index);
        if ( environment().isEmpty(loc) )
            emptyNbrs.add(loc);
    }

    return emptyNbrs;
}

/** Modifies this fish's location and notifies the environment.
 * @param newLoc    new location value
 **/
protected void changeLocation(Location newLoc)
{
    // Change location and notify the environment.
    Location oldLoc = location();
    myLoc = newLoc;
    environment().recordMove(this, oldLoc);

    // object is again at location myLoc in environment
}

/** Modifies this fish's direction.
 * @param newDir    new direction value
 **/
protected void changeDirection(Direction newDir)
{
    // Change direction.
    myDir = newDir;
}

// THE FOLLOWING METHOD IS NEW FOR CHAPTER 3 !!!
/** Removes this fish from the environment.
 **/
protected void die()
{
    Debug.println(toString() + " about to die.");
    environment().remove(this);
}
}

```

## DarterFish.java

```
import java.awt.Color;

/**
 * Marine Biology Simulation:
 * The DarterFish class represents a fish in the Marine
 * Biology Simulation that darts forward two spaces if it can, moves
 * forward one space if it can't move two, and reverses direction
 * (without moving) if it cannot move forward. It can only "see" an
 * empty location two cells away if the cell in between is empty also.
 * In other words, if both the cell in front of the darter and the cell
 * in front of that cell are empty, the darter fish will move forward
 * two spaces. If only the cell in front of the darter is empty, it
 * will move there. If neither forward cell is empty, the fish will turn
 * around, changing its direction but not its location.
 *
 * DarterFish objects inherit instance variables and much
 * of their behavior from the Fish class.
 *
 * @version 1 July 2002
 */

public class DarterFish extends Fish
{
    // constructors

    /** Constructs a darter fish at the specified location in a
     * given environment. This darter is colored yellow.
     * (Precondition: parameters are non-null; loc is valid
     * for env.)
     * @param env environment in which fish will live
     * @param loc location of the new fish in env
     */
    public DarterFish(Environment env, Location loc)
    {
        // Construct and initialize the attributes inherited from Fish.
        super(env, loc, env.randomDirection(), Color.yellow);
    }

    /** Constructs a darter fish at the specified location and direction in a
     * given environment. This darter is colored yellow.
     * (Precondition: parameters are non-null; loc is valid
     * for env.)
     * @param env environment in which fish will live
     * @param loc location of the new fish in env
     * @param dir direction the new fish is facing
     */
    public DarterFish(Environment env, Location loc, Direction dir)
    {
        // Construct and initialize the attributes inherited from Fish.
        super(env, loc, dir, Color.yellow);
    }
}
```

```

/** Constructs a darter fish of the specified color at the specified
 * location and direction.
 * (Precondition: parameters are non-null; loc is valid
 * for env.)
 * @param env    environment in which fish will live
 * @param loc    location of the new fish in env
 * @param dir    direction the new fish is facing
 * @param col    color of the new fish
 **/
public DarterFish(Environment env, Location loc, Direction dir, Color col)
{
    // Construct and initialize the attributes inherited from Fish.
    super(env, loc, dir, col);
}

// redefined methods

/** Creates a new darter fish.
 * @param loc    location of the new fish
 **/
protected void generateChild(Location loc)
{
    // Create new fish, which adds itself to the environment.
    DarterFish child = new DarterFish(environment(), loc,
                                     environment().randomDirection(),
                                     color());
    Debug.println(" New DarterFish created: " + child.toString());
}

/** Moves this fish in its environment.
 * A darter fish darts forward (as specified in nextLocation) if
 * possible, or reverses direction (without moving) if it cannot move
 * forward.
 **/
protected void move()
{
    // Find a location to move to.
    Debug.print("DarterFish " + toString() + " attempting to move. ");
    Location nextLoc = nextLocation();

    // If the next location is different, move there.
    if ( ! nextLoc.equals(location()) )
    {
        changeLocation(nextLoc);
        Debug.println(" Moves to " + location());
    }
    else
    {
        // Otherwise, reverse direction.
        changeDirection(direction().reverse());
        Debug.println(" Now facing " + direction());
    }
}

```

```

/** Finds this fish's next location.
 * A darter fish darts forward two spaces if it can, otherwise it
 * tries to move forward one space. A darter fish can only move
 * to empty locations, and it can only move two spaces forward if
 * the intervening space is empty. If the darter fish cannot move
 * forward, nextLocation returns the fish's current
 * location.
 * @return the next location for this fish
 */
protected Location nextLocation()
{
    Environment env = environment();
    Location oneInFront = env.getNeighbor(location(), direction());
    Location twoInFront = env.getNeighbor(oneInFront, direction());
    Debug.println(" Location in front is empty? " +
        env.isEmpty(oneInFront));
    Debug.println(" Location in front of that is empty? " +
        env.isEmpty(twoInFront));
    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.
    Debug.println(" Darter is blocked.");
    return location();
}
}

```

## SlowFish.java

```
import java.awt.Color;
import java.util.ArrayList;
import java.util.Random;

/**
 * Marine Biology Simulation:
 * The SlowFish class represents a fish in the Marine Biology
 * Simulation that moves very slowly. It 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 beyond its
 * own cell, its movement behavior is the same as for objects of the
 * Fish class.
 *
 * SlowFish objects inherit instance variables and much of
 * their behavior from the Fish class.
 *
 * @version 1 July 2002
 */

public class SlowFish extends Fish
{
    // Instance Variables: Encapsulated data for EACH slow fish
    private double probOfMoving; // defines likelihood in each timestep

    // constructors

    /** Constructs a slow fish at the specified location in a
     * given environment. This slow fish is colored red.
     * (Precondition: parameters are non-null; loc is valid
     * for env.)
     * @param env environment in which fish will live
     * @param loc location of the new fish in env
     */
    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
    }
}
```

```

/** Constructs a slow fish at the specified location and direction in a
 * given environment. This slow fish is colored red.
 * (Precondition: parameters are non-null; loc is valid
 * for env.)
 * @param env    environment in which fish will live
 * @param loc    location of the new fish in env
 * @param dir    direction the new fish is facing
 */

```

```

public SlowFish(Environment env, Location loc, Direction dir)

```

```

{
    // Construct and initialize the attributes inherited from Fish.
    super(env, loc, dir, 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
}

```

```

/** Constructs a slow fish of the specified color at the specified
 * location and direction.
 * (Precondition: parameters are non-null; loc is valid
 * for env.)
 * @param env    environment in which fish will live
 * @param loc    location of the new fish in env
 * @param dir    direction the new fish is facing
 * @param col    color of the new fish
 */

```

```

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

```

```

{
    // Construct and initialize the attributes inherited from Fish.
    super(env, loc, dir, col);

    // 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
}

```

```

// redefined methods

```

```

/** Creates a new slow fish.
 * @param loc    location of the new fish
 */

```

```

protected void generateChild(Location loc)

```

```

{
    // Create new fish, which adds itself to the environment.
    SlowFish child = new SlowFish(environment(), loc,
                                   environment().randomDirection(),
                                   color());
    Debug.println(" New SlowFish created: " + child.toString());
}

```

```

/** Finds this fish's next location. A slow fish moves so
 * slowly that it may not move out of its current cell in
 * the environment.
 **/
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
    {
        Debug.println("SlowFish " + toString() +
            " not attempting to move.");
        return location();
    }
}
}
}

```