JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2010**

Vol. 9, No. 1, January – February 2010

EDUCATOR'S CORNER

Darwin's World Simulation in C#: An Interpreter

Richard Wiener

1 INTRODUCTION

Nick Parlante, a Lecturer at Stanford University, invented a pedagogical game entitled "Darwin's World". The original specification (for Pascal) was published and presented at SIGCSE in 1999. Many variations have been published more recently.

http://nifty.stanford.edu/darwin/

http://users.dickinson.edu/~braught/NSFIntegrating/course/labs/darwin/handout.html

In Darwin's world the user creates robot-like graphical creatures with behavior defined by a simple programming language. These creatures migrate around a small two-dimensional grid, each according to its simple program created by the user. The GUI application that controls and displays the location of these creatures must interpret the program instructions supplied in a simple text file for each creature type in real-time and display the movement and behavior of these creatures. Since some creatures may "infect" another creature in an adjacent grid location, converting the infected creature into the same type as the creature administering the infection, the population of the various creature types changes over time although the total number of creatures remains constant. The application tracks and displays the population dynamics as the simulation evolves. New creature types may be created by supplying a text file that contains the "program" that the application interprets. This necessitates changes to the GUI that displays the dynamics of the simulation.

This application has become popular among computer science educators because the application is fun to observe, fun for many students to create and provides a rich assortment of concepts that are useful in computer science and software development.

This application is being presented in a more advanced programming course that features the effective use of the C# programming language and the .NET framework.

As a quick example of how to "program" a creature (we call this creature **Trap** and its file is "Trap.txt") consider the instructions given below:

```
ifenemy 3
left
go 0
infect
go 0
.
Trap
```

What does this user supplied program mean?

The first line of code is line 0. Each creature has a defined direction of movement that is either north, east, south or west. An "enemy" is any creature whose type is different than the given creature's type. The first instruction (on line 0) stipulates that if the Trap creature is facing an enemy in the cell location one away from it in the direction that the Trap creature is pointing then transfer control to the instruction in line 3, the "infect" instruction. This "infect" instruction causes the enemy creature to become a Trap creature. Its color, and symbol change (for display purposes) but most importantly it acquires the program given in the "Trap.txt" file so that its future behavior is that of a Trap creature. If the instruction on line 0 is false (the adjacent cell is empty, a boundary or another Trap creature), then line 1 is executed. This line causes the Trap creature to rotate left by 90 degrees (e.g. if it was originally pointed north it is now pointed west). The last line of "code" in this simple program is denoted by a dot. So, the "program" for the Trap creature causes the Trap creature to remain in its initial grid location either rotating to the left with each move or infecting an adjacent creature if it is an enemy.

A more complex creature, the **Rover**, is governed by the following program:

```
ifenemy 10
ifwall 5
ifsame 5
hop
go 0
ifrandom 8
left
go 0
right
go 0
infect
go 0
.
Rover
```

The ifwall statement (on line 1) directs program execution to line 5 which contains the instruction ifrandom. This instruction directs control to line 8 with 50 percent probability (if it returns true) or the next line (line 6 containing the left instruction). So with equal chance the rover rotates either left or right if it encounters either a wall or another Rover creature in the adjacent cell that it is pointing to. If the adjacent cell is empty it performs the hop command (it vacates its current position and moves to the adjacent cell position). Like the Trap, the Rover is capable of infecting an enemy. But because it is mobile (always moves in the same direction until it encounters a wall or another Rover), its infection range extends potentially over the entire world (the entire grid).

The simplest creature type is **Food**. Its program is given as:

```
left
go 0
.
Food
```

Food creatures cannot move or infect other creatures. They are essential fixed targets that rotate to the left and can be infected by other creatures.

A more complex, created by the author, is **Android**. Its program is:

```
ifrandom 3
left
go 4
right
ifenemy 11
ifwall 9
ifsame 9
hop
go 0
ifrandom 3
go 1
infect
go 0
.
Android
```

It starts by rotating either to the left or right by 90 degrees. Then its behavior is similar to Rover. So instead of moving in a straight line like the Rover creature, its motion is jagged. Like the Rover and Trap creatures, it too can infect enemy creatures in adjacent grid locations.

Another complex creature, created by the author is the **Hopper**. Its program is:

```
ifenemy 9
ifenemyleft 10
ifenemyright 12
ifwall 14
ifsame 14
hop
go 0
```

```
ifrandom 2
go 4
infect
left
go 0
right
go 0
if random 17
left
go 0
right
go 0
.
Hopper
```

It is somewhat of a hybrid between an Android and a Rover with some capability shared by neither. The Hopper creature, like the Rover, first determines whether it can infect an enemy in the direction that it is facing. If not, it rotates either to the left or to the right if an enemy is to the left or right. Then it hops in the direction it is facing.

The final creature, also created by the author, is the **ChangeDirectionRover**. Its program is:

```
ifenemy 17
ifwall 12
ifsame 12
hop
ifenemy 17
ifwall 12
ifsame 12
hop
ifenemy 18
ifwall 12
ifsame 12
hop
ifrandom 15
left
go 0
right
go 0
infect
go 0
.
ChangeDirectionRover
```

See whether you can figure out by reverse engineering the program what the behavior of the ChangeDirectionRover does (if the name of this creature does not already explain its behavior.

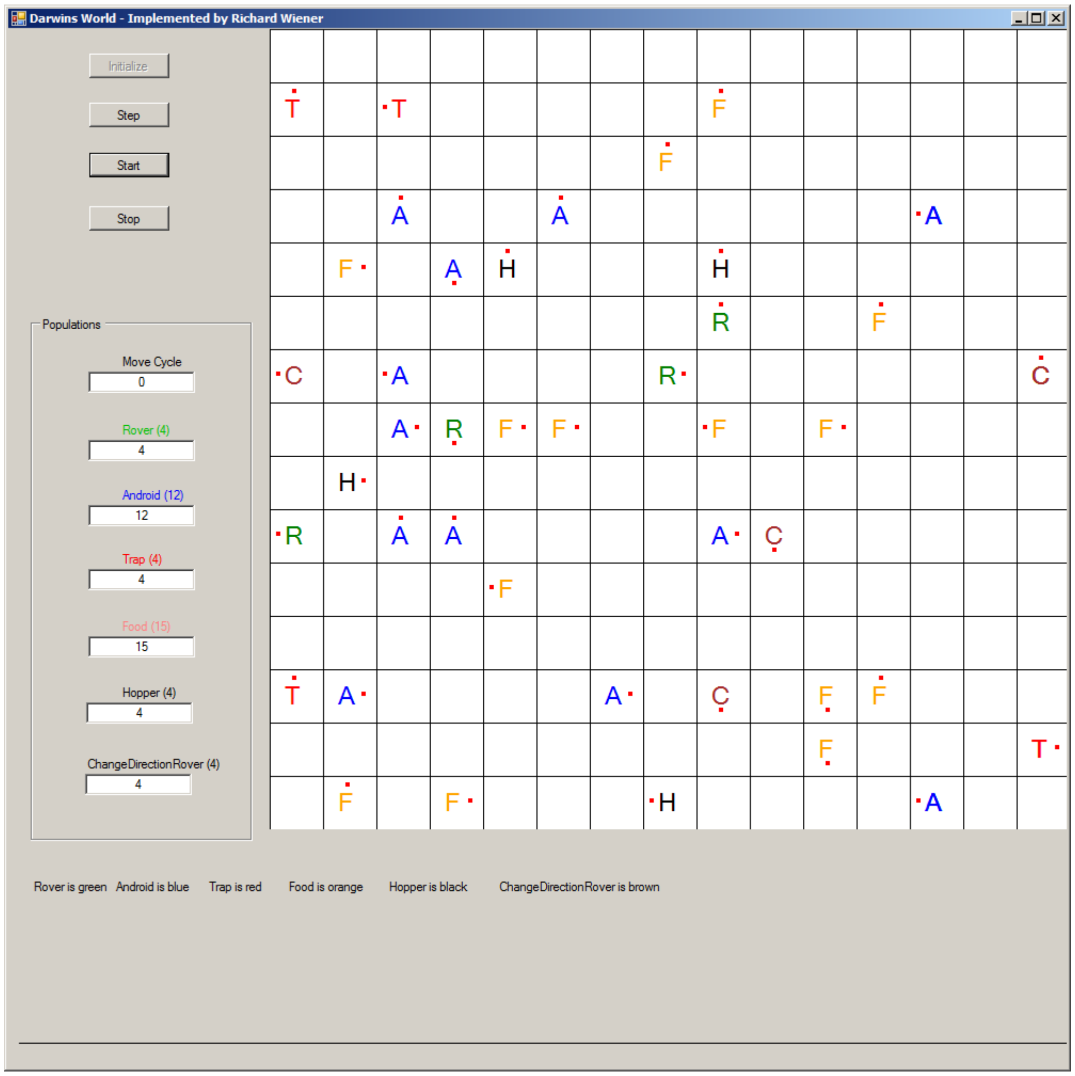
The specifications for the GUI simulation that we are to construct are given below.

2 SPECIFICATIONS

- The GUI specifies the initial number of each creature type that must be placed at random locations in the grid with each creature pointing in a randomly assigned direction.
- Each of the programs for the various creature types must be loaded from simple text files (Android.txt, Rover.text, Food.txt, Trap.txt, Hopper.txt and ChangeRover.txt). If other creature types are created, the GUI class as well as other class modifications must be made. These text files must be in the same sub-directory as the executable.
- The user supplied programs that control each creature type must be interpreted in real time by the GUI application that displays the movement and behavior of each creature as well as the population dynamics.
- A move for a creature terminates if it is instructed to go left, go right, hop or infect.
- The next line of code (the next instruction) for the creature must be saved so that when the creature is told to move again it knows what line of code to use.
- A move cycle is completed when all the creatures inhabiting the grid have moved exactly once.
- The sequence of moves must be shuffled after each move cycle (a new random sequence of moves must be applied to the existing collection of creatures).
- The population dynamics must be shown on the GUI over time (move-cycles).

A screen shot of a game that has just been initialized is shown next.

A screen shot of this game in progress after 16 move cycles is shown below (each game displays different emergent population dynamics – that makes the game fun to observe):



Move Cycle: 0

Rover (4): 4

Android (12): 12

Trap (4): 4

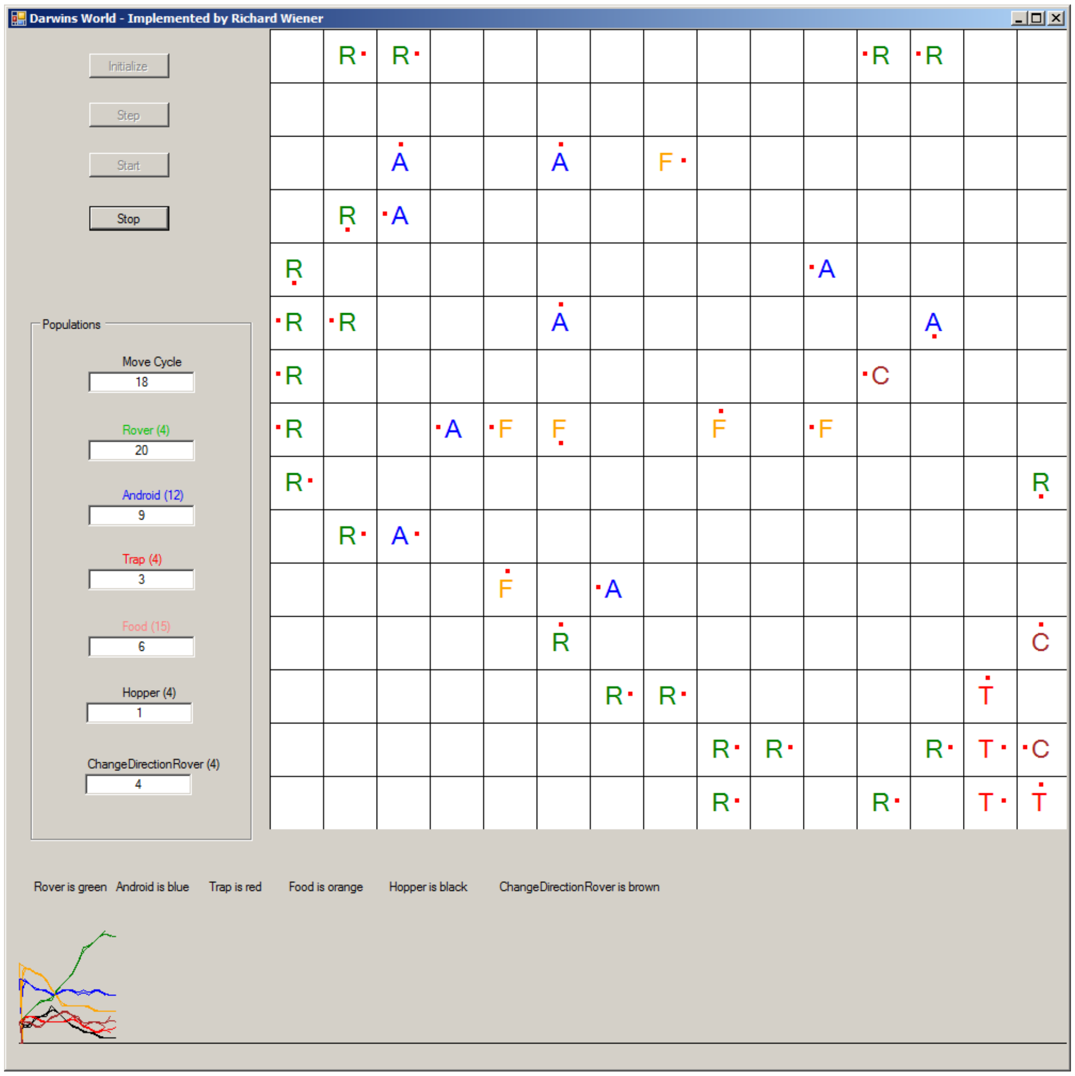
Food (15): 15

Hopper (4): 4

ChangeDirectionRover (4): 4

T		T						F						
							F							
		A			A							A		
	F		A	H				H						
								R			F			
C		A					R							C
		A	R	F	F			F		F				
	H													
R		A	A					A	C					
				F										
T	A					A		C		F	F			
										F				T
	F		F				H					A		

Rover is green Android is blue Trap is red Food is orange Hopper is black ChangeDirectionRover is brown


Darwins World - Implemented by Richard Wiener

Populations

Move Cycle	18
Rover (4)	20
Android (12)	9
Trap (4)	3
Food (15)	6
Hopper (4)	1
ChangeDirectionRover (4)	4

	R	R									R	R		
		A			A		F							
	R	A												
R										A				
R	R				A							A		
R											C			
R			A	F	F			F		F				
R														R
	R	A												
				F		A								
					R									C
						R	R						T	
								R	R			R	T	C
								R			R		T	T

Rover is green Android is blue Trap is red Food is orange Hopper is black ChangeDirectionRover is brown

It is noted that for this run of the simulation, after only 16 move cycles, the population of Hopper creatures is indicated as one.There is no Hopper creature shown when this screen snapshot was taken. That is because population updates are done only at the end of a move cycle but creature updates are done as soon as a creature moves. So upon the completion of move cycle 16, the population of Hopper creatures will be zero.

3 DESIGN AND IMPLEMENTATION

Three enum types, *OpCode*, *Direction* and *CreatureType*, are defined to support the implementation. These are presented below. (Using statements are omitted throughout to save space but are included in the actual implementation).

```
namespace DarwinsWorld {
  public enum OpCode {Hop, Left, Right, Infect, IfEmpty,
                                            fWall,   IfSame,   IfEnemy,
IfRandom, Go}
}

namespace DarwinsWorld {
  public enum Direction {North, East, South, West}
}

namespace DarwinsWorld {
  public enum CreatureType {Android, ChangeDirectionRover,
                Food, Hop, Hopper, Rover, Trap}
}
```

Next a class *Instruction* that encapsulates each line of the user's program code is presented. An instruction object encapsulates an op code and line number.

```
namespace DarwinsWorld {

  public class Instruction {
    // Fields
    private OpCode code;
    private int lineNumber;

    public Instruction(OpCode code, int lineNumber) {
      this.code = code;
      this.lineNumber = lineNumber;
    }

    // Properties
    public OpCode Code {
      get {
        return code;
      }
    }

    public int LineNumber {
      get {
        return lineNumber;
      }
    }
  }
}
```

The next design decision concerns the use of a few global entities that are acessible throughout the application domain. These are defined in class ***Global***.

```
namespace DarwinsWorld {

  public class Global {
    // Global fields
    public static int ROWS = 15;
    public static int COLS = 15;
    public static Dictionary<OpCode, TakeAction> actions =
                                new Dictionary<OpCode, TakeAction>();
    public static Creature[,] creatures =
                                new Creature[Global.ROWS, Global.COLS];

    public static void SetTable(OpCode code, TakeAction action) {
      actions[code] = action;
    }

  }
}
```

A *Dictionary*, ***actions***, that asssociates each *OpCode* with some action, defined as a delegate type ***TakeAction***, is defined. So the "data" that is associated with each OpCode key is in fact a method that represents an instance of the *TakeAction* delegate type. In other words we are associating concrete actions with each *OpCode* key in the dictionary.

The *TakeAction* delegate type is defined as:

```
public delegate bool TakeAction(Point oldPos, Point newPos,
                                                      String
programFileName);
```

The game model is defined as a two-dimensional array of type *Creature*, yet to be defined.

The class ***Creature*** is implemented next. It is tempting to create a hierarchy of concrete classes that are descendents of an abstract *Creature* class. This design defines only one concrete *Creature* class that contains a list of instructions that defines its behavior. The basis of this decision is that creatures differ in their cosmetic representation (their symbol, color and most importantly instruction set). These attributes can best be modeled using a whole-part relationship rather than an inheritance relationship.

Class *Creature* is given next.

```
namespace DarwinsWorld {

  public class Creature {

    // Fields
    private Point position;
    private Direction direction;
    private int nextInstruction;
    private List<Instruction> instructions;
    private String programFileName;
    private Color color;
    private CreatureType type;

    // Constructors
    public Creature(String programFileName,
                              Point position, Direction direction,
                                       Color c, CreatureType type)
{
      this.programFileName = programFileName;
      this.position = position;
      this.direction = direction;
      color = c;
      this.type = type;
      BuildInstructions();
    }

    // Properties
    public CreatureType CreatureType {
      get {
        return type;
      }
      set {
        type = value;
      }
    }

    public String ProgramFileName {
      get {
        return programFileName;
      }
      set {
        programFileName = value;
      }
    }

    public Point Position {
      get {
        return position;
      }
      set {
        position = value;
      }
    }

    public Color Color {
      get {
        return color;
      }
    }

    public Direction Direction {
      get {
        return direction;
      }
    }

    // Commands
    public void Move(World world) {
      Instruction instruction = instructions[nextInstruction];
      Instruction oldInstruction = null;
      do {
        TakeAction action = Global.actions[instruction.Code];
        Point newPosition = position;
        if (instruction.Code == OpCode.Infect ||
            instruction.Code == OpCode.Hop ||
            instruction.Code == OpCode.IfEnemy ||
            instruction.Code == OpCode.IfSame) {
          newPosition = NewPosition();
        } else if (instruction.Code == OpCode.IfEnemyLeft) {
          newPosition = PositionLeft();
        } else if (instruction.Code == OpCode.IfEnemyRight) {
          newPosition = PositionRight();
        }
        if (newPosition.Y < 0 ||
                              newPosition.Y > Global.ROWS - 1 ||
          newPosition.X < 0 ||
                                                newPosition.X > Global.COLS
- 1) {
          newPosition = position;
        }
        Point oldPosition = position;
        bool      result      =      action(oldPosition,      newPosition,
programFileName);
        if       (instruction.Code       ==       OpCode.Infect       ||
                                 instruction.Code == OpCode.Hop ||
                                 instruction.Code == OpCode.Left ||
            instruction.Code == OpCode.Right) {
          world.FireDisplay(
                                                instruction.Code          ==
OpCode.Infect,
                                                oldPosition,   newPosition,
  programFileName[0].ToString(),
                                                color, direction);

          // Update world.List
          if (instruction.Code == OpCode.Infect) {
            // Find the infected creature
            List<Creature> list = world.List;
            int index = 0;
            foreach (Creature creature in list) {
              if (creature.Position.Equals(

  newPosition)) {
                break;
              }
              index++;
            }
            Creature infectedCreature = list[index];
            infectedCreature.color = this.Color;
            infectedCreature.direction = this.Direction;
            infectedCreature.CreatureType                             =
                                       this.CreatureType;
          }

        }
        if (result == true) {
          nextInstruction = instruction.LineNumber;
        } else {
          nextInstruction++;
        }
        oldInstruction = instruction;
        instruction = instructions[nextInstruction];
      }     while     (oldInstruction.Code     !=     OpCode.Left     &&
                                       oldInstruction.Code             !=
OpCode.Right &&
           oldInstruction.Code         !=         OpCode.Infect        &&
                                       oldInstruction.Code             !=
OpCode.Hop);
    }

    public void BuildInstructions() {
      instructions = new List<Instruction>();
      nextInstruction = 0;
      // Read program file for creature
      TextReader reader = new StreamReader(programFileName);
      String line = reader.ReadLine();
      while (!line.Equals(".")) {
        if (line.Contains("hop")) {
          instructions.Add(new Instruction(OpCode.Hop, 0));
        } else if (line.Contains("ifenemyleft")) {
          int                            operand                      =

  Convert.ToInt32(line.Substring(11).Trim());
          instructions.Add(new

  Instruction(OpCode.IfEnemyLeft, operand));
        } else if (line.Contains("ifenemyright")) {
          int                            operand                      =

  Convert.ToInt32(line.Substring(12).Trim());
```

```
            instructions.Add(new
    Instruction(OpCode.IfEnemyRight, operand));
          } else if (line.Contains("go")) {
            int                        operand                         =
    Convert.ToInt32(line.Substring(2).Trim());
            instructions.Add(new                  Instruction(OpCode.Go,
    operand));
          } else if (line.Contains("left")) {
            instructions.Add(new Instruction(OpCode.Left, 0));
          } else if (line.Contains("right")) {
            instructions.Add(new Instruction(OpCode.Right, 0));
          } else if (line.Contains("infect")) {
            instructions.Add(new
                                             Instruction(OpCode.Infect,
  0));
          } else if (line.Contains("ifenemy")) {
            int                        operand                         =
    Convert.ToInt32(line.Substring(7).Trim());
            instructions.Add(new
                                             Instruction(OpCode.IfEnemy,
  operand));
          } else if (line.Contains("ifwall")) {
            int                        operand                         =
    Convert.ToInt32(line.Substring(6).Trim());
            instructions.Add(
                                             new
  Instruction(OpCode.IfWall, operand));
          } else if (line.Contains("ifsame")) {
            int                        operand                         =
    Convert.ToInt32(line.Substring(6).Trim());
            instructions.Add(new              Instruction(OpCode.IfSame,
    operand));
          } else if (line.Contains("ifrandom")) {
            int                        operand                         =
    Convert.ToInt32(line.Substring(8).Trim());
            instructions.Add(
                                             new
  Instruction(OpCode.IfRandom, operand));
          }
          line = reader.ReadLine();
        }
      }

      public void RightDirection() {
        if (direction == Direction.North) {
          direction = Direction.East;
        } else if (direction == Direction.East) {
          direction = Direction.South;
        } else if (direction == Direction.South) {
          direction = Direction.West;
        } else if (direction == Direction.West) {
          direction = Direction.North;
        }
      }

      public void LeftDirection() {
        if (direction == Direction.North) {
          direction = Direction.West;
        } else if (direction == Direction.West) {
          direction = Direction.South;
        } else if (direction == Direction.South) {
          direction = Direction.East;
        } else if (direction == Direction.East) {
          direction = Direction.North;
        }
      }

      // Queries
      public Point NewPosition() {
        switch (direction) {
          case Direction.North:
            return new Point(position.X, position.Y - 1);
          case Direction.East:
            return new Point(position.X + 1, position.Y);
          case Direction.South:
            return new Point(position.X, position.Y + 1);
          case Direction.West:
            return new Point(position.X - 1, position.Y);
        }
        return new Point(0, 0);
      }

      public Point PositionLeft() {
        switch (direction) {
          case Direction.North:
            return new Point(position.X - 1, position.Y);
          case Direction.East:
            return new Point(position.X, position.Y - 1);
          case Direction.South:
            return new Point(position.X + 1, position.Y);
          case Direction.West:
            return new Point(position.X, position.Y + 1);
        }
        return new Point(0, 0);
      }

      public Point PositionRight() {
```

```
          switch (direction) {
            case Direction.North:
              return new Point(position.X + 1, position.Y);
            case Direction.East:
              return new Point(position.X, position.Y + 1);
            case Direction.South:
              return new Point(position.X - 1, position.Y);
            case Direction.West:
              return new Point(position.X, position.Y - 1);
          }
          return new Point(0, 0);
        }
      }
    }
```

Each creature is modeled as containing a *List* containing base-type *Instruction*. The *BuildInstructions* method parses each line of the file supplied by the user and constructs this list.

The *Move* command takes the game model class *World* (yet to be defined) as a parameter. A variable *action* is obtained from the global *Dictionary* called *actions*. The action is invoked using:

```
    bool result = action(oldPosition, newPosition, programFileName);
```

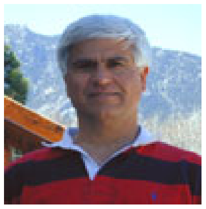
The concrete method that is executed is defined in the *World* class and stored in the globally available *actions*.

The command *FireDisplay* is invoked on the *world* object passed to the creature from the *World* model class. This is how each creature object notifes the GUI (yet to be defined) about its movement.

The *Move* command in class *Creature* is our interpreter. It moves from one instruction to another and associates each instruction with a stored action command in the global dictionary *table*.

In the sequel article, "**Darwin's World Simulation in C#: The Model/View Classes",** to be published in the March/April, 2010 issue of JOT, the details of event handling and link the ***World*** class to the ***WorldUI*** class will be presented and discussed.

About the author



Richard Wiener is Chair of Computer Science at the University of Colorado at Colorado Springs. He is also the Editor-in-Chief of JOT and former Editor-in-Chief of the Journal of Object Oriented Programming. In addition to University work, Dr. Wiener has authored or co-authored 22 books and works actively as a consultant and software contractor whenever the possibility arises. His latest book, published by Thomson, Course Technology in April 2006, is entitled *Modern Software Development Using C#/.NET*.