JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 4, Mai - June 2007

EDUCATOR'S CORNER

A New Software Development Project Using an Old Game

Richard Wiener, Editor-in-Chief, JOT, Chair, Department of Computer Science, University of Colorado at Colorado Springs

1 INTRODUCTION

As many of us who teach more advanced programming courses know, finding software development projects that are engaging to students and that provide sufficient richness to allow for interesting designs and implementations while being solvable within the time-constraints of a semester is challenging. Some games provide a basis for satisfying these requirements since they do not require prior domain expertise. The rules can be specified precisely in contrast to many real-world problems where only fuzzy and changing specifications are the norm.

In the past, I have assigned major semester programming projects based on variations of two games that are presented in my recently published book, "Modern Software Development Using C#/.NET" (Thomson Course Technology, 2007). Both of these games involve computer simulations in which the computer controls all players and there is no human involvement in the strategy.

The first of these games is presented in Chapter 6 of my book. It is a variation on the classic Parker Brothers Monopoly Game, and is called "OOPLY". Ten board positions are defined in OOPLY, each providing different behavior when a player lands on the position. This provides the student the opportunity to focus on the differences and commonality in the behavior of the various board positions in defining the class architecture of the software. After presenting an initial solution in Chapter 6, four additional solutions are presented using re-factoring.

The second game is presented in Chapter 16 of my book. It too does not involve the actions of a human player but only involves a computer simulation. It is an ecological simulation (initially implemented over 20 years ago) involving various predators and prey objects. Like the "OOPLY" simulation, it provides the students with an opportunity to

Cite this column as follows: Richard Wiener: "A New Software Development Project Using an Old Game", in Journal of Object Technology, vol. 6, no. 4, May - June 2007, pp 39 – 53 http://www.jot.fm/issues/issue_2007_05/column5

focus on the similarities and differences in behavior among a rich collection of actors that provides the basis for the class design.

With the publication of these two projects in my book I have needed to look for new projects for my 300-level advanced object-oriented programming course using C#/.NET.

After recalling a wonderful game of strategy that I purchased over twenty years ago in a New York store that sold exotic games and after rummaging through my closet to find this game, I have found such a new project using an old game, "Hair and Tortoise" first published and copyrighted in England in 1974 by Intellect Games. The game has resurfaced more recently under different ownership by David Parlett.

There is a minimum of chance and a maximum of strategy in this wonderful game. It can be played by two players up to six players with four players the norm. I decided in advance that I would allow the computer to control three of the players and require one human player. That would make the finished product more interesting to the programmer since he or she could actually compete against the three computer controlled players.

Like the other two games mentioned above, this game involves many different types of board positions and rules of behavior (64 separate board positions with about a dozen different behaviors) so it provides a rich basis for class design and event-handling in a GUI context. I have implemented three levels of play (Novice, Intermediate and Advanced) in my own implementation. I require my students to implement the Novice strategy that I fully specify. Extra credit is available for students who also implement the intermediate or advanced levels of play (they must find suitable algorithms to accomplish this).

Because I wish to be able to reuse this project in the future, I will not present the details of my design and implementation in this paper but only the specifications, some small snippets of code and some commentary.

2 SPECIFICATIONS OF HARE AND TORTOISE GAME

The following is the specification document provided to my students for this major programming project. It includes the rules of the game, screen shots, snippets of code, as well as general guidance related to my grading of the project.

Background

The board game Hare and Tortoise was first published in London in 1974 by Intellect Games. The inventor of the game presents its history at: http://www.davidparlett.co.uk/haretort/

Here are a few quotes from this website:

"Hare & Tortoise was first published in Britain by Intellect Games UK in 1974 and has been in print ever since, though Intellect has long since vanished from the scene. The first German edition was published by Ravensburger in 1978 and became the first ever

game to win the now prestigious *Spiel des Jahres* award. It has since sold some 2 million units in at least 10 languages, including two known pirated editions. The current edition (top right) is published by Abacus Spiele (AbacusSpiele Verlags KG, Schopenhauerstraße 41, D-63303 Dreieich, Germany) with an English-language equivalent published in America by Rio Grande Games, available in Britain from The Board Game Company, P.O. Box 3633, Newport Pagnell, Bucks, MK16 8ND *Telephone* 01908 611722, *Email* info@boardgamecompany.co.uk (price 19.99 GBP + p&p)."

Hare & Tortoise is a uniquely original race game in that movement is governed by skill rather than chance. Instead of rolling dice to find out how far to move, you can always move forwards as far as you like - but only if you can afford to pay for it. This you do by consuming units of energy called carrots. The 65 carrots you start with are just enough to get you home one square at a time by spending one carrot per move. Alternatively, they are enough to get you up to ten squares forward in a single leap, and you can earn more by carefully choosing which square to land on.

But there's a catch!

The catch is that the further you move in one turn, the faster the cost of moving accelerates. Therefore

- If you play like a hare, taking great leaps forward at each move, you risk running out of carrots so fast that you may lose valuable time trying to get them back. But:
- If you play like a tortoise, plodding along as cheaply as possible, you risk lagging so far behind that you may never have time to catch up with the others.

The skill of the game lies in choosing which square to move to, and deciding whether to play hare-wise or tortoise-wise depending on your position. The fun of the game lies in changing other runner's positions by overtaking them - or even undertaking them by moving backwards.

So, unlike traditional race games, Hare & Tortoise is won by superior strategy and player interaction. The element of chance is not only reduced to a minimum, but can be eliminated altogether by agreeing to avoid landing on Hare squares, which are, by design, the only external chance elements in the game.

The actual cost of moving is: 1 for the first square, plus 2 for the second, plus 3 for the third... and so on. It therefore costs 1 carrot to move 1 square, 3 to move 2, 6 to move 3, 10 to move 4, etc. To generalise, moving forwards **n** squares in one turn costs n(n+1)/2 carrots. So, given 65 carrots to start with, you might play tortoise-wise and get home in 65 moves at 1 carrot each and still have 1 carrot left over.

Playing hare-wise, you could get home in just one move, but only if you could afford the 2080 carrots such a leap would cost. To add to your problems, the further ahead you are, the fewer the carrots you earn when you land on a pay-out square. In Hare & Tortoise, unlike certain other games, you don't collect 200 carrots every time you pass Go."

A photo of the game board from the original Intellect Games is presented in Figure 1 below.

Figure 1 – Photo of actual game board in Hare and Tortoise

A screen shot of the game (implemented by Richard Wiener) is shown in Figure 2 below.

Figure 2 – Screen shot of initial board configuration

Although the shape of the board has been modified to accommodate the limited space available on the screen, if you look closely, you will see that the sequence of board positions is the same as in the original game.

Another screen shot depicts the game after several moves when it is player 3's turn to move. Only the first player is controlled by the human player while players 2, 3 and 4 are controlled by the computer. Each computer move is actuated by clicking the "Computer Move" button. The detailed results of all moves are shown in the "Transactions Log" scrollable *RichText* control.

Figure 3 – Screen Shot of game after several moves with player 3 next to move

You will be supplied with 50 x 50 pixel bitmaps of each distinct image that you will need to place on the board as part of your implementation of this game. I generated these by downloading some images, creating others and using Photoshop to trim the images to the correct size and bitmap format.

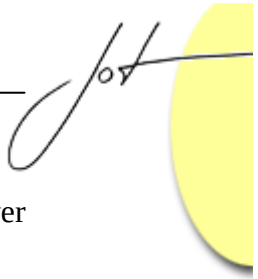
Overview of Hair and Tortoise (4 player game)

Each player starts the game with 65 carrots and 3 lettuces. Carrots are used to fuel moves. Each player must "chew" all of their lettuces before they cross to the finish line. In addition, a player cannot have more than 20 carrots when it crosses the finish line.

Players move in sequence, starting with player 1. When it is a player's turn to move, she may move forward as many squares on the board as she has carrots to pay for the move. A forward move of one board position costs 1 carrot, two board positions cost 3 carrots, three board positions cost 6 carrots, and in general, the cost is given by:

```
int cost = 0;
for (int index = 1; index <= n; index++) {
   costs += index;
}
```

where n is the number of board positions moved.



Clearly, in order to navigate across the 64 board positions to the finish line, a player must acquire carrots during the game.

The only element of chance in the game is when a player moves to a Hair position. This could result in moving ahead of the player directly in front of the given player, moving behind the player just behind the given player, missing a turn, not paying in carrots for the given turn, moving again, etc (exact rules given later).

As indicated earlier, the first player that moves to the finish position with 20 or fewer carrots and no lettuces wins. In the game to be implemented, the human player controls player 1 and the computer controls players 2, 3 and 4. There are three levels of skill associated with the computer. You will be required to implement only the "Novice" level. Extra credit will be awarded for implementing the other two more advanced levels.

Detailed Rules of Hair and Tortoise (4 player game)

When it is a player's turn to move, he may move forward constrained by the number of carrots in his possession. He may also move back only to the tortoise square immediately behind him, if it is unoccupied. If this is elected, the player acquires carrots given by 10 times the number of board positions moved backwards.

- A player CANNOT move forward to a tortoise square.
- A player CANNOT move to a square occupied by another player.
- A player CANNOT move onto a lettuce square if the player has chewed all three of its lettuces.
- A player cannot move backwards except to the tortoise square immediately behind it (if it is unoccupied).

Only the resources of the human player, player 1, are shown as the game progresses.

If a player moves to a lettuce square (must have at least one lettuce to do this) then on her next turn she can elect to move away normally or "chew" a lettuce (nothing else happens on the turn in which she lands on the lettuce square and no move away from the Hair positions occurs when a lettuce if "chewed"). This chewing of lettuce reduces the number of lettuces held by the player (need zero when crossing the finish line). In addition, carrots are awarded to the player based on the player's current rank in the game (leading the race, in second place, in third place and last) given by 10 x rank. So if a player is in third place, he receives 30 carrots on the turn in which he chews the lettuce.

On the next turn, after chewing the lettuce, the player is not allowed to dwell on the lettuce square and is required to vacate the lettuce square. On a later turn, a player is allowed to move to a lettuce square previously used.

If a player moves to a Carrot square she must wait for the next turn (nothing else happens on the turn in which the player lands on the Carrot square). On the next turn, the player can either move away or chew a carrot. Chewing a carrot results in either acquiring 10 more carrots or losing 10 carrots. The player must make this choice. At the beginning and middle of the game it is usually desirable to gain carrots. Near the end of

the game it might be desirable to lose 10 carrots since no more than 20 carrots can be held by the player crossing the finish line.

If a player moves to a numbered square (either “1”, “2”, “3” or “4”), nothing else happens on that turn. On the player’s next turn, if the player’s relative rank corresponds to the numbered square, that player acquires 10 x rank carrots. So, for example, if player 1 moves to square “2” and on the next turn is ranked second in board position, he receives 20 carrots. Of course one of the opposing players may move to block this ranking. In fact a good defensive strategy is to move to alter the ranking of a player that has moved to a numbered position in order to deny that player this acquisition of carrots when it is her turn to move from the numbered square.

If a player moves backwards to the unoccupied Tortoise square directly behind him (cannot skip backwards to another tortoise square), he acquires carrots given by the number of board positions moved backwards multiplied by 10.

Finally, Hair squares. The following table indicates the consequences of moving to a Hair position. A random value from 1 to 6, with equal likelihood, is used to determine the outcome.

Random Integer	**In First Place**	**In Second Place**	**In Third Place**	**In Fourth Place**
1	Miss a turn	Miss a turn	Miss a turn	Miss a turn
2	Move back to nearest unoccupied carrot square	Move back to nearest unoccupied carrot square	Move back to nearest unoccupied carrot square	Move forward to nearest unoccupied carrot square
3	Drop back one position	Drop back one position	Move forward one position	Move forward one position
4	Chew a carrot	Chew a lettuce	Chew a lettuce	Chew a lettuce
5	Last turn free	Last turn free	Last turn free	Last turn free
6	Move again	Move again	Move again	Move again

Miss a turn

Player forfeits a move when it is her next turn to move.

Chew a carrot

Player does not move from Hair square. He decides whether to acquire 10 carrots or lose 10 carrots.

Chew a lettuce

Take the same action as would be taken on a lettuce square including acquiring carrots. If the player has no lettuces, nothing happens.

Drop back one position

Move back to the square immediately behind the player positioned behind the given player if that square is unoccupied. If it is occupied, continue to move back one square until the first unoccupied square is found. No carrots are acquired if this brings the player to a tortoise square.

Move forward one position

Move forward to the square immediately ahead of the player positioned ahead of the given player if that square is unoccupied. . If it is occupied, continue to move forward one square until the first unoccupied square is found. No further carrots are expended in making this forward move.

Move backward to a carrot square

Move backwards to the nearest carrot square behind the given player. If the nearest carrot square is occupied, move backwards to the next carrot square. Continue this until the first unoccupied carrot square is found. If none is found, the player does not move.

Last turn free

Get back the carrots used to move to the Hair square.

Move again

Immediately move again after landing on the Hair square.

3 SPECIFIC DETAILS OF COMPUTER IMPLEMENTATION

Please download the zip file that contains the bitmap images and my executable. Unzip the contents into any sub-directory and launch the application using the executable provided.

You will see that if the mouse hovers over a given square (I implemented each square using a 50 x 50 *Panel* component with a bitmap image embedded in the panel), a message is generated that indicates whether the move to that position is legal. If legal, the number of carrots that is will cost to move to that square are indicated. If the move is backwards to the first unoccupied tortoise square, the number of carrots to be acquired is shown. If the move is illegal, “Illegal Move” is indicated. When the mouse moves off the square, the bitmap image of the square is refreshed.

The presence of a player on a square is indicated by a large white numeral with the player's number. As players move forwards or backwards, the numeral must be removed from the square previously occupied and added to the square that the player has moved to.

When it is Player 1's turn, a move is accomplished by clicking the mouse on the square you wish to move to (assuming it is an allowed square). If the square is not allowed (because it is occupied, a lettuce square when the player has no more lettuces or costs more to move to than the player has carrots), a click of that square causes nothing to happen.

When it is player 2, player 3 or player 4's turn to move, this move is calculated by the computer but actuated by the user by clicking the button "Computer Move". The result of the move is shown and its details chronicled in the "Transactions Log" scrollable *RichText* box. In fact this transactions log provides the entire history of the game for all players. It is most useful to carefully inspect as you are debugging your implementation.

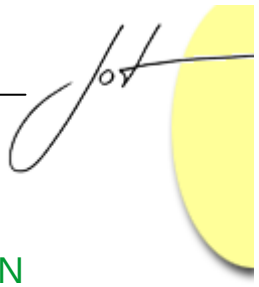
The rules that I used to implement the computer move at the "Novice" level are the following:

- If currently on a lettuce square (with one or more lettuces), chew the lettuce.
- If it is possible to move to the finish square with no lettuces and 20 or fewer carrots (if this is not possible, the finish square is shown as "Illegal Move"), move to the finish square and win the game.
- If it is possible to move forward to a lettuce square (must have one or more lettuces), move forward to a lettuce square.
- If on board position 56 or greater with more than 36 carrots, move to carrot square 59 if it is unoccupied. The goal would be to dispose of 10 carrots on the next turn (to ensure being able to finish with 20 or fewer carrots).
- If none of the above conditions hold, move randomly to one of the legal squares that are possible.

If either the human player (Player 1) or a computer player cannot move on a given turn (all legal squares occupied), that player must simply skip that turn. Your implementation must detect this condition and post this information to the transactions log and update the player move according.

I would advise you to play the game many times using my implementation while carefully observing the transactions log in order to become familiar and comfortable with the domain (the rules).

Please send me email if you have any questions based on this document or based on the actual playing of the game using my implementation.



4 REQUIREMENTS FOR YOUR DESIGN AND IMPLEMENTATION

Although it might be possible for you to come up with a complete implementation in a single large GUI class in which all the business rules are included, this is not acceptable. Such a solution will result in a grade of no more than 70 percent even if such an implementation works and meets the specifications.

Recall that one of the goals of CS 302 is to learn how to use the C# language and the .NET framework effectively. This implies using good techniques of object-oriented architecture. The Model/View approach that we will be spending considerable time discussing and illustrating recommends a sharp separation between the classes that represent the business logic (classes that implement the rules of Hair, Tortoise, Carrot, Lettuce, etc.) and the GUI class that listens to events generated by objects of the business classes and updates its view based on these events.

The conduit that allows a business object (e.g. an object of type *Carrot*) to communicate with the GUI object is using event types that are each associated with a delegate type (Chapter 8). Please collect all the delegate types that you need (there will be several) and put them into a separate file, *Delegates.cs*. As you will learn in Chapter 8, each delegate is a class in its own right so these delegate declarations stand outside of any other class. By placing all within the namespace of the application but in a separate file, it will be easier for you and for me to see what delegates you have chosen to provide the needed communication links between event source classes (many of the model classes) and the event listener class (the one GUI class). Then when an instance (object) of some model class needs to notify the GUI to update its view of the game, it fires an appropriate event. You will see many examples of this in Chapter 8. I will also be glad to work with each of you in learning and using this critically important Model/View architecture.

Deliverables (Due on April 30, 2007 – No late work accepted)

In an envelope pleases include:

1) A hard copy of your source (there will probably be over a 1000 lines of code distributed over many classes so there will be many pages of source listing).
2) A CD or floppy that contains all the sub-directories produced by Visual Studio including the .sln file. This will enable me to bring up your project very quickly if I wish to carefully examine your source code and compile your application. An application that does not compile will get a grade of zero.
3) A detailed self-evaluation document containing:
   a) A list of know bugs or omissions. I will be much more lenient for a bug or omission from the specifications if you cite this bug or omission in a section entitled “Known bugs and Omissions.”

b) The usual comments related to any difficulties you encountered during the project and how these difficulties were overcome (if they indeed were overcome).

Deviations

You are at liberty to change the appearance (layout) of the GUI (compared to the one that I supplied you as my own implementation) provided that you strictly maintain the same sequence of board positions and provide the same level of detail in the GUI. You must include a scrollable transactions log. I will use this when I am testing your implementation. You are not allowed to modify the rules of the game in any way. If you detect any bugs in my implementation, please bring them to my attention and I will share them with the class. You are still responsible for implementing the rules as stated.

Hints

The architecture that I used involves over a dozen model classes (classes that model each element in the domain as well as some abstract classes) in addition to the one GUI class. There is no need for you to adhere to this number but I will look to see that you have created the desired separation between the classes that implement the game rules (business logic) from the GUI class whose only responsibility is to update and maintain a view of the game as it progresses.

I started my design by creating a class called *GameModel*. It uses the services of many other model classes (*Hair*, *Tortoise*, *Carrot*, *Lettuce*, etc.). The GUI class (in my implementation named *HairTortoiseUI*) contains a field of type *GameModel* named *model*. This is the only domain object contained within the GUI class.

I have used 64 *Panel* components in achieving the layout of the game (in the GUI class). They are stored in a one-dimensional array of type *Control* (a standard .NET class) in the GUI class. Listeners for the mouse events in class *Panel* namely *MouseHover*, *MouseLeave* and *Click* are defined in this GUI class. My details of method *MouseLeaveHandler* (my handler for the *MouseLeave* event) are given below (to demonstrate how an array of controls can be used in trapping the event of the mouse leaving one of the panel components contained in the GUI:

```
private void MouseLeaveHandler(Object sender, EventArgs e) {
   int index = 0;
   foreach (Control c in controls) {
        if (c is Panel && sender.Equals(c)) {
        model.RefreshDescription(index);
         break;
      }
      index++;
   }
}
```

A small portion of my GUI class including a few of the important fields and the code for my array of controls is given as follows (where *pos1*, *pos2*, …, *pos64* are the 64 panel components that I defined in the GUI class):

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace HairTortoise {

    // Enum type
    public enum GameLevel {
        Novice, Medium, Proficient
    }

    public partial class HairTortoiseUI : Form {

        // Fields
        private GameModel model = new GameModel();
        private Control[] controls; // Holds the 64 panels

      // Holds graphics objects for each panel
        private Graphics [] g = new Graphics[65];

   // Other fields not shown

   public void InitializePanels() {
            // Assign panel controls to the controls array
            controls = new Control[] {null, pos1, pos2,
              pos3,
              pos4, pos5, pos6, pos7, pos8,
              pos9, pos10, pos11, pos12, pos13, pos14, pos15,
              pos16, pos17, pos18,pos19, pos20, pos21, pos22,
              pos23, pos2, pos25, pos26, pos27, pos28, pos29,
              pos30, pos31, pos32,
                  pos33, pos34, pos35, pos36, pos37, pos38,
              pos39, pos40, pos41, pos42, pos43, pos44,
              pos45, pos46, pos47, pos48, pos49, pos50,
              pos51, pos52, pos53, pos54, pos55, pos56,
                  pos57, pos58, pos59, pos60, pos61, pos62,
                pos63, pos64};

            // Assign three handlers to each panel control
            foreach (Control c in controls) {
                if (c != null) {
```

```
                    c.MouseHover += new
                        EventHandler(MouseHoverHandler);
                    c.MouseLeave += new
                        EventHandler(MouseLeaveHandler);
                    c.Click += new
EventHandler(MouseClickHandler);
                }
            }

            // Assign graphics objects for each panel
            for (int index = 1; index <= 64; index++) {
                g[index] = controls[index].CreateGraphics();
            }
        // Rest not shown
```

You are of course free to use any mechanism for implementation that you wish.

Extra Credit

You are required only to implement novice-level play (you can omit the radio buttons if you plan on only implementing the required novice-level computer play).

If you decide to attempt extra credit by implementing “Intermediate” or “Advanced” computer play, please do the following:

1) Indicate in English narrative, the algorithm for each additional level of play (as I did for “Novice” level of play).
2) Include the three radio buttons in your implementation so that I may experiment with your levels of play.

Grading and Help

I will be as generous as possible in giving you credit for the work you have done. As indicated earlier, I will be more generous with those of you that are totally upfront with your detailed list of known bugs and omissions.

The grading will be based in part on how high a plateau you have reached. The plateaus from lowest to highest are the following:

1) Produced the layout of the game board GUI with all the bitmaps where they belong.
2) Correctly implemented the mouse hover and mouse leave events which show the consequence of attempting to move to any board position from any given board position that the human player is occupying.
3) Correctly implements the pane that shows the details of each move and correctly allows human moves to occur.
4) Correctly implements the response to the “Computer Move” button using the “Novice” logic described.
5) Allows a complete game to be correctly played at the “Novice” level.

6) Extra credit: Implements the “Computer Move” at either the “Intermediate” or “Advanced” level. If you pursue this extra credit and optional feature, please describe in words and in detail the algorithm that you use for each of the two more advanced levels for computer play.

Start your work as soon as possible on this project so that I can provide you with as much help as possible in a timely manner.

There will be class discussion of this project as well as individual help sessions (in my office scheduled during regular class time).

5 SOME COMMENTS

The snippets of GUI-based C# code that I have presented as part of the project specification document provides the basis for discussing event-handling on a collection of controls. Instead of assigning a unique event handler to each of the 64 panel components that represent the board positions, a single event-handler is defined using an array of controls. The same strategy is used for modifying and refreshing the graphics of the 64 bitmaps in response to mouse hover and mouse leave events.

The students are encouraged to use the Model/View (Observer) pattern and define a collection of event/delegate types that provide the basis for communication between the various model classes and the GUI class whose only responsibility is to present the game graphics/status and show a detailed chronological log of all the events that have occurred in the game. This log of events provides a wonderful debugging aid as the game is being developed.

The results of this project will not be known until April 30, after this paper is published. I may publish the results of this project in a future paper if any interesting insights are obtained from the students’ performance on this project.

About the author

Richard Wiener is the 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*.