CIS 201 Computer Science I
Fall 2009 Lab 08
Tic Tac Toe

October 26, 2009

- Implementing a class revisited
- Using `ArrayList`
- Game components with "state"
- Restarting a game

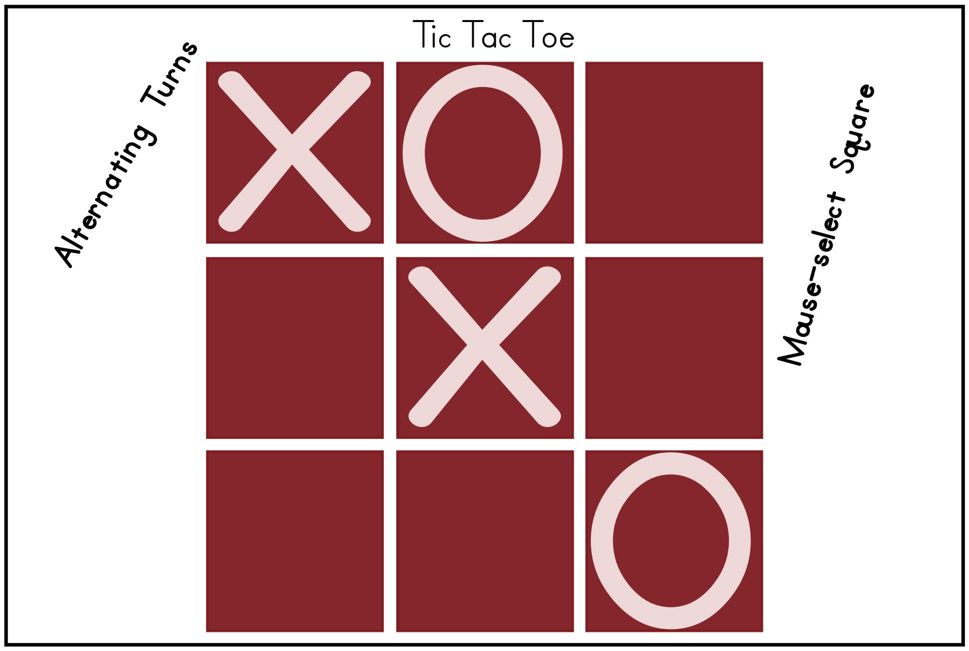

Figure 0.1: TicTacToe Design

Checkpoint 1 Getting Started

Consider a simple version of Tic Tac Toe: your `Game` creates and positions nine sprites on the screen. Then, in `advance` you would check for mouse clicks, comparing any click to each of the sprites. If one is clicked *and* it is empty, update the sprite with the current player's symbol and change who's turn it is. Make sure to check for winning and draws somewhere.

Think about how `advance` would look. Assuming the sprites are in an `ArrayList`, there would be a big loop with many nested `if` statements in it. This code is too complex to hold in your mind, all at once.

How do computer programmers respond to complexity?

Abstraction! (Hopefully you got that one.)

Abstraction is the hiding of details of some rule or some component. It can be applied by writing methods with descriptive names that make it easier to express the solution at a higher level. Abstraction can also be applied by breaking responsibility for the solution across multiple components.

The "simple" approach given above is difficult because the Tic Tac Toe game is responsible for handling both game-level details *and* square-level details. A separation of responsibility is possible if we build "smart" squares; this is a standard technique in object-oriented design, the creation of objects that "know what to do".

Before reading on, you and your partner should pull out a piece of paper and write down what activities a Tic Tac Toe game must support. Divide up your list into game-level and move-level activities.

Show your list of activities to one of the lab instructors; document your list by including it in a header comment for the `Game` extending class.

Show your work on Checkpoint 1 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Checkpoint 2 Define classes

When designing this lab, the following responsibilities seemed necessary. Each square is responsible for:

- Advancing the state of the square (each frame):
  - Checking whether it is legal to click on the square (it is not legal if the square is not empty)
  - Checking whether it has been clicked
  - If it has been clicked, updating according to which player's turn it was and ending the turn.
- `isEmpty` to determine whether or not the square is occupied.
- Return the current state of the square ("X", "O", or "").

The game is then responsible for

- Setting up the game: creating the squares, scaling and positioning them.
- Advancing every square every frame.
- Showing status.
- Returning the current player ("X", "O").
- Ending a turn.
  - Checking if the current player has won.
  - Checking if there is a cat's game.
  - Otherwise, change which player's turn it is.

This lab builds two cooperating classes, `TicTacToe` and `TicTacToeSquare`, each fulfilling one of these sets of requirements.

Note *how* the two classes cooperate: `TicTacToe` needs references to all of the squares so that it can call `advance` for each of them; `TicTacToeSquare` needs a reference to the game of which it is a part to be able to check for mouse clicks, determine whose turn it is, and to be able to end a turn. While `Game.getCurrentGame()` suffices for mouse clicks, the other actions are `TicTacToe` specific: each square is constructed with a reference to the game of which it is a part.

Start the `TicTacToe` and `TicTacToeSquare` classes in a Lab 8 directory.

One should extend `Game`, the other `CompositeSprite`. As discussed in class, build this lab one piece at a time; that means first get the squares to draw in the right places. Then make them aware of mouse clicks, and then, finally, we will make the game check for end-of-game conditions.

`TicTacToeSquare` should create a rectangle sprite of a dark (non-black) color. The "X"/"O" sprite comes later. This means writing a *constructor* for the class; there is no need for anything else right now.

What *parameters* does the constructor require? Since a `TicTacToe` is required for determining whose turn it is and ending a turn, there needs to be a field in `TicTacToeSquare` referring to a `TicTacToe` game. How would you declare a *field* of type `TicTacToe`? Declare such a field in `TicTacToeSquare`.

If the constructor takes a `TicTacToe` reference, how is it called when constructing a new square? Remember `this`? In any method in `TicTacToe`, `this` is a reference to a `TicTacToe` object. The call (probably in `setup`) in `TicTacToe` looks something like this:

```
    TicTacToeSquare someSquare = new TicTacToeSquare(this);
```

The reference to the `TicTacToe` makes it possible to check who's turn it is and get mouse clicks.

`TicTacToe` should construct nine `TicTacToeSquare` objects and put them in an `ArrayList`. They should be distributed in a 3 x 3 grid against the top of the screen. Each should be 0.20 screens square, centers of rows and columns should be 0.25 apart (at 0.25, 0.50, and 0.75 for the columns, at 0.10, 0.35, and 0.60 for the rows).

How the heck should we go from an index into an `ArrayList` to its corresponding screen coordinates? You should create two methods for `TicTacToe`, `row` and `column`, each of which takes a single integer parameter, the index into the `ArrayList` and then returns the row (or column) where that element belongs (0, 1, or 2). Then use a simple linear equation to go from column number to screen coordinates (and the same for rows).

```
  /**
   * Convert the index, range [0-9), into the corresponding row
   * on the screen, range [0-2).
   */
   public int row(int index) {
     // ... your code here ...
   }

  /**
   * Convert the index, range [0-9), into the corresponding column
   * on the screen, range [0-2).
   */
   public int column(int index) {
     // ... your code here ...
   }
```

You should also set up an integer to keep track of which player's turn it is (0 for "X", 1 for "O"), a `StringSprite` to hold a status message (scale to 0.10, position at the bottom-center of screen at 0.9 down (the reason the board is so high)).

Explain your `row` and `column` methods to the lab instructor.

Show your work on Checkpoint 2 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Checkpoint 3 Update status.

Create a method, `updateStatus`, which takes a `String` and updates the message displayed in the status string. Modify `setup` so that the status line displays which player's turn it is using your new method.

Show your work on Checkpoint 3 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Checkpoint 4 Square state

As documented above, a square must track its contents; this means a `content` field. The type should probably be an integer (to match the turn tracker in the game to ease picking what symbol to put in the square). Initialize content to a third value (not the one for "X" or "O") to indicate empty; perhaps -1.

Implement both a setter and a getter for the `content` field.

In the setter, you need to update a display value. Add a `StringSprite` to `TicTacToeSquare`. The scale should be 1.0 and it should initially (in the constructor) hold the empty string. It should use some non-FANG Blue light color. When setting the content of the square, set the text to "X" if the content is set to the x content value or "O" if the content is set to the o content value.

Implement `isEmpty` (a public, Boolean method) which returns `true` if the square is empty, `false` otherwise.

Show your work on Checkpoint 4 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Checkpoint 5 Square advance

Declare a new method for `TicTacToeSquare` called `advance`; model it on the `advance` in the game. Inside the method do the following:

```
if ((this square is empty) && (the mouse has been clicked))
  if (this sprite intersects mouse click)
    update content
    update appearance of StringSprite
    call theGame.finishTurn
```

In `TicTacToe`, call the `advance` method for every square on every frame. If you have to do something with every element in an `ArrayList`, *what should you be thinking*?

You can comment out the call to `finishTurn`; that should permit you to compile your program and click on the various squares and make them all "X" (turn doesn't change yet). Remember, incremental development, getting a little bit working before moving on, is another way to control complexity.

Show your work on Checkpoint 5 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Checkpoint 6 Finish turn

Implement `TicTacToe.finishTurn`: it takes no parameters and just changes who's turn it is. If turn was `0`, make it `1` and if it was `1`, make it `0`.

Uncomment the call to `finishTurn` in the `advance` method of `TicTacToeSquare`.

Show your work on Checkpoint 6 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Checkpoint 7 Winning and losing

Add checking for winning board position and cat's game. You should modify `finishTurn` to something like the following:

```
if (winner(turn)) {
  // handle win
} else if (catsGame()) {
  // handle cat's
} else {
  // change who's turn it is as before
}
```

Now you have to write the two methods listed. How will you check if it is a cat's game? Given that squares can give you their content it should be easy to check whether or not there are any empty squares: call `isEmpty` in `catsGame` for every square in the game...every square in the `ArrayList`...*what should you be thinking*?

The `winner` method is a touch messier: check the eight different ways that a game can be won. The player who just moved is provided as a parameter to the method so you can just check for contents matching that value.

To make the logic easier to follow, use a method, say `ndx` which, given a row and a column, returns the index in the `ArrayList` corresponding to that element. That means that checking one of the diagonals would just be:

```
(board.get(ndx(0, 0)).getContent() == turn &&
board.get(ndx(1, 1)).getContent() == turn &&
    board.get(ndx(2, 2)).getContent() == turn)
```

This Boolean expression will be true if and only if the values on the main diagonal equal the turn variable. So long as `turn` is either 0 or 1, we will only return true for a win along that diagonal.

Show your work on Checkpoint 7 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Checkpoint 8 New game

Is starting over a *square*-level responsibility or a *game*-level responsibility?

In the correct class define a `boolean` field, `waiting`. When `waiting` is `true`, the game is waiting to start over. In `advance`, rather than checking for mouse clicks on squares, check for the user pressing the space bar. If they do, call `startOver()` to restart the game (`setup` is called again).

The `waiting` flag is cleared (set to `false`) in `setup`. Where is it set `true`? Consider the three-way `if` statement where you check for winning or cat's games. If the game is over (win or cat's), set the flag and update the status message. Whenever a game ends, `waiting` is set and when a game starts it is cleared.

Show your work on Checkpoint 8 to the lab monitor, answering any necessary questions for them. Have them sign before continuing.

Also make sure that both partners have copies of the program. This program has some bearing on upcoming assignments; you'll want this code around. Comments will also be very helpful in this program for exactly that reason.

Log off of the lab computer you are using before leaving they lab. Anyone entering the lab has unlimited access to your files if you remain logged on. **DO NOT** turn off lab computers! They are a shared resource and there might be someone else logged in to "your" machine.