

Lab Tic Tac Toe

- Implementing a class revisited.
- Game components with “state”

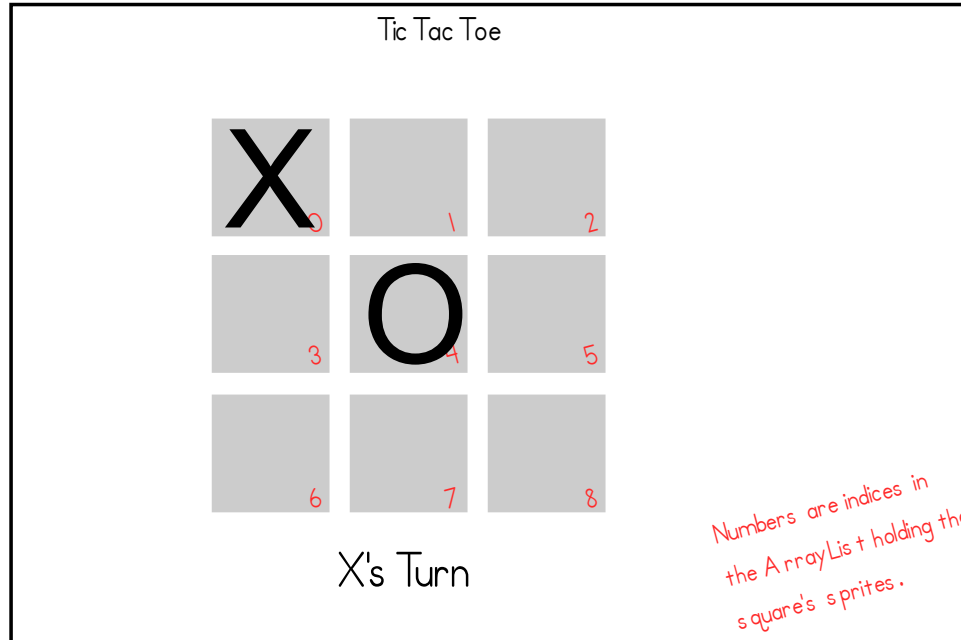


Figure 0.1: TicTacToe Design

Phase1. 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 separate variables, there would be a variable to hold a mouse click 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!

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.

Checkpoint 1:: Show your list of activities to one of the lab instructors before proceeding. This is very important. You will also type your list into the header comments of your game.

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 will need 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. This means that squares will be constructed with a reference to their game. (`Game.getCurrentGame()` is insufficient – you need to be able to determine whose turn it is).

Phase2. Start the TicTacToe and TicTacToeSquare classes. One should extend Game, the other should extend CompositeSprite. As we have discussed in class, we will build this lab one piece at a time; that means we will first get the squares to draw in the right places. Then we will 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. You will add the “X” and “O” sprite 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 Game is required for getting colors (and, eventually, checking for mouse clicks), you should pass in the game when constructing a square. You will have a field in TicTacToeSquare to refer to the game; perhaps something like this:

```
private TicTacToe theGame;
```

Then you will call the constructor with something like this (in TicTacToeSquare).

```
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).

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)).

Checkpoint 2:: When you have the nine dark boxes on the screen, raise your hand and signal one of the lab instructors to come and look at your code.

Phase3. Create methods to update the status message. You will need to be able to tell the user who's turn it is, who won, and that the game is a draw. Also, in the game ending messages, tell the user to press the space bar to play again. Modify your setup program so that it displays which player's turn it is.

Phase4. `TicTacToeSquare` needs to track its own contents. This means you need a content field (which should match the type of the turn tracker in the game). Initialize the content to -1 (to indicate empty).

Implement `isEmpty` (a public, Boolean method) and a getter method for the content (so `TicTacToe` can see the contents; why does the game need that information?).

To display the content, add a `StringSprite` field. Scale should be 1.0 and it should initially contain the empty string. Set its color to some light color and make sure the sprite is added to the composite. When you set the content of your square, you will set the text to either "O" or "X".

Phase5. Create an advance method in `TicTacToeSquare` and call it from `TicTacToe`'s 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.

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.

Phase6. 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.

Checkpoint 3:: Show one of the lab instructors your current program. It should be possible to click on each of the empty squares and turn it to an "X" or an "O".

Phase7. 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.

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.

Checkpoint 4:: Flag down a lab instructor and show them that your game detects a cat's game and a win.

Phase8. Finally, set up so the game can start over. Add a flag, `waiting` which is set to `false` in `setup`. Then, in advance, if `waiting` is `true`, check for the player pressing the space bar. If they do, call `startOver()` to restart the game.

Where would `waiting` be set? How about in the three-way `if` where you check for winning and cat's games. If it is a win (or a cat's game), then the game is waiting to start over so you set the flag. That way the space bar doesn't reset the game until the current game is over.

Phase9. Upload Your Java files.

Make sure both authors' names are in all of the header comments!

Type the two lists of features (from check point 1) into the header comment of `TicTacToe`. Clearly label them as to which are square and which are game responsibilities.

Go to Moodle; you will find an assignment in week 9 labeled Lab8. Go there and upload all of the Java files you created/modified in lab today. Note that there will be other files in your Lab8 directory. You want to make sure you upload just the `.java` file.

Also make sure that both partners have copies of the program. This program has some bearing on the current assignment so you probably want to have this code around. Comments will also be very helpful in this program for exactly that reason.