CS134 Lecture 27:
Tic Tac Toe 3

Announcements & Logistics

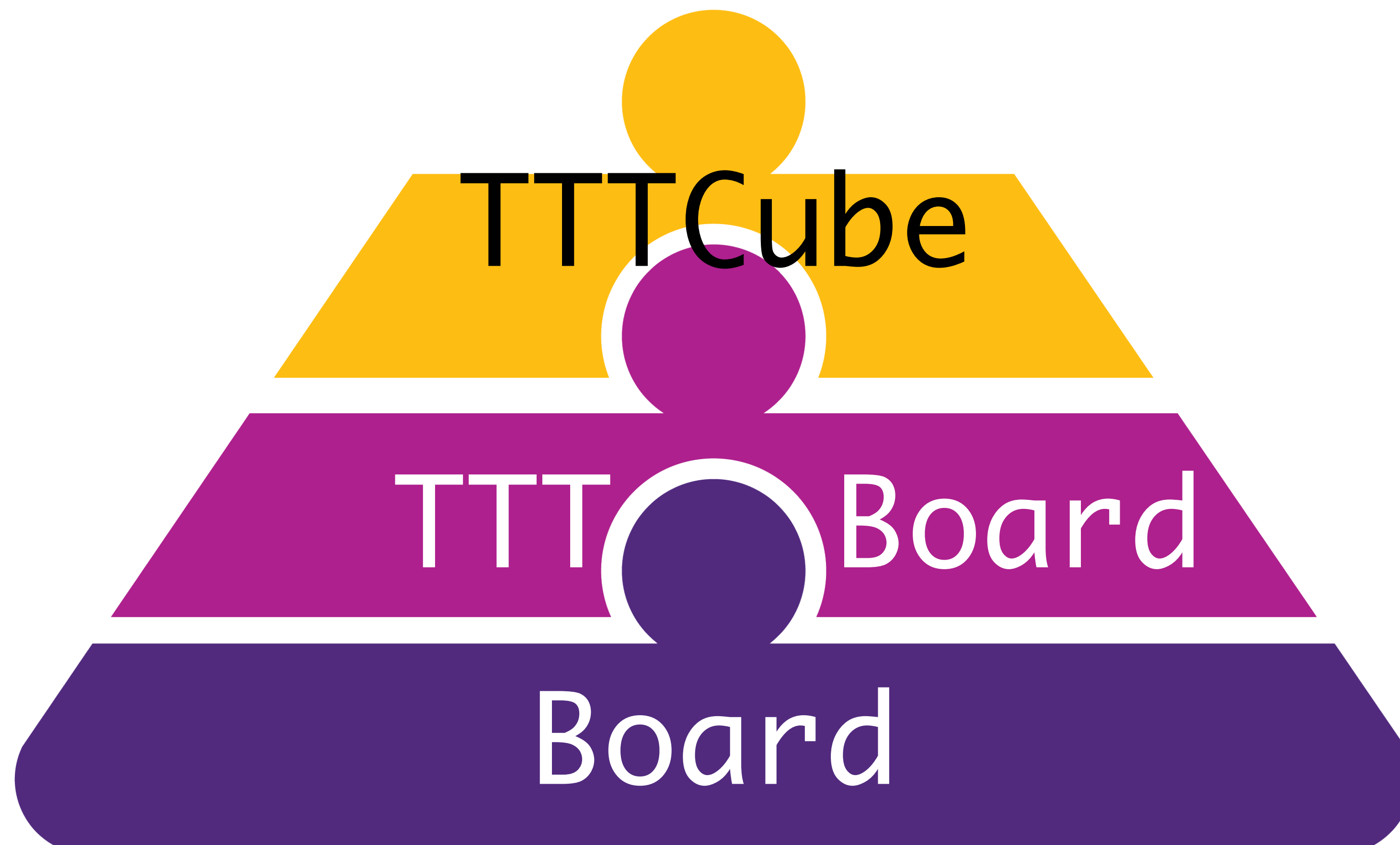
- **HW 8** due tonight @ 10 pm
- **Lab 9 Boggle**: two-week lab released
  - **Part 1** due next Wed/Thur 10 pm
  - **Part 2** due May 1/2 (handout will be posted soon)
  - Both parts have a **prelab** due at the beginning of lab
  - Can solve jointly with partner/ or individually and then discuss
  - Have it ready on a sheet of paper at the start of lab

Do You Have Any Questions?

Last Time

- Implemented a text-based class to represent a TTTBoard and TTTCube
- Discussed the game logic through a flow-diagram
- Before that, we discussed a graphical `Board` class to display a board
- Today we will bring these together:
    - Use graphical Board class to design a graphical tic-tac-toe game

TTTCube Class



TTTCube Class

- Attributes of text based class from last time?
  - `_letter ("X", "O")`
- In a graphical game, the TTTCube is placed on a board grid cell
  - What type of new data attribute can capture this location?
  - `_row, _col`
- Let's start with a TTTCube with these attributes
  - Later, we might want to add more (e.g., color of cube?)

TTTCube Class

```
class TTTCube:
    """A TTT Cube has several attributes that define it:
           *  _letter:  denotes the letter 'X', 'O', or '-'
           *  _row, _col:  denotes the position on the grid this
cube is placed
    """
    __slots__ = ['_letter', '_row', '_col']

    def __init__(self, row=-1, col=-1, letter=""):

        # set row, column and letter attributes
        self._row = row
        self._col = col
        self._letter = letter

    def get_row(self):
        return self._row

    def get_col(self):
        return self._col

    def get_letter(self):
        return self._letter

    def set_letter(self, char):
        if char in "XO-":
            self._letter = char
```

What other methods will be useful to have in this class?

TTTCube Class

```
class TTTCube:
    """A TTT Cube has several attributes that define it:
           *  _letter:  denotes the letter 'X', 'O', or '-'
           *  _row, _col:  denotes the position on the grid this cube is placed
    """
    # Continued


    def place_cube(self, board, fill_color="white"):
        '''Updates the grid cell on Board to display TTTCube'''
        row, col, let = self.get_row(), self.get_col(), self.get_letter()
        board.set_grid_cell(row, col, let,  "black", fill_color)

    def __str__(self):
        l, row, col = self.get_letter(), self._row, self._col
        return "{} at Board position ({}, {})".format(l, row, col)

    def __repr__(self):
        return str(self)
```

Updates the graphical grid cell to display the letter on the board

TTTCube: Testing

- Let's test the class by adding code to `if __name__ == "__main__":`

```
if __name__ == "__main__":
    win = GraphWin("Tic Tac Toe", 400, 400)
    board = Board(win, rows=3, cols=3)

    board.draw_board()

    tttcube1 = TTTCube(1, 1, "X")
    tttcube2 = TTTCube(1, 2, "O")

    tttcube1.place_cube(board, "light blue")
    tttcube2.place_cube(board, "pink")

    # pause for mouse click before exiting
    point = win.getMouse()
    win.close()
```

Create a graphical window of size 400 by 400 pixels with title "Tic Tac Toe"

Create a board with a 3 x 3 grid

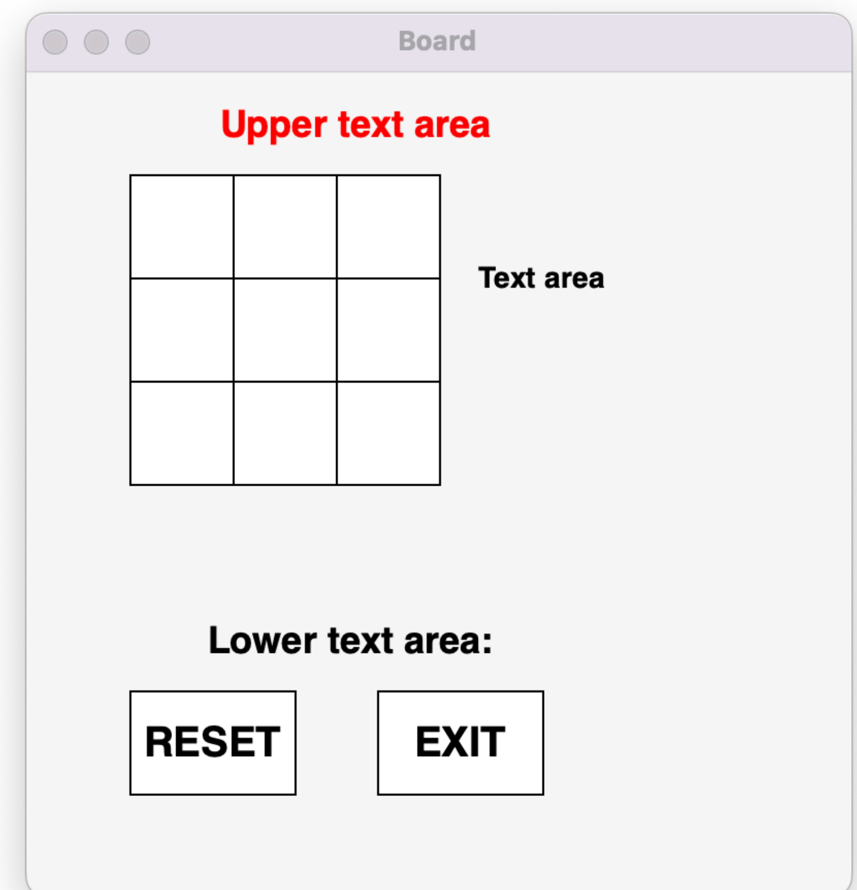
Display cubes by placing them on the grid

TTTBoard: Code in Class

TTTBoard Class

TTTBoard Class

- TTTBoard class will inherit all its graphical features from Board
- Recall that the Board class creates a generic graphical board with a grid, reset and exit buttons, and three text areas
- TTTBoard will inherit these (no need to write rewrite any code)
- What additional TTT specific attributes/methods should the board have?
    - TTTCubes that go on the grid
    - TTT game specific methods to check for win, etc

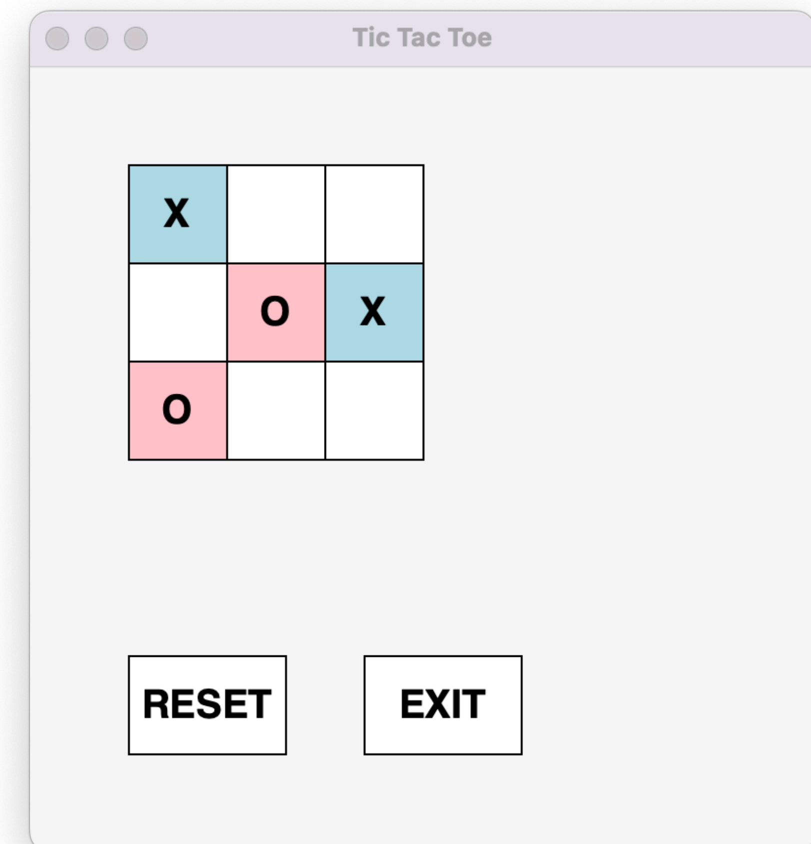


Review: Board Class

- Let's review the key features of the Board Class for using it
- Useful data attributes:
  - `_rows, _cols`: dimensions of the play area represented by the grid
  - `_grid:` list of list of "grid cells"
    - each cell is a TextRect object from the graphics module
- Useful methods:
  - `get_position(point):` given a point in the screen, returns the row, col of the grid cell if that point is in the grid
  - `set_grid_cell(row, col, text, text_color, fill_color)`
  - setter methods to change the text on the 3 text areas

`TTTBoard`: Design

- New attribute: `_cubes` (list of TTTCubes)
  - cubes get "placed" on the corresponding row, col on the board grid
- Cubes vs grid:
  - Cubes hold the "data" (letter, row, col)
  - Grid cells handle the graphics
- Separating graphics and other state is good
  - Abstraction and encapsulation
  - Makes it easier to debug as well



Initializing the TTT Board

```
def __init__(self, win):

    # call Board init
    super().__init__(win, rows=3, cols=3)

    # initialize new attribute
    self._cubes = []
    for row in range(self._rows):
        cube_row = []
        # next part could be a list comprehension!
        for col in range(self._cols):
            # create new TTTCube, specifying grid coord
            cube = TTTCube(row, col)

            # add TTTCube to row
            cube_row.append(cube)

        # add column to grid
        self._cubes.append(cube_row)

    # display the cubes on the board
    self.place_cubes_on_board()
```

Inherit from Board

Call Board's __init__ method

List of list of TTTCubes

Getter Methods: TTTBoard

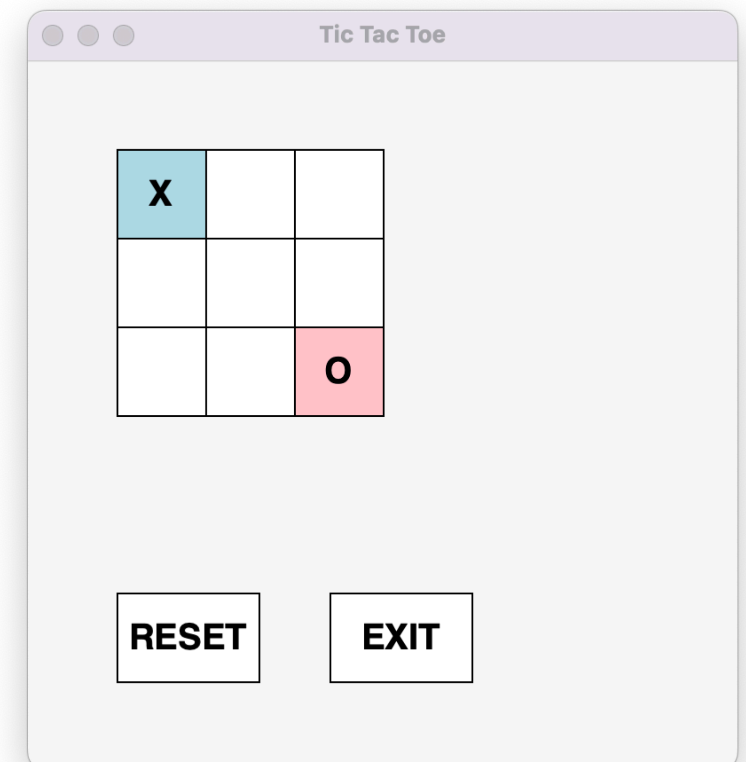
- TTTBoard acts as the middle that layer that communicates between the interactive game (mouse clicks) and the graphical base (Board)
- To do that effectively, need a way to translate points on graphical window to grid location and consequently TTTCubes on it
    - Board does some of these (`get_position` gives grid coordinates of a point in the grid)
- Need another getter method to map point in screen to the `TTTCube`

```
def get_ttt_cube_at_point(self, point):
    """Returns the TTTCube at point on window (a screen coord tuple)"""
    if self.in_grid(point):
        # get_position returns grid coords
        (row, col) = self.get_position(point)
        return self._cubes[row][col]
    return None
```

Setter Methods: TTTBoard

- What TTTBoard change might we want to change?
    - Set graphics to display TTTCubes
    - Set/reset board state for play

```
def place_cubes_on_board(self):
    '''Updates the board to display the letters on TTTCubes'''
    for row in range(self._rows):
        for col in range(self._cols):
            let = self._cubes[row][col].get_letter()
            self._grid[row][col].setText(let)

# reset all letters and colors of grid
def reset(self):
    """Clears the TTT board by clearing
    letters and colors on grid"""
    for x in range(self._rows):
        for y in range(self._cols):
            # get letter out of grid and reset it
            board.set_grid_cell(x, y, "")
```



`TTTBoard` Helper Methods: Checking for Wins

Checking for Win

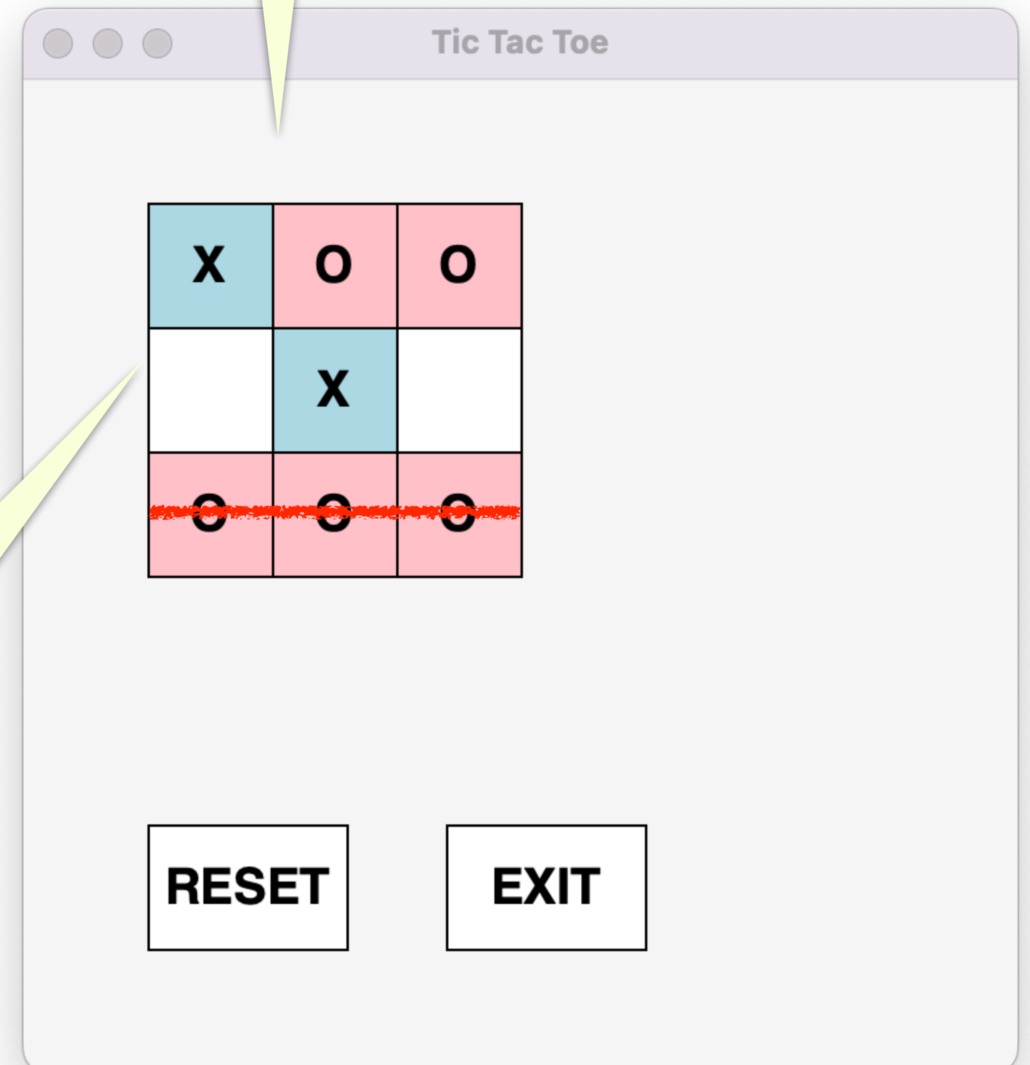
- A player ("X" or "O") wins if:
  - There exists a column filled with their letter, OR
  - There exists a row filled with their letter, OR
  - There exists a diagonal that is filled with their letter
- Let's break that down into separate private helper methods
  - `_check_rows`
  - `_check_cols`
  - `_check_diagonals`

Checking the Rows

- For a given letter ("X" or "O"), we need to find if there is ANY row that is made of only `letter`
- How can we approach this?

Grid positions are `(row, col)`

```python
def _check_rows(self, letter):
    """Check rows for a win (3 in a row)."""
    # does letter appear in an entire row?
```



check_rows checks the board **horizontally**

Checking the Rows

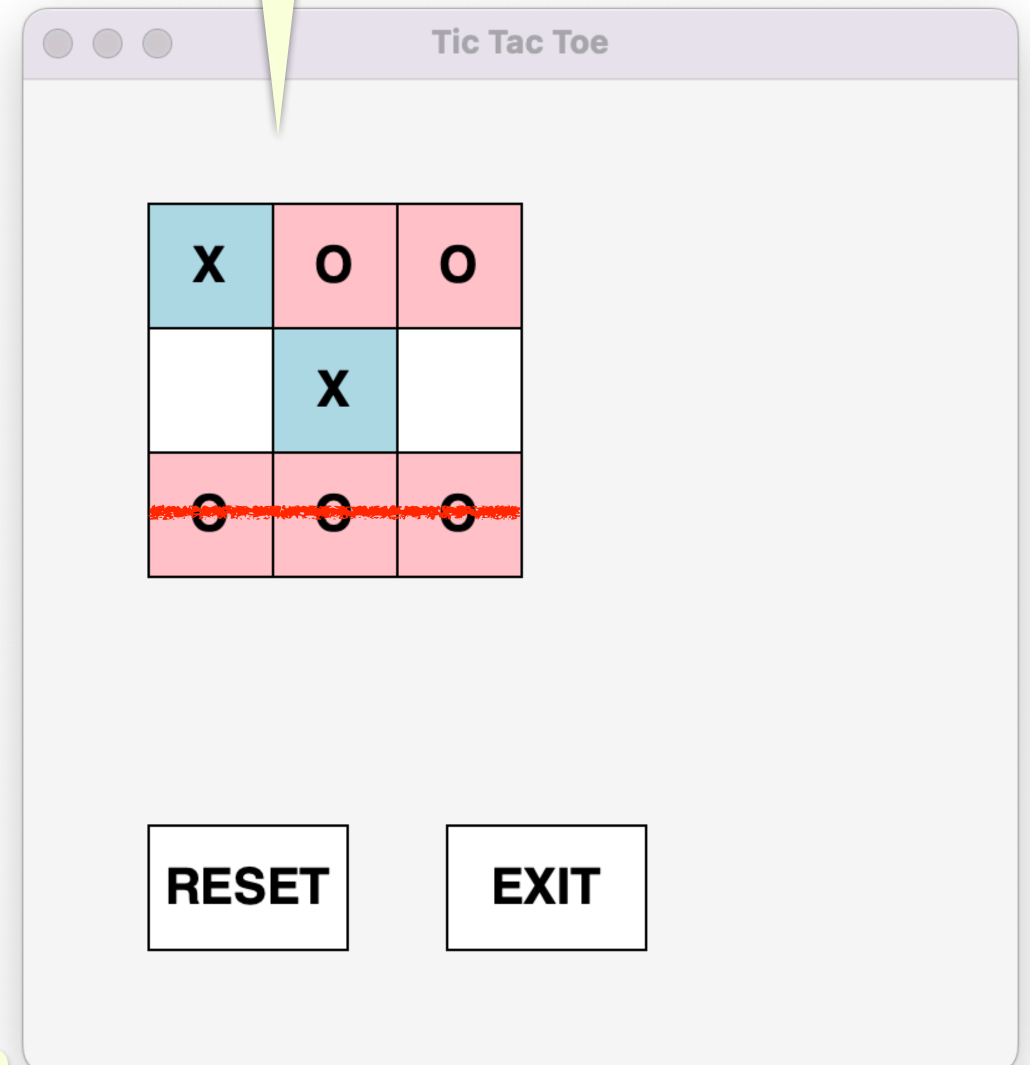
- For a given letter ("X" or "O"), we need to find if there is ANY row that is made of only `letter`
- How can we approach this?

Why initialize `count` here?

```
def _check_rows(self, letter):
    """Check rows for a win (3 in a row)."""
    for row in range(self._rows):
        count = 0
        for col in range(self._cols):
            cube = self._cubes[row][col]

            # check how many times letter appears
            if cube.get_letter() == letter:
                count +=1

        # if this is a winning row
        if count == self._rows:
            return True

    # no winning row found
    return False
```

If all letters match, return True

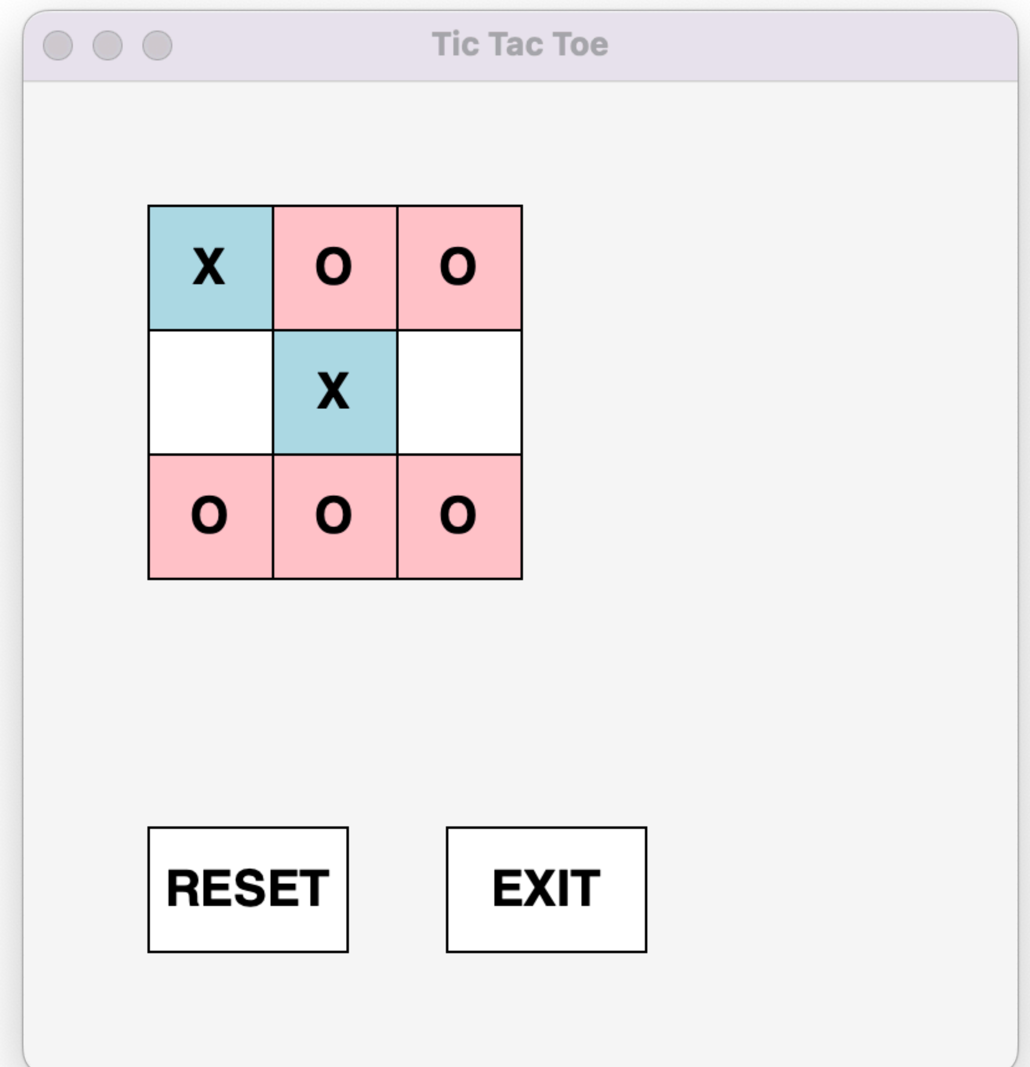
Grid positions are `(row, col)`



Similarly Check Columns

- We can similarly check a column for a win

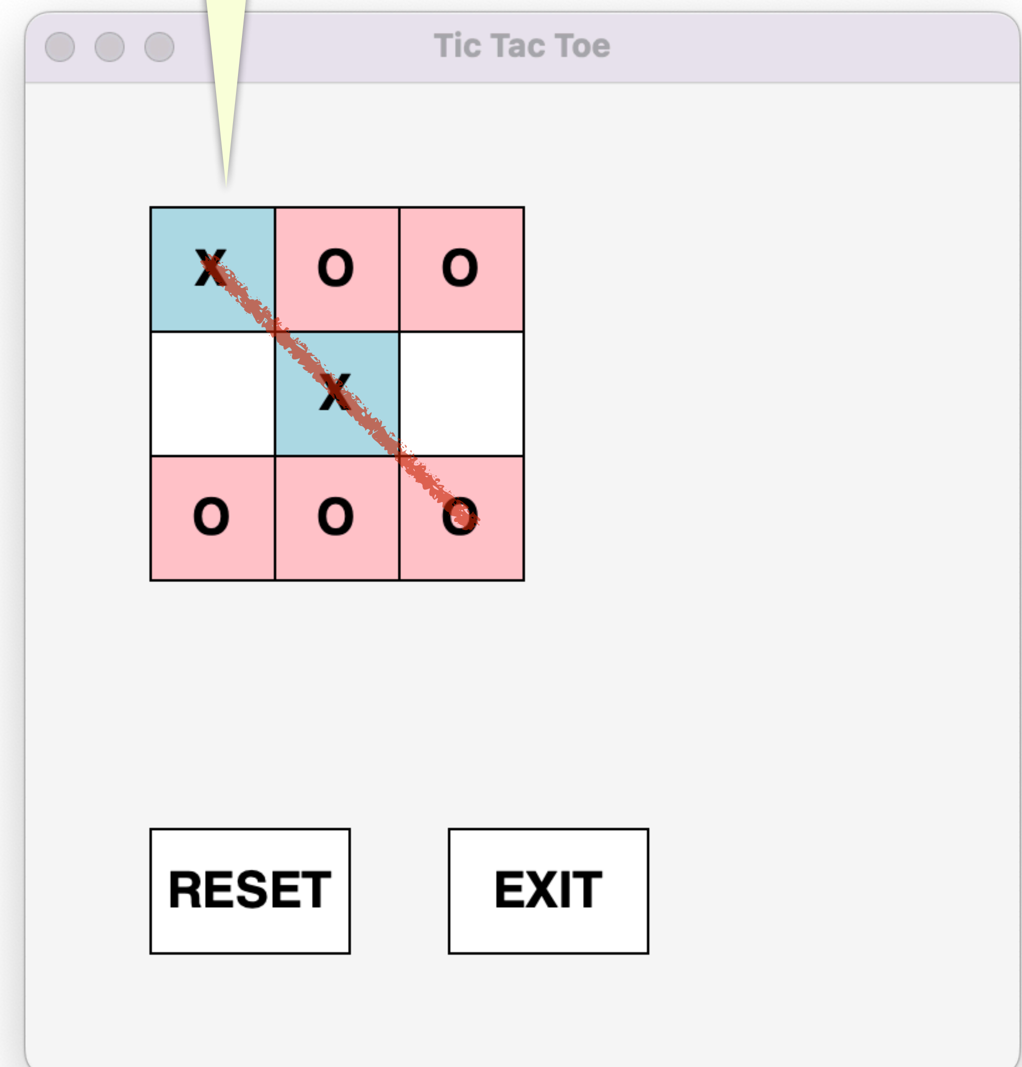
```
def _check_cols(self, letter):
    """Check columns for a win (3 in a row)."""
    for col in range(self._cols):
        count = 0
        for row in range(self._rows):
            cube = self._cubes[col][row]

            # check how many times letter appears
            if cube.get_letter() == letter:
                count +=1

        # if this is a winning row
        if count == self._cols:
            return True

    # if no winning rows
    return False
```



Check Diagonals

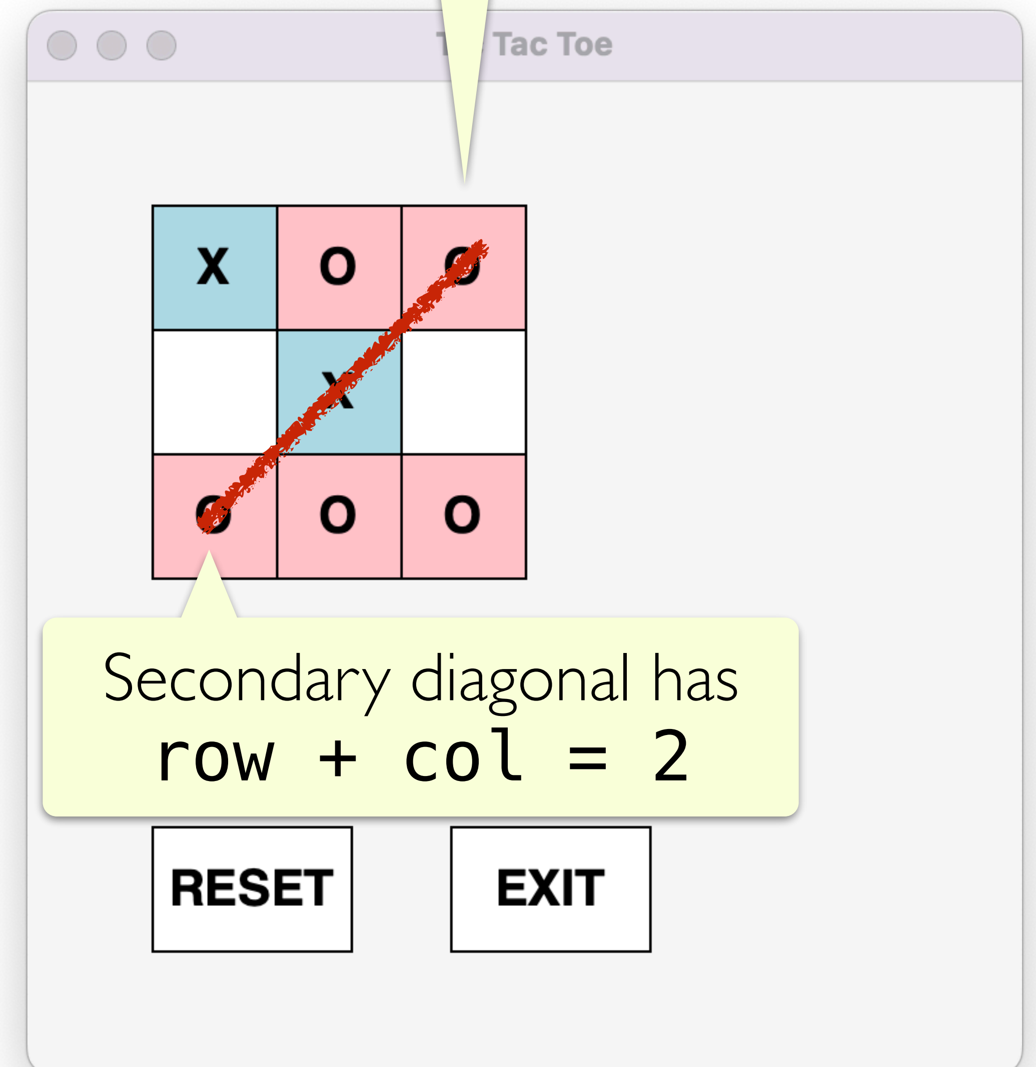
```python
def _check_diagonals(self, letter):
    """Check diagonals for a win (3 in a row)."""
    # counts for primary and secondary diagonal
    count_primary, count_second = 0, 0

    for col in range(self._cols):
        for row in range(self._rows):
            cletter = self._cubes[col][row].get_letter()

            # update count for primary diagonal
            if (row == col and cletter == letter):
                count_primary += 1

            # update count for secondary diagonal
            if (row + col == self._rows - 1 and
                                cletter == letter):
                count_second += 1

    # return true if either win
    primary_win = count_primary == self.get_rows()
    second_win = count_second == self.get_rows()
    return primary_win or second_win
```

Primary diagonal has row/col same



Check Diagonals

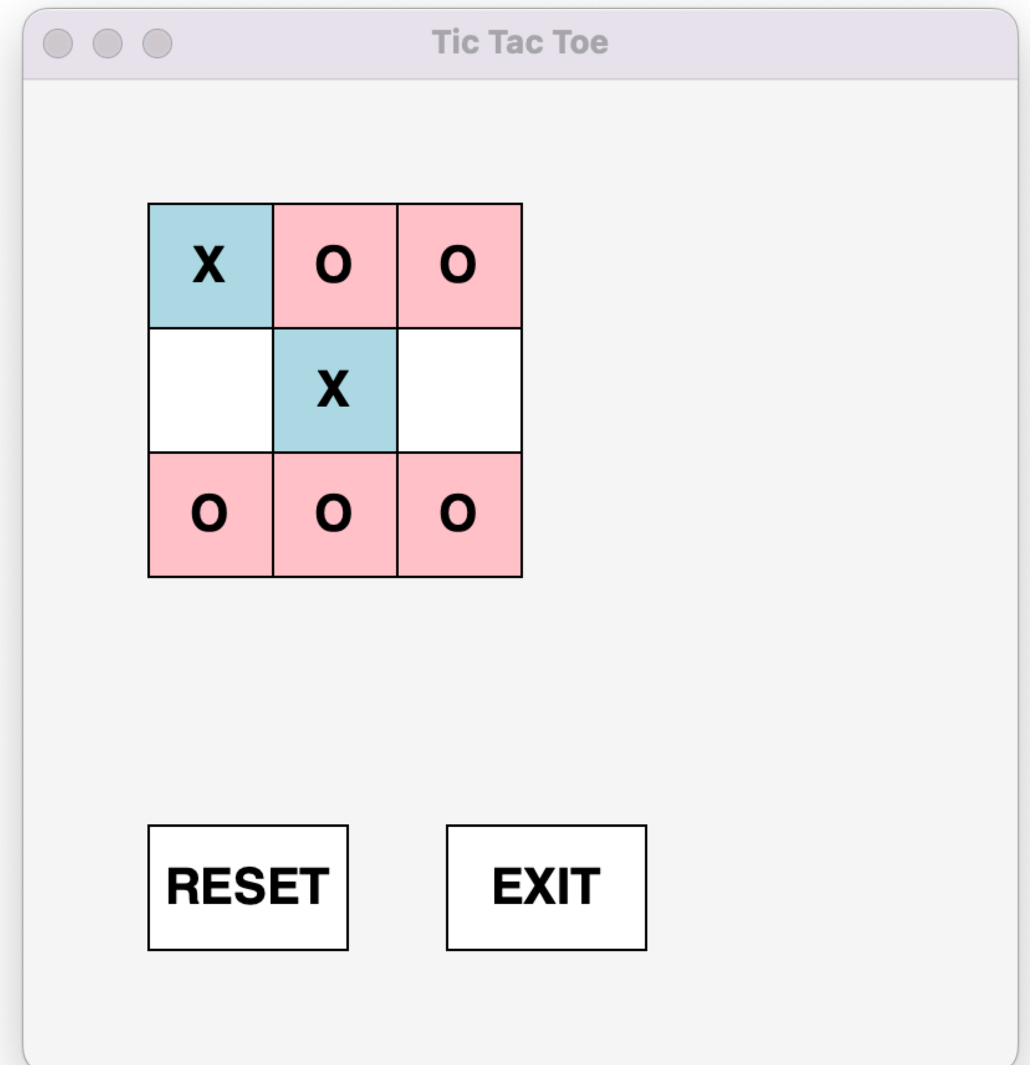
```
def _check_diagonals(self, letter):
    """Check diagonals for a win (3 in a row)."""
    # counts for primary and secondary diagonal
    count_primary, count_second = 0, 0

    for col in range(self._cols):
        for row in range(self._rows):
            cletter = self._cubes[col][row].get_letter()

            # update count for primary diagonal
            if (row == col and cletter == letter):
                count_primary += 1

            # update count for secondary diagonal
            if (row + col == self._rows - 1 and
                                cletter == letter):
                count_second += 1

    # return true if either win
    primary_win = count_primary == self.get_rows()
    second_win = count_second == self.get_rows()
    return primary_win or second_win
```



Final Check for Win

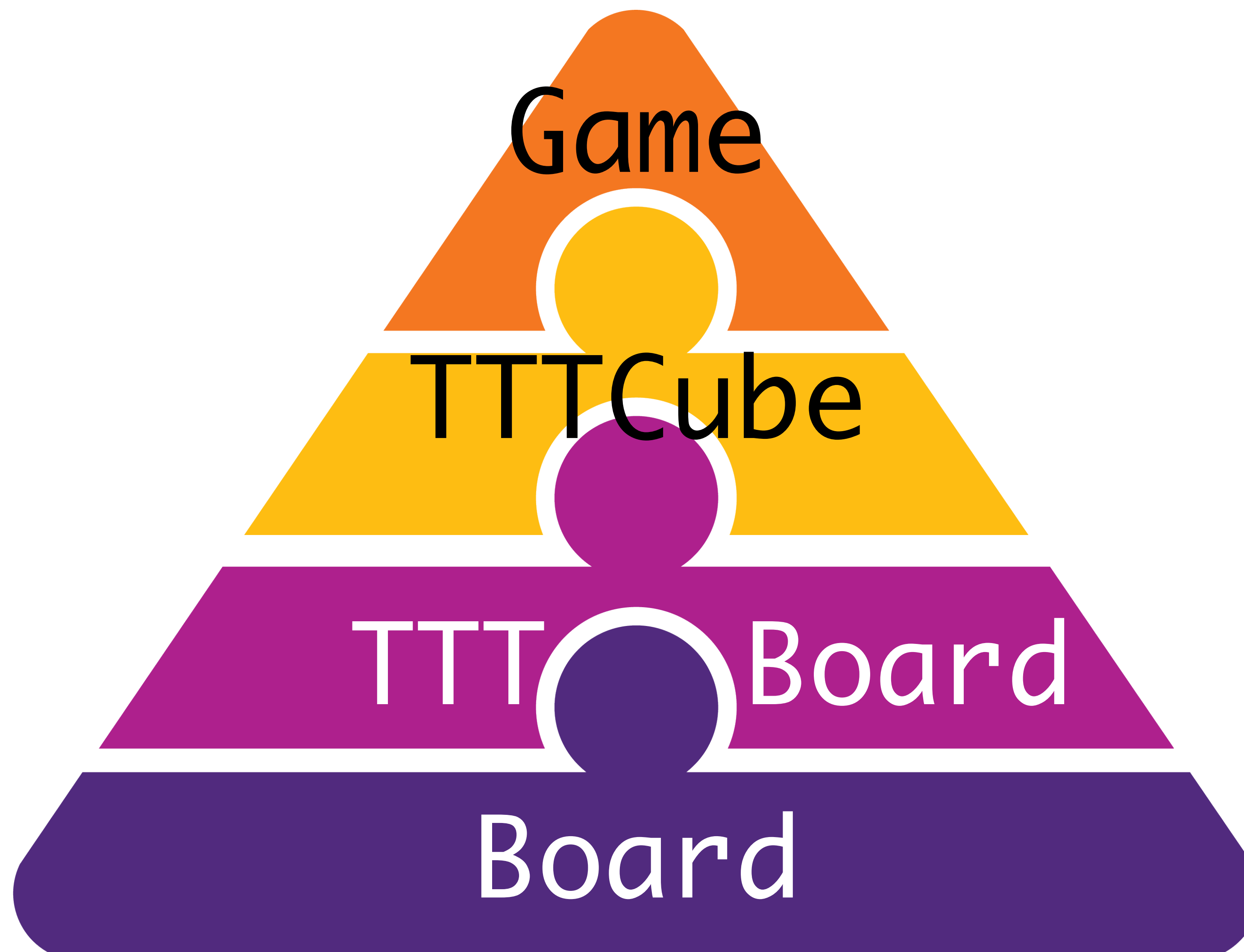
- Putting it all together: the board is in a winning state if any of the three winning conditions are true
- We will make this method public as it will needed outside of this class

```
def check_for_win(self, letter):
    """Check board for a win."""
    row_win = self._check_rows(letter)
    col_win = self._check_cols(letter)
    diag_win = self._check_diagonals(letter)

    return row_win or col_win or diag_win
```



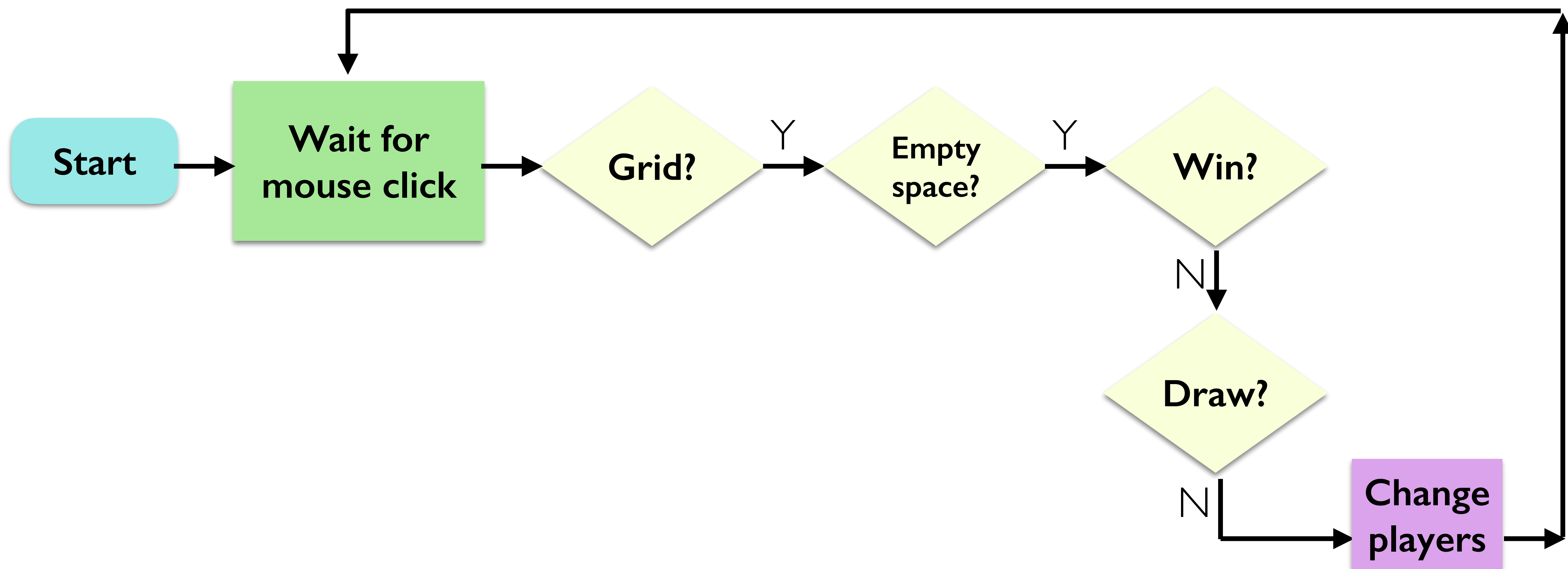
TTTGame Logic

TTT Game Logic



Finally…TTT Game Logic

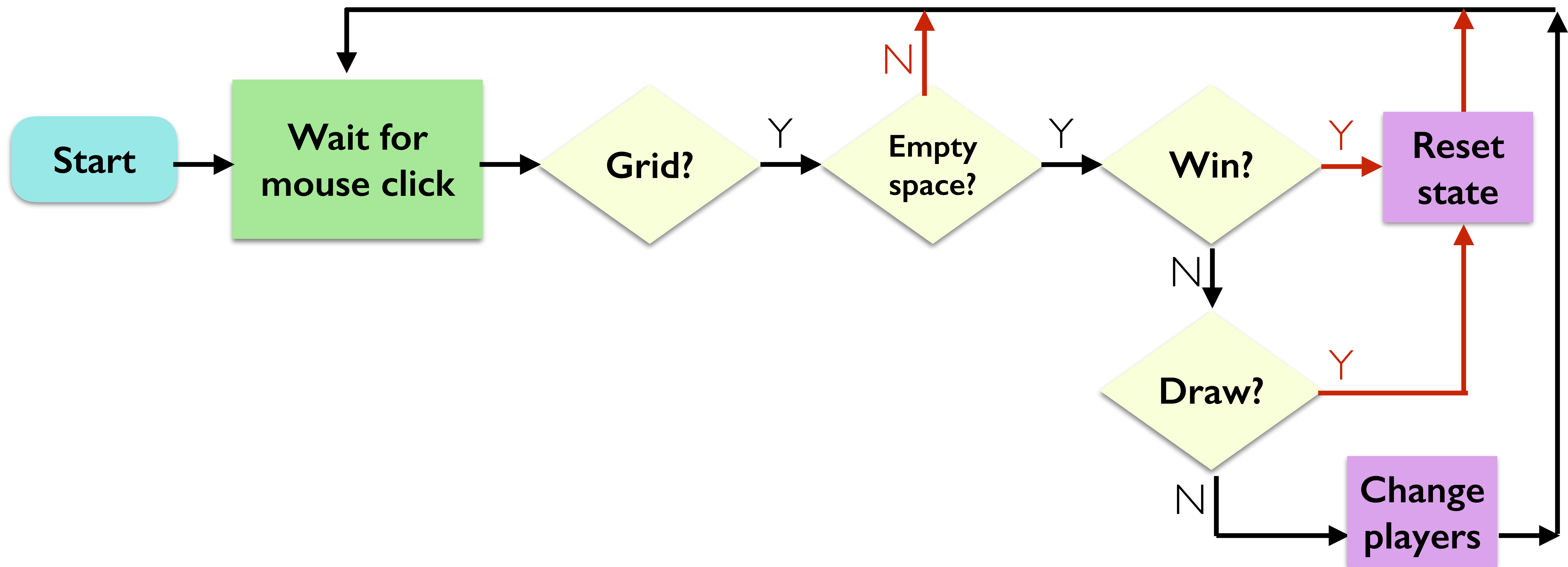
- Let's create a TTT flowchart to help us think through the state of the game at various stages



Let's think about the "common" case: a valid move in the middle of the game

Finally…TTT Game Logic

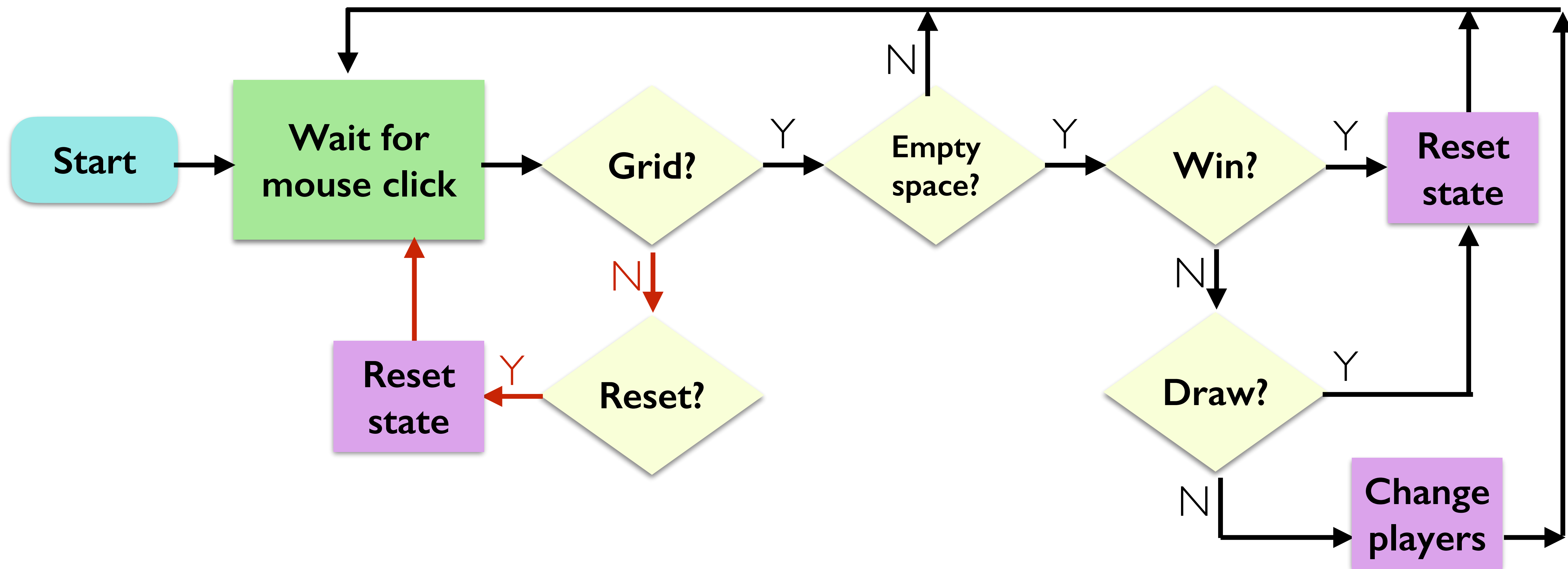
- Let's create a TTT flowchart to help us think through the state of the game at various stages



Now let's consider the case of a win, draw, or invalid move

Finally…TTT Game Logic

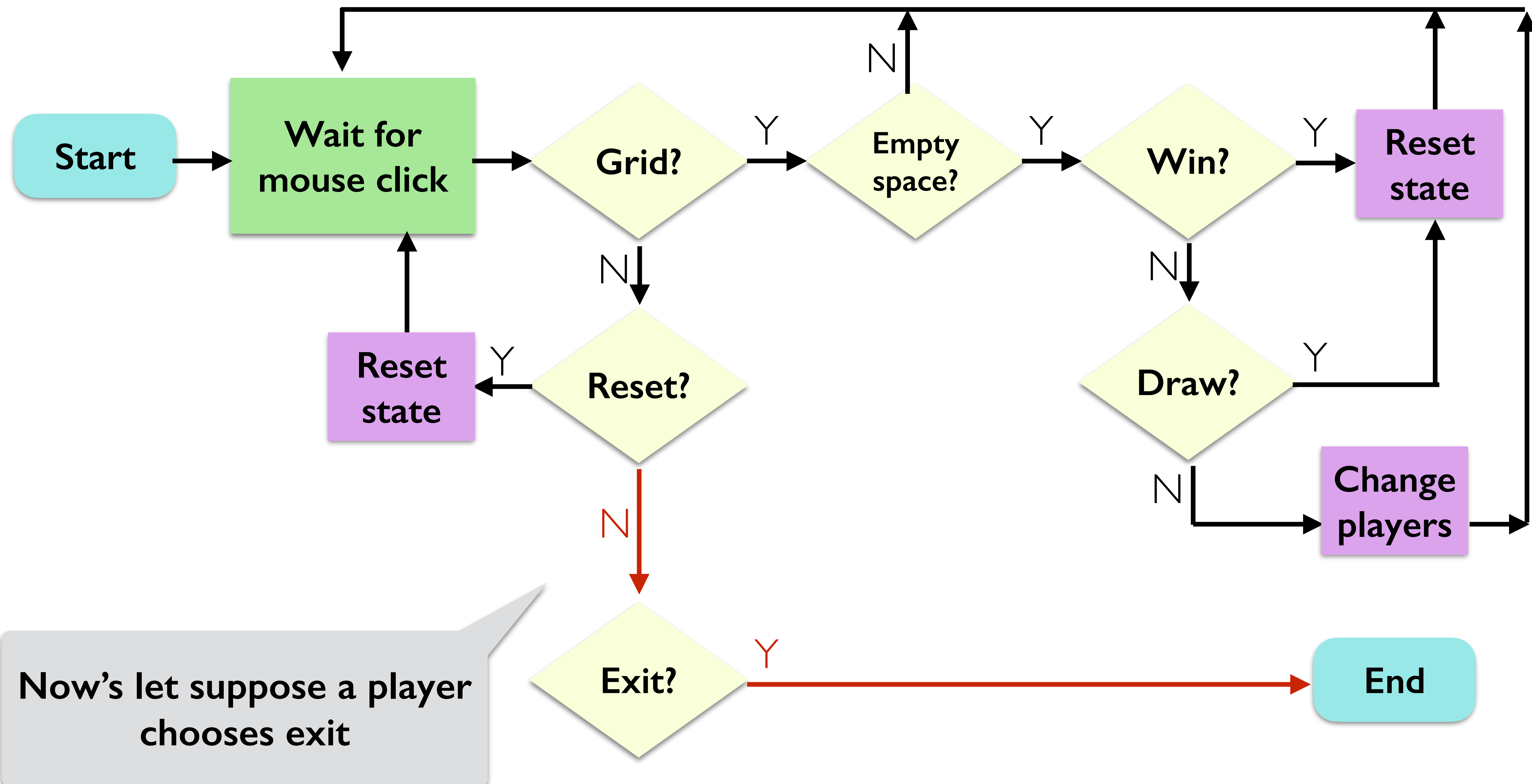
- Let's create a TTT flowchart to help us think through the state of the game at various stages



Now's let suppose a player chooses reset

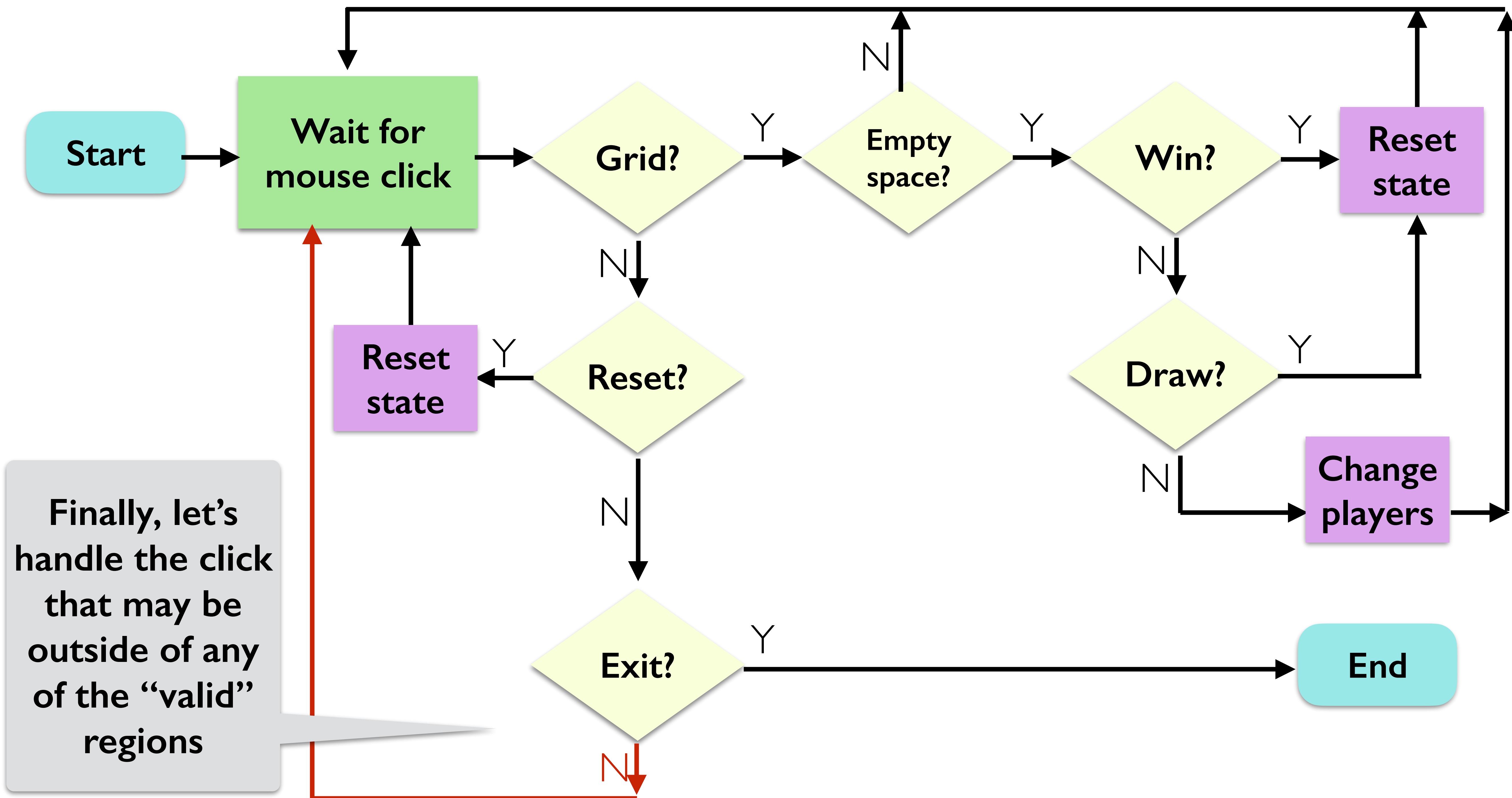
Finally…TTT Game Logic

- Let's create a TTT flowchart to help us think through the state of the game at various stages



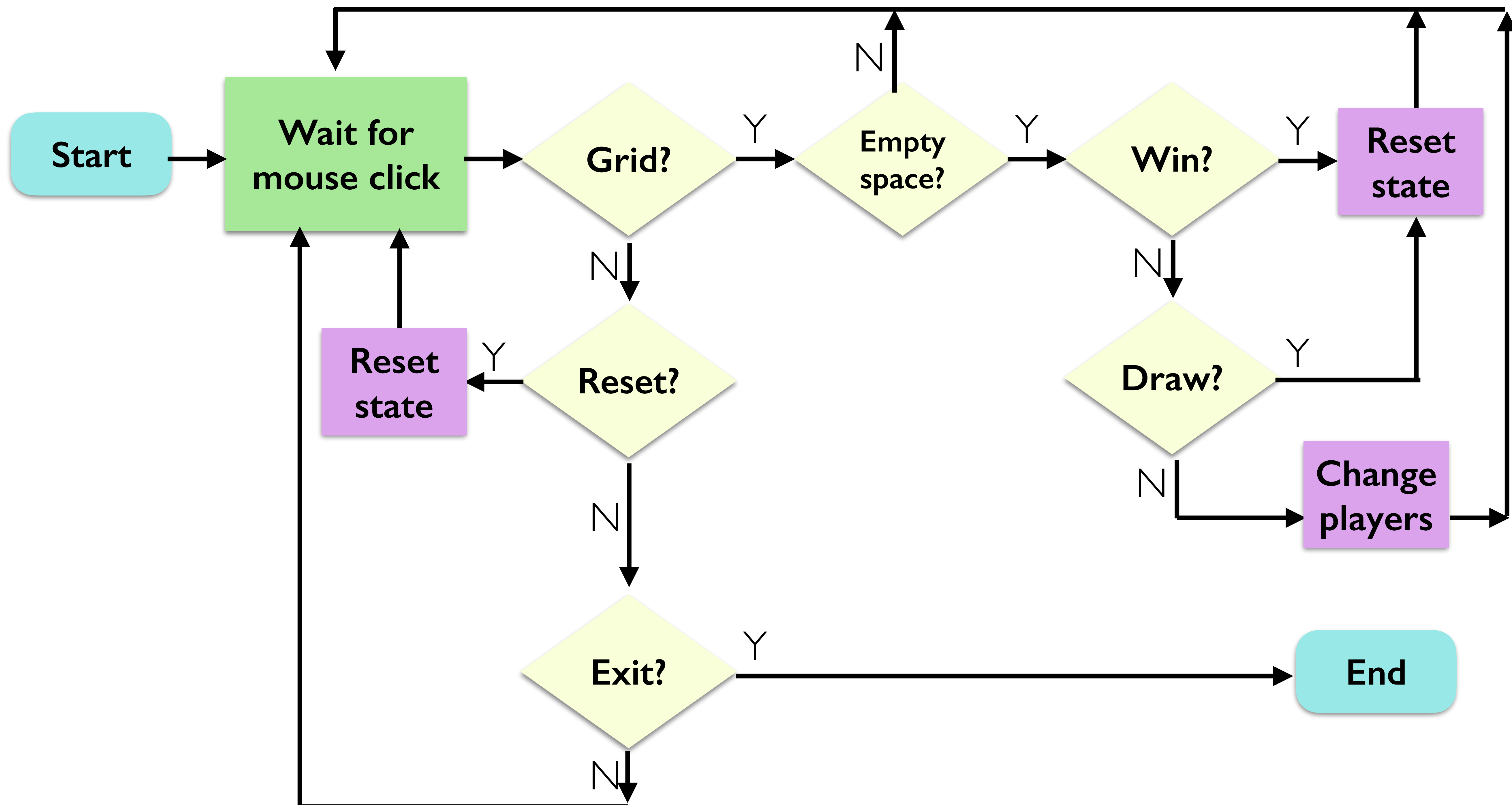
Finally…TTT Game Logic

- Let's create a TTT flowchart to help us think through the state of the game at various stages



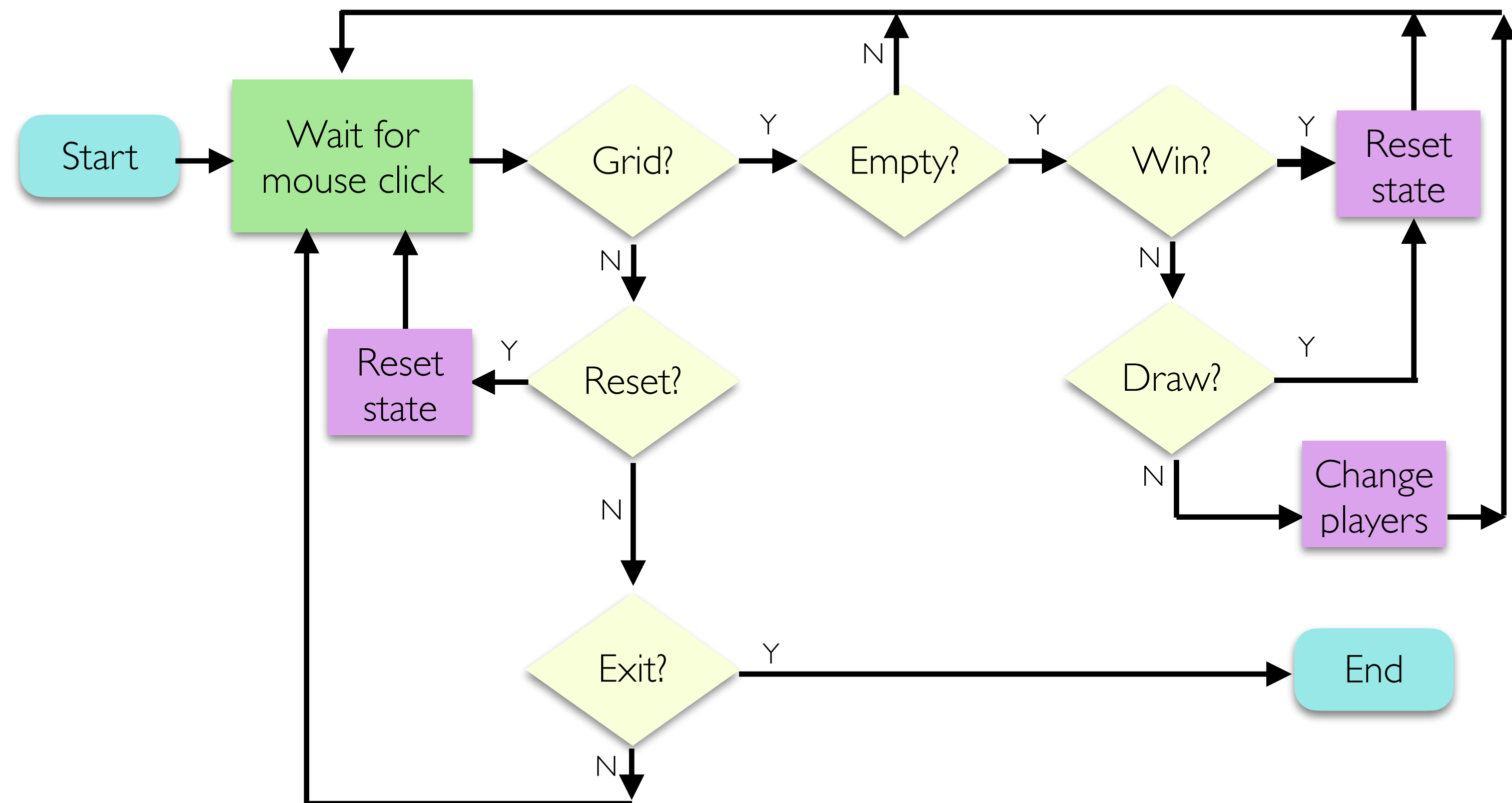
Finally…TTT Game Logic

- Let's create a TTT flowchart to help us think through the state of the game at various stages



Translating our Logic to Code

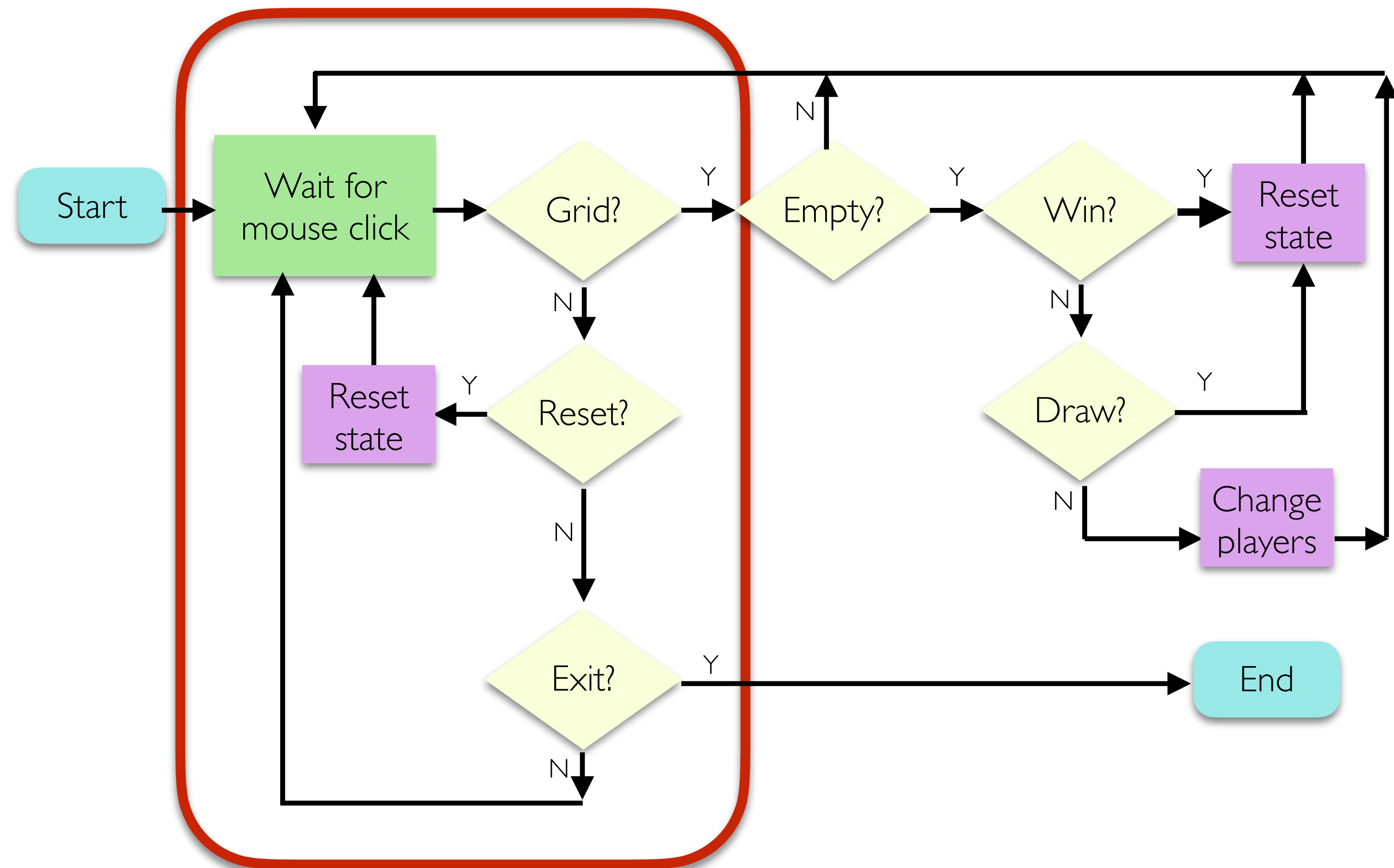
- Let's think about `__init__`:
  - What do we need?
    - a `board`, player, and maybe `num_moves` (to detect draws easily)



Translating our Logic to Code

- Now let's write a method for handling a single mouse click (point)
- The game continues (waits for more clicks) if this method returns True
- If this method returns False, game ends

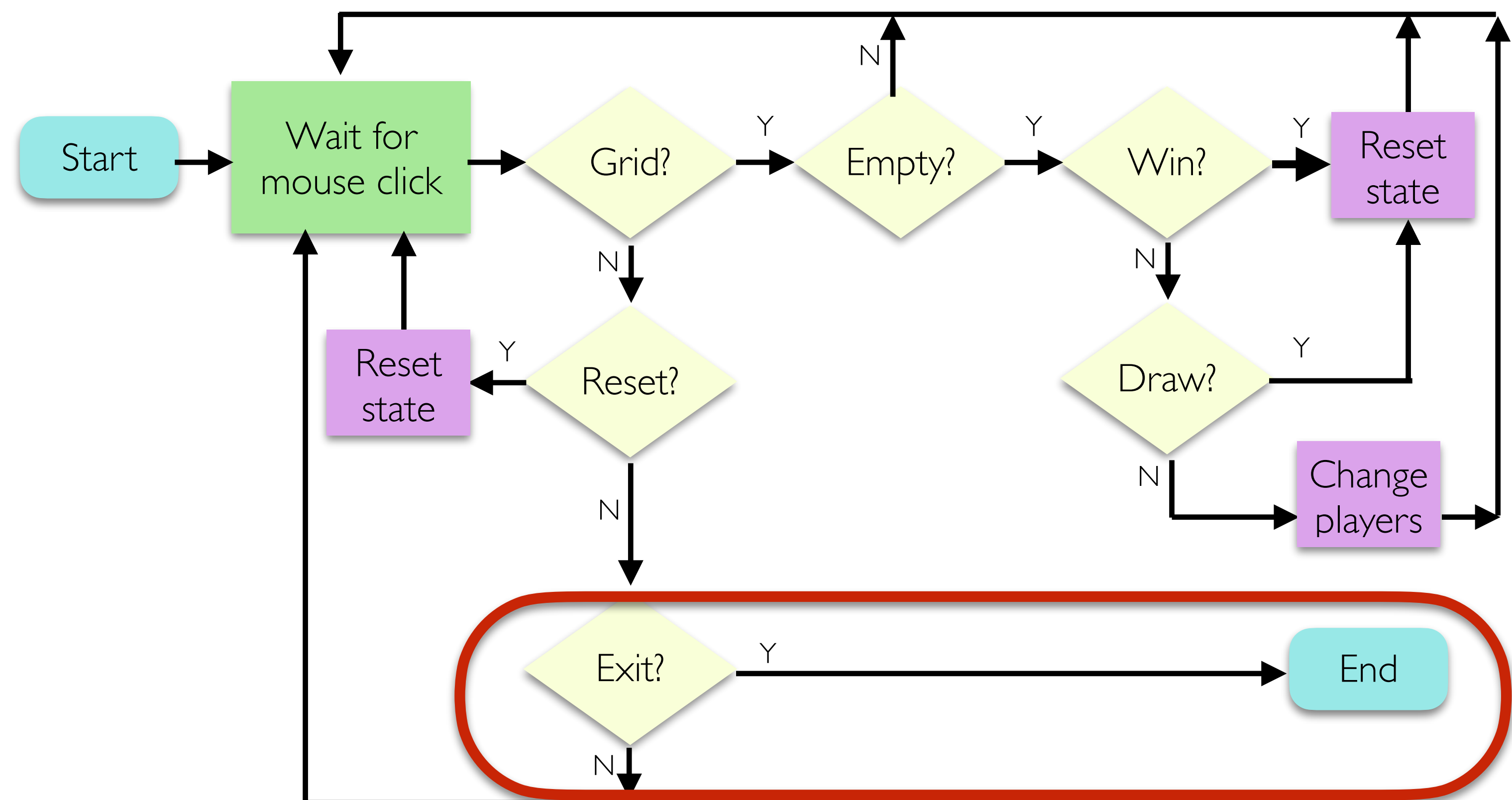
```python
def do_one_click(self, point):

    # step 1: check for exit button
    if self._board.in_exit(point):
      # TODO

    # step 2: check for reset button
    elif self._board.in_reset(point):
      # TODO


    # step 3: check if click on the grid
    elif self._board.in_grid(point):
      # TODO


    # keep going!
    return True
```



Translating our Logic to Code

- Let's handle the "exit" button first (since it's the easiest)

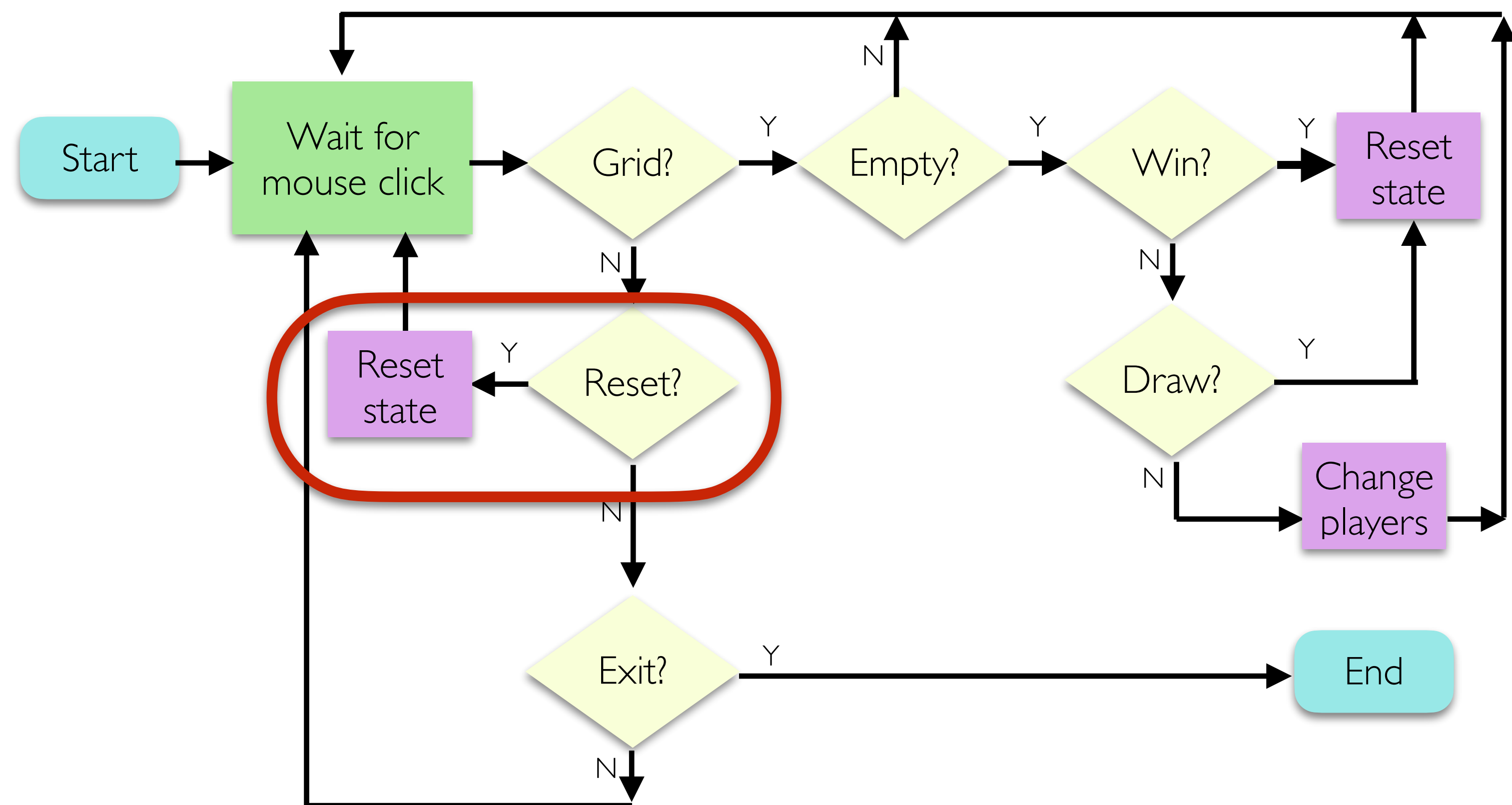
```
if self._board.in_exit(point):
    print("Exiting...")
    # game over
    return False
```



Translating our Logic to Code

- Now let's handle reset

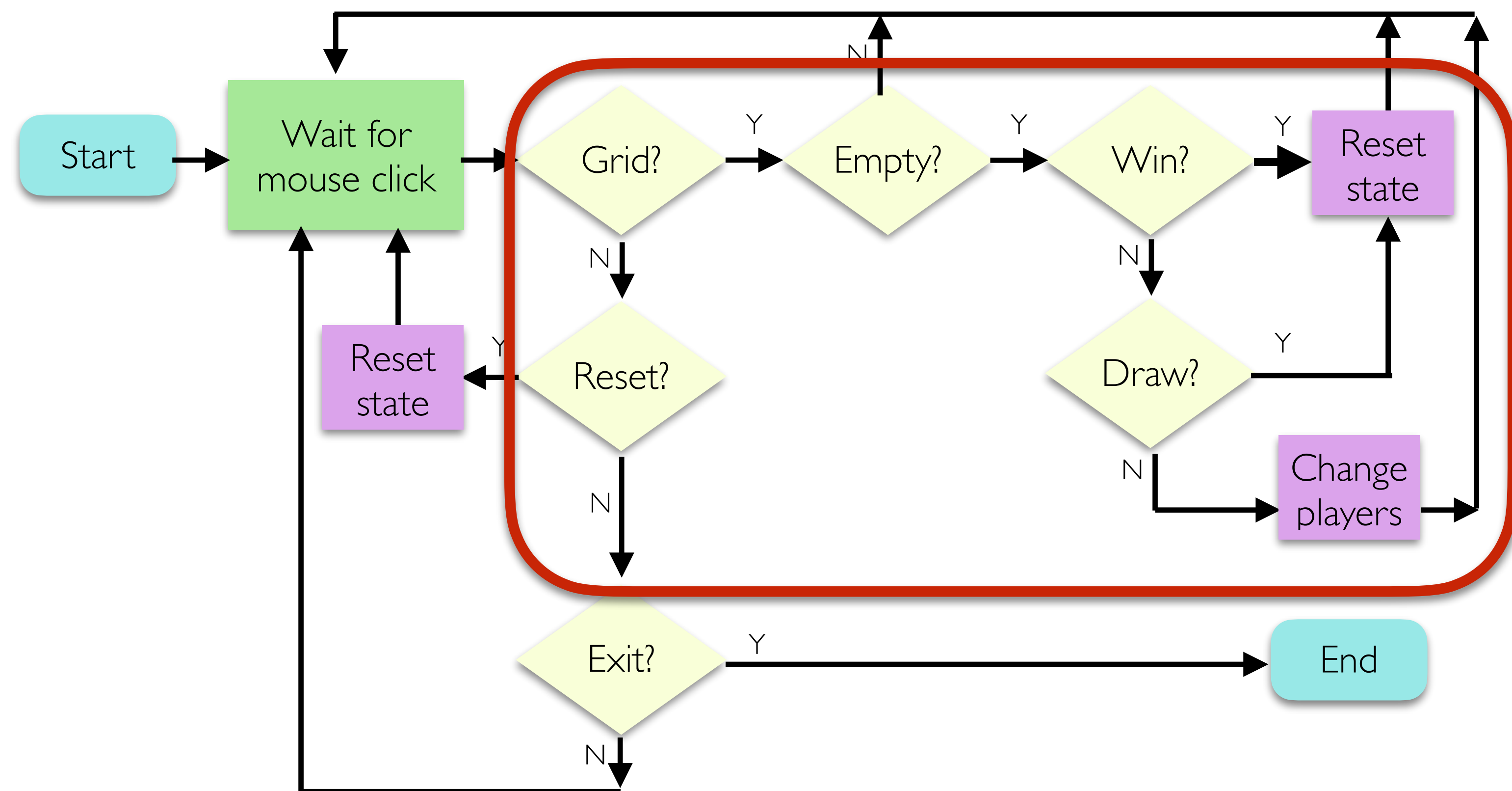
```
elif self._board.in_reset(point):
    print("Reset button clicked")
    self._board.reset()
    self._board.set_string_to_upper_text("")
    self._num_moves = 0
    self._player = "X"
```



Translating our Logic to Code

- Finally, let's handle a "normal" move. Start by getting point and TTTCube

```
elif self._board.in_grid(point):

    # get the cube at the point the user clicked
    tcube = self._board.get_ttt_cube_at_point(point)
```



Translating our Logic to Code

- The rest of our code checks for a valid move, a win, a draw, and updates state accordingly
- At the end, if the move was valid, we swap players

```
elif self._board.in_grid(point):

    # get the cube at the point the user clicked
    tcube = self._board.get_ttt_cube_at_point(point)

    # make sure this square is vacant
    if tcube.get_letter() == "":
        tcube.set_letter(self._player)
        tcube.place_cube(self._board)

        # valid move, so increment num_moves
        self._num_moves += 1

        # check for win or draw
        win_flag = self._board.check_for_win(self._player)
        if win_flag:
            self._board.set_string_to_upper_text(self._player + " WINS!")
        elif self._num_moves == self._board.get_rows()
                                 * self._board.get_cols():
            self._board.set_string_to_upper_text("DRAW!")
        # not a win or draw, swap players
        else:
            # toggle player!
            self._player = "O" if self._player == "X" else "X"

# keep going!
return True
```

TTT Summary

- Basic strategy
  - `Board`: start general, don't think about game specific details
  - `TTTBoard`: extend generic board with TTT specific features
    - Inherit everything, update attributes/methods as needed
  - `TTTCube` isolate functionality of a single TTT cube on board
    - Think about what features are necessary/helpful in other classes
  - `TTTGame`: think through logic conceptually before writing any code
    - Translate logic into code carefully, testing along the way

Boggle Strategies

- At a high level, Tic Tac Toe and Boggle have a lot in common, but the game state of Boggle is more complicated
- In Lab 9 you should follow a similar strategy to what we did with TTT
- ***Don't forget the bigger picture as you implement individual methods***
- Think holistically about how the objects/classes work together
- Isolate functionality and test often (use `__str__` to print values as needed)
- **Discuss logic with partner/instructor before writing any code**
- Worry about common cases first, but don't forget the "edge" cases
- Come see instructors/TAs for clarification

GOOD LUCK and HAVE FUN!