Tic-Tac-Toe

Software Development 2017
Department of Computer Science
University of Copenhagen

Alexander Christensen <alex_c007@hotmail.com>
Oleks Shturmov <oleks@oleks.info>

Version 1;
May 26, 2017

This is a sample technical report for Software Development B3-4, 2017. We designed a small, command-line based Tic-Tac-Toe game in C#, which we describe in this report. You will find the code on our GitLab:

`https://git.dikunix.dk/su17/TicTacToe/tree/v1`

Contents

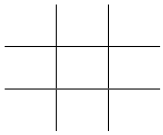
1 Background

Tic-Tac-Toe is a simple, grid-based game, which can be traced back to the Ancient Egyptians. Playing Tic-Tac-Toe requires few interface contraptions: you can play with a friend or foe with sticks in the sand, with pencils on a sheet of paper, or alone against an "impossible" AI in a Google Search doodle[1].

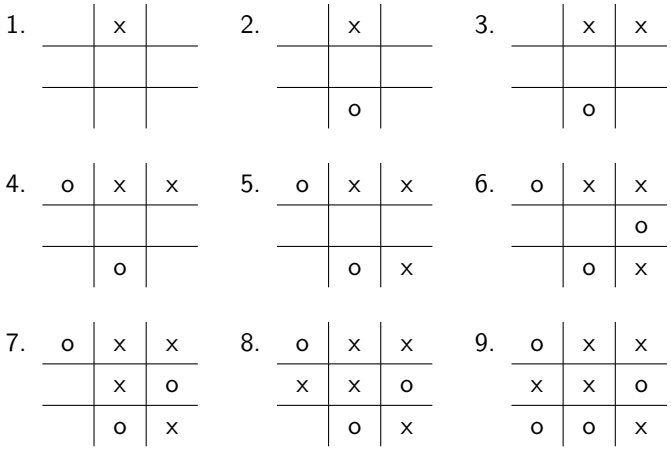
[1] `https://www.google.dk/search?q=tic-tac-toe`.

The background for *our* implementation is to provide a baseline implementation to describe in this report. As such, the implementation should be short and sweet, employing the development environment and programming techniques introduced earlier in this course — the target audience, for our code and report, are the current students on the Software Development course.

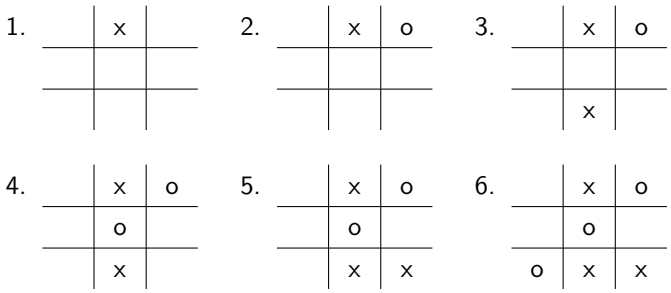
The onset for a game of Tic-Tac-Toe is a 3x3 grid:



Two players then take turns to place a cross (x) or a circle (o), with the objective to get 3 crosses, or 3 circles, in a row, column, or along a diagonal. The first one to do so, wins. For instance, if x begins, the game may develop as follows:



That's a tie; now let's reset and see o win:



The purpose of this project is to enable two humans to compete in a game of Tic-Tac-Toe, playing on the same physical keyboard, on a machine with Mono (version $\geq$ 4.0.0), and MonoDevelop / Xamarin Studio (version $\geq$ 5.9) installed. The purpose is *not* to showcase `DIKUArcade`, the game framework otherwise used throughout the course.

2 Analysis

As this project is not intended to showcase `DIKUArcade`, it does not have to build on `DIKUArcade`. This has the added-in benefit that we reduce the chance of introducing complexities due to the mere use of `DIKUArcade`.

There is also no need to introduce a graphical user interface: DIKU students can be expected to be contempt with a command-line example, and using a command-line interface keeps the implementation simple: there is no need for timers, threads, events, or graphical widgets. In particular, a "frame" of the game can constitute the following:

2.1. Clear the console window.

2.2. Draw the level in ASCII art by writing to the console.

2.3. Position the console cursor in one of the cells, and accept user input.

2.4. Depending on user input, change game state, and draw the next frame.

The game should be a multi-player game, enabling `x`'s and `o`'s to compete on the same machine, using the same physical keyboard. The game play should be quite fast, and both players need to move about the board. It would be annoying for the players to have to lift their fingers.

One way to resolve this conflict is to segregate of the available keys into those for `x`, and those for `o`.

For instance, if it is `x`'s turn, the game may *only* respond as follows:

[A] Move the cursor one cell left, if possible.

[S] Move the cursor one cell down, if possible.

[D] Move the cursor one cell right, if possible.

[W] Move the cursor one cell up, if possible.

[ ] Place an `x` in the current cell, if possible.

Placing a `x` in a cell succeeds *only* if the cell is empty. If an `x` is placed, the turn goes to `o`.

Similarly, if it is `o`'s turn, the game *only* responds to the keys [J], [K], [L], [I], and [ ], where `o` can use the keys [J], [K], [L], [I] to move the cursor similarly to `x`, and use the [ ] key to place an `o` in the current cell, subject to similar constraints.

Other design goals relating to technology stack choices:

2.1. The game should be written in C#, as a MonoDevelop / Xamarin Studio project, version controlled with Git — the technology stack of our course.

2.2. The game should run across the variety of operating systems that one might reasonably expect on our course.

Other design goals relating to coding style:

2.3. The game should be designed in terms of self-aware entities that each are responsible for drawing themselves, and similarly, leave it to the their constituents to draw themselves.

Inevitably, this will lead to a design that is readily portable to a more advanced user-interface framework, such as GtkSharp.

2.4. The game should be designed in terms of methods that each do *one* thing, and do that one thing *well*.

2.5. The game should be designed in terms of classes that each have *one* responsibility, and meet that responsibility well.

2.6. The game should be designed such that variables have the least possible scope, to reduce the complexity the possible *side-effects* of calling any given method.

2.7. The game should be designed in terms of individually testable methods and classes, i.e. *units* suitable for unit-testing.

3 Design

Design goal 2.2. can be met by relying Mono's command-line utilities, we ensure that our Tic-Tac-Toe is cross-platform up to platform-specific differences in these core Mono utilities.

The rest of the design goals are addressed in assorted fashion.

Overall the game is structured as follows:

`Game`
: A singleton class, around which the entire game evolves.

  This class has a non-terminating method `Interact`, which iteratively (1) draws a frame of the game, (2) listens for user input at the console, and (3) upon input, updates the game state, and repeats.

  This class also maintains the current `Board`, `Cursor`, and `Player`, accessible to others via dedicated properties.

`Board`
: A class that represents a Tic-Tac-Toe board. The board is 3x3 by grid of cells, where each cell may be either empty, a cross, or a circle. The board draws itself by iterating over the cells and issuing their draw methods.

  A player can attempt to place a cross or a circle at a given position. If this succeeds, the board will check to see if this leads to a player winning, and if so, updates the state of the board to represent an end-of-game board.

  To start a new game, you must create a new instance of `Board`.

`Cell`
: A cell is an enumerated (as specified above) type, with a draw method.

`Cursor`
: A class that represents the game cursor. In general, exactly one cell is selected in any frame of the game. Hence, a cursor has a position, can be moved around, one cell at a time, and knows how to draw itself.

  To start a new game, you must create a new instance of `Cursor`.

`Player`
: An abstract class with 2 concrete implementations: `Circle` and `Cross`. Players responds to key-strokes (to each their own). Players know which methods of the `Board` or `Cursor` classes to call to get things done.

  Players have no state, and so can be implemented as singletons.

  The method relating to making a game move (i.e., set cross or circle) returns an instance `Player`, yielding to the next player, upon a successful game move.

It was not a design goal to let the game to handle board sizes other than the classical 3x3. It would however be fairly easy to extend this functionality because our game contains not "magical numbers", but instead uses well-named constants such as `Constants.SIZE` which defines the game board to have a width of exactly 3 pieces. In a similar fashion we could also extend the border size of the game area without affecting the game board itself. It would not, however, be a trivial case to let a single game piece fill out more than a single character in the console.

4 Implementation

Our Tic-Tac-Toe is implemented as a single C# Console Project, with an adjacent NUnit Library Project for unit tests (see also Section 6).

The entry point of Tic-Tac-Toe is located in `Program.cs`, but does little else than fetch an instance of `Game`, and call its `Interact` method. The `Game` class is really the epicentre of all the game functionality.

The `Interact` method is fairly straight-forward. There is use of threads, timers, events, or graphical drawing methods, due to the fact that Tic-Tac-Toe is a command-line based game. Overall, the `Interact` method in `Game` looks like this:

```
/// <summary>
/// Game loop. Called from Main method.
/// Starts a game and keeps it running until closed.
/// </summary>
public void Interact()
{
    int key;
    while(_game_running)
    {
        Draw();
        CheckGameRunning();
```

```
        key = Console.Read();
        Respond(key);
    }
}
```

The following, further implementation details are relevant if you would like to dive extend or maintain our code:

`Position`
: This *struct* is used to represent a position throughout the game. It is by-and-large copy-pasted from `DIKUArcade`.

`Constants`
: This *static* class contains a handful important constants related to the layout of the game. This includes the size of the board, as well as information about the borders around the board. This keeps our Tic-Tac-Toe free from "magic constants".

  This design is a bit fragile in the following sense: if you change the drawing methods such that there are more lines or columns before the board is drawn, you have to remember to update these constants.

`Board`
: A Tic-Tac-Toe board is represented by a two-dimensional array of cells in row-major layout. This implementation detail is not important, and is hidden as best as possible.

  For instance, the method `PlayCross` takes in a `Position`, indexes into the two dimensional array, and sends a *reference* to the cell along to method that actually attempts to place a cross. This way, there are only a couple methods that explicitly index the array.

"Draw"
: Despite the design goal that each game entity should know how to draw itself, no general base-class or interface with a method called "Draw" has been implemented. This is because the game is fairly simple, and we would also like to draw such mundane things as *enum*s.

  Enumerations can't have proper methods in C#, only *extension methods* (see also `Cell.cs`), and extension methods cannot be subject to base-class, or interface constraints.

  If such a base-class or interface becomes necessary in the future, all you would have to do is wrap the enumerations (e.g., `Cell`) in proper classes.

4.1 Drawing

To clear the console, we system `System.Console.Clear`. To draw the board, borders, and instructions, we use the methods `System.Console.Write` and `System.Console.WriteLine`. To "draw" the cursor, we use `System.Console.SetCursorPosition`. All this makes the order in which the entities are drawn

important. For instance, the cursor cannot be set before adequate lines have been written. Overall, the `Draw` method of the `Game` class looks like this:

```
/// <summary>
/// Draw the game by calling the constituent draw methods in order.
/// </summary>
private void Draw()
{
    Console.Clear();
    DrawTitle();
    DrawHelp();

    CheckGameRunning();

    Board.Draw();
    Cursor.Draw();
}
```

5 User Guide

The game-play should be fairly self-explanatory at this point. However, our application will show the controls currently available to each player when it is their turn. Furthermore, there is support for the following keys:

`r` Restart the game.

`q` Quit the game.

To run Tic-Tac-Toe, clone our project (see URL on the front page), open `src/TicTacToe.sln` in MonoDevelop / Xamarin Studio, and run the Console Project. You can also build and run from the command-line:

```
$ xbuild /p:Configuration=Release src/TicTacToe.sln
$ mono src/TicTacToe/bin/Release/TicTacToe.exe
```

6 Evaluation

From manually running and testing the game, it seems to work as intended, and seems to be fairly robust. To convince ourselves further of the quality of our implementation, and to support further developments of the game, we have also implemented unit tests for all non-trivial parts of the application.

You will find the tests under the `TicTacToeTests` solution, which uses the following additional packages:

- NUnit (version = 3.6.1)
- FsCheck (version = 2.9.0)

- FSharp.Core (version = 4.1.17)

Our project on GitLab is also configured to run the tests every time we push, ensuring a simple kind of continuous integration.

The tests cover:

- Board-related operations: ensuring that a winner is called as soon as there are 3 crosses, or 3 circles, in a row, column, or along a diagonal, and that the winner stays put in the light of further board operations.
- Cursor-related operations: ensuring that the cursor can be moved around the whole board, and not beyond it.
- Initial game state: checking the initial cursor position and board state.
- Game-play: by emulating key strokes.

In the last two cases, as a side-effect, we also test the game restart functionality. This is used in the test-method-level set-up methods, where we restart the game, to ensure that no previous game state interferes with the given test.

This makes the tests fairly covering (although formal coverage analysis remains to be done), and there are both *positive* and *negative* tests (testing the intended functionality, as well as unintended, but possible cases, respectively). We are hereby fairly confident in the quality of our application.

One of our design goals was to have a clear distinction between the responsibilities of the different classes — a central feature of our design is a high adherence to The Single-Responsibility Principle [1]. This enabled us to test the individual units of game functionality without always having to spawn an entire game instance to do so.

7 Conclusion

We have designed a fairly simple application for showcasing in this technical report. The Tic-Tac-Toe game compiles, runs, and plays well.

The application had no up-front specification, and so is trivially complete. However, given more time, the next things to tackle would be an AI to play against.

The application is fairly well-structured, and we have done our best effort to refactor the code before a final release. The application is well-tested, and the tests showcase that the application is fairly robust. The tests are part of a continuous integration setup on our GitLab, so the application is ready to extensions and maintenance by other developers.

References

[1] Robert C Martin and Micah Martin. *Agile principles, patterns, and practices in C*. Pearson Education, 2006. 8