

# Final Programming Project: Tic Tac Toe

Tesini Simone  
UniBZ

24 December 2025

## 1 Introduction

### 1.1 Project overview

The requirements for this assignment were:

- Tic tac toe game playable in a terminal interface.
- Accept user input to play or quit the game, and also show instructions if asked.
- During the game, on each turn, ask the user for input on where to place their mark.
- **“Normal”** computer player that selects random moves, described in [Section 3.1](#).
- **“Smart”** computer player that learns and avoids losing moves, described in [Section 3.2](#).
- Accept flags when executing the program to choose the kind of computer player to play against. See [Section 1.2](#) for more.

All of these requirements were implemented. In addition, I also implemented:

- **Colorful and friendly** user interface using ANSI codes and Box drawing characters
- **“Memory smart”** computer player, an extension of the “smart” computer, that also saves losing combinations in a file, so that it can remember them in future games. Described in [Section 3.3](#).
- **“Very smart”** computer player, that uses a custom algorithm to always pick the best move. It tries to always reach at least a draw, making it practically unbeatable. Described in [Section 3.4](#).
- Additional CLI flags for new kinds of computer players.
- Ability to specify two CLI flags to let two computers play against eachother.

### 1.2 CLI Usage

If you haven’t specified any flags when starting the program, this will make 2 humans play together. You can use flags when running the program to choose a computer player (*“bot”*) to play against:

- -n: human vs. normal bot
- -s: human vs. smart bot
- -m: human vs. smart memory bot
- -S (capital S): human vs. very smart bot

You can also specify a percentage when playing against the very smart bot, by following the -S with a percentage like: -S-40 to get 40% difficulty.

You can also let two bots play against eachother, by specifying two flags. For example, -n -S-40, would pit a normal bot against a very smart bot at 40% difficulty.

## 2 Design choices

Since I wanted to implement multiple algorithms for **computer players** (“bots”), I started right away by using an interface that all types of players would have to implement.

Players need to implement, along with a few helper methods, a `nextMove` method, that should handle:

1. Input for human players
2. Calculations for all kinds of bots

and then return a `Move` object containing the row and column pair, and whether to surrender or not.

This design choice makes it **very easy** to implement new kinds of bots as it decouples the bot's behaviour from the way that the `Game` class manages the game flow (turns, checking wins and draws, etc.).

## 3 Algorithms for bots

### 3.1 “Normal” algorithm

This was the first algorithm that was requested in the specification.

It works by simply picking a random cell from the  $3 \times 3$  grid.

### 3.2 “Smart” algorithm

This was the second algorithm that was requested in the specification.

It works by “intercepting” loss events (called `Situations` internally), and then saving the grid without the last move in an `ArrayList` of grids.

Then when the bot has to play a turn, it will pick a random move, but if that move would bring it to one of the losing combinations, it will try another random move.

If all possible moves would lead to a losing combination, the bot will just surrender.

### 3.3 “Memory Smart” algorithm

This is an algorithm that I decided to add as I found that the smart algorithm took too many matches to get going, so I decided to implement this algorithm that works the exact same way as the smart one, but it also saves the losing combinations to a file.

Internally it is implemented by extending `SmartBotPlayer`:

1. on loss, save the losing combinations to an `ArrayList` AND save it to a file.
2. when constructing an instance of `MemorySmartBotPlayer`, load the file, parse it, and add the losing combinations to the `ArrayList`.

Each line of the file represents a losing combination, represented as 9 characters, one for each cell.

An example losses file can look like:

```

0 X0X
X 0X 0
X 0 X X00
X0X X 00
X X0X0 0
X X0X00
X X00X0
X 0 XX0 0
X 0 X 0X0

```

### 3.4 “Very smart” algorithm

The last algorithm I implemented is very interesting. Instead of remembering losing combinations, it dynamically picks the best possible move on every turn.

At 100% difficulty, it is unbeatable. I never managed to win against it, and I even tried doing 5000 matches against a normal mode bot and the very smart bot never lost.

It works by checking each of these conditions in order:

0. Based on the difficulty, randomly choose if we should actually perform the best move or a random one. On 100% difficulty it will always choose the best move, on 0% difficulty it is essentially the same as playing against the normal algorithm.
1. Is there a row, column or diagonal, where placing a mark would either make me win, or prevent the opponent from winning? if so, place a mark there.

For example on this grid (X = human, 0 = very smart bot, . = empty):

```

X.0
.X.
.0.

```

The algorithm would pick the bottom right cell to prevent X from winning.

2. Is the center cell empty? If so, place a mark there. The center cell is strategically the best cell to occupy.
3. Is any corner empty? If so, place a mark there.
4. If no other best move is possible, place a mark on a random cell.

## 4 Conclusion

I implemented a complete terminal Tic Tac Toe game with a modular design for players, thanks to Java’s interfaces.

All requirements were fulfilled, along with some additional features, including a dynamic algorithm which was fun to implement.

Some additional improvements that could be done in the future:

- GUI frontend for the game
- Networked multiplayer (human versus human)
- Even smarter bot using the *minimax* algorithm (<https://www.geeksforgeeks.org/dsa/finding-optimal-move-in-tic-tac-toe-using-minimax-algorithm-in-game-theory>)

All code can be found on my github: <https://github.com/Tesohh/UniBZ-TicTacToe>

## 5 References

Some references were used during the making of the project. **All code and ideas are original.**

- Box drawing characters: [https://en.wikipedia.org/wiki/Box-drawing\\_characters](https://en.wikipedia.org/wiki/Box-drawing_characters)
- ANSI color codes: <https://gist.github.com/JBlond/2fea43a3049b38287e5e9cefc87b2124>
- Java records: <https://www.baeldung.com/java-record-keyword>
- Appending a line to a file: <https://www.baeldung.com/java-write-to-file>