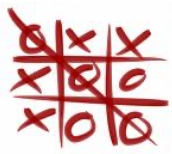
Tic-Tac-Toe Kata



The exercise involves developing a Tic-Tac-Toe game strictly adhering to the TDD rules.

Tic Tac Toe - Game Rules:

https://en.wikipedia.org/wiki/Tic-tac-toe

UAT Scenarios:

1 – Game Board Creation phase:

```
Game Board Creation…
 | |
-+-+-
 | |
-+-+-
 | |

Board Created.
The game will start with player X
```

2 – Player X won with a vertical line

```
Player X:
X| |
-+-+-
X|O|
-+-+-
X| |O

PLAYER X WON!
```

3 – Player O won with a horizontal line

```
Player O:
X| |X
-+-+-
O|O|O
-+-+-
X| |

PLAYER O WON!
```

4 – Player X won with a diagonal line

```
Player X:
X| |
-+-+-
O|X|
-+-+-
O| |X

PLAYER X WON!
```

5 – Game ends with a draw

```
Player X:
X|O|X
-+-+-
O|O|X
-+-+-
X|X|O

GAME ENDS WITH A DRAW!
```

Kata Objective:

The system should run in BOT mode (random BOT moves for player X & O) to print on the screen all the player's moves (with a 2 seconds timeout between each round) until someone won or the game ends with a draw.

Graduation test scoring system:

The kata must be stored in GitHub. The first push must be an empty directory. This push will start the graduation test. It will be scored with following rules:

- ⇒ Done in Solo mode (NO PAIR - NO MOB – NO copy & paste)
- ⇒ Timebox: 2 hours -- 4 pomodori from the first git push (the empty repository)
- ⇒ Notes as in pair programming for every pomodoro (even doing it in solo)
- ⇒ For every 'pomodoro cycle' commit the NOTES.md to show the goal for every time slot
- ⇒ NOTES.md must show your simple design approach and how you organised the code growth between features VS tech debt and refactor.
- ⇒ Simple design & emerging architecture approach (no big thinking upfront ;)
- ⇒ the code has to be done in a strict TDD way, via cycles of Red/Green/Refactor
- ⇒ commit any Red-Green-Refactor cycle to have readable history in git
- ⇒ 100% code coverage
- ⇒ White belt Refactoring pillars:
  - ⇒ 1 – the test suite looks like a book that explains the Tic Tac Toe game (DDD vocabulary)
  - ⇒ 2 – the code and test use the same test suite wording (DDD vocabulary)
  - ⇒ 3 – the code and the test are readable like a book (hidden behavioral complexity)
- ⇒ Last but not least… **the code must be working software**.