# Learning TDD for the second time

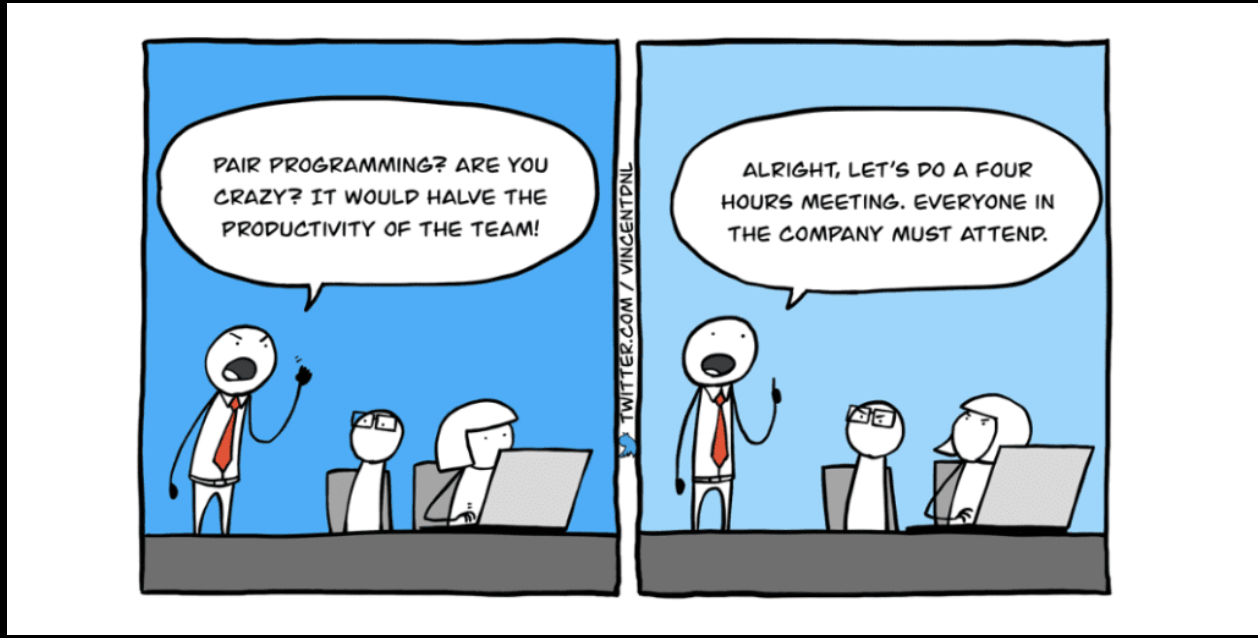
## my key takeaways from learning to walk with a mob

Elisabeth Forland – elisabeth.forland@bouvet.no

Bouvet

# Background

- Worked as a developer for 19 years
- First experience with TDD in 2007
- Mostly learnt from internal presentations and coding dojos at work
- The degree of automated testing has been varying depending on project
- Eager to «get back into it»



# Mob programming

- Navigator – the person making the final descision on what to do next, the person the driver should be listening to.
- Driver – the person typing, not required to think

- Mob – everyone else in the room. They observe and discuss with the navigator the way forward.

# Red – Green – Refactor

- Red: write a failing test ( and it should fail for the right reasons!)
- Green: write code to make the test pass
- Refactor: when all tests are green, you are allowed to refactor

- Rule of Three:
- Extract duplication only when you see it for the third time!

# Good habits are important!

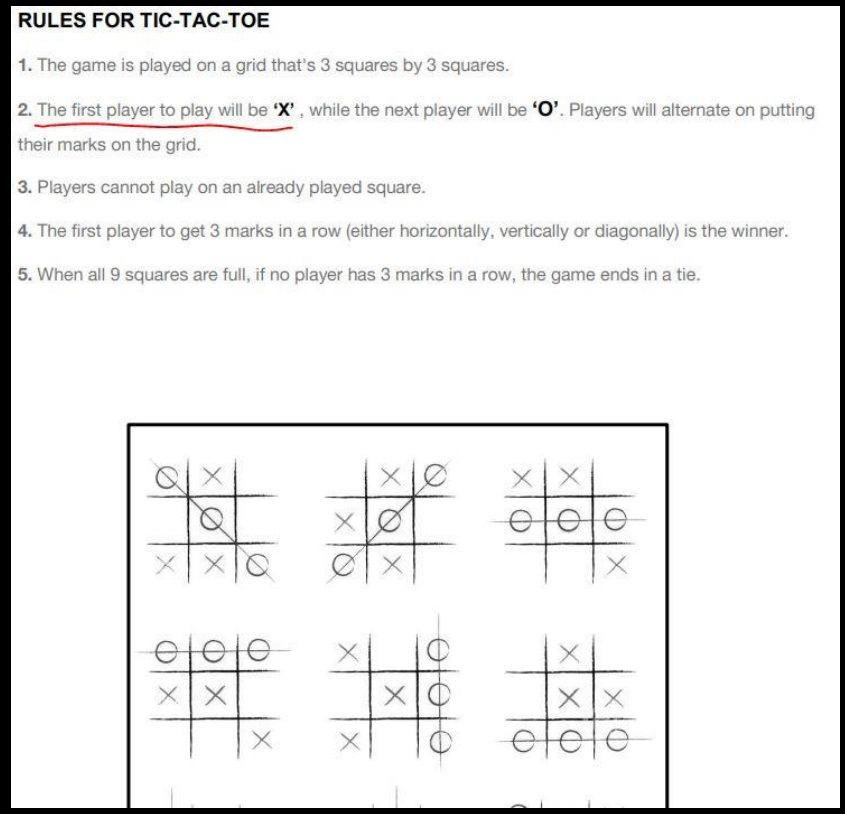
- Commit often
- Refactor aggressively
- But only refactor when all tests are green!

# Test behaviour – not implementation

- Focus on finding the behaviour, starting with the simplest one.
  - When given a task I tend to think out the entire implementation in my head first, so changing the focus will need some practicing.

- Finding the behaviour can be tricky, often we will find restrictions or requirements first and focus on them instead of looking for the behaviour

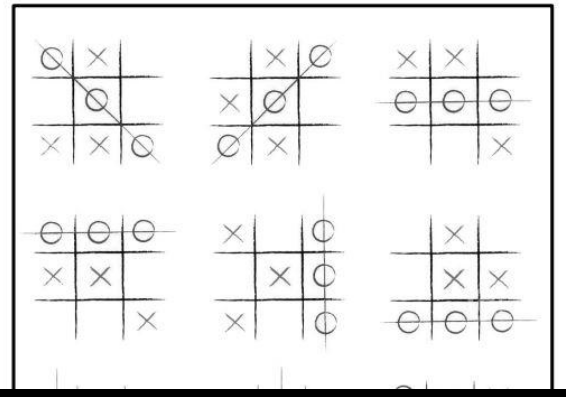
- In the Tic Tac Toe, we focused on having a board with 9 squares instead of what happens when you first start the game.

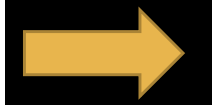


# Find the most basic behaviour

**RULES FOR TIC-TAC-TOE**

1. The game is played on a grid that's 3 squares by 3 squares.
2. The first player to play will be **'X'** , while the next player will be **'O'**. Players will alternate on putting their marks on the grid.
3. Players cannot play on an already played square.
4. The first player to get 3 marks in a row (either horizontally, vertically or diagonally) is the winner.
5. When all 9 squares are full, if no player has 3 marks in a row, the game ends in a tie.





```
[TestFixture]
public class TicTacToeOCShould
{

    [Test]
    public void MakeFirstPlayerX()
    {
        TicTacToeOC ticTacToeOC = new TicTacToeOC();

        var _currentPlayer = ticTacToeOC.CurrentPlayer();

        Assert.That(_currentPlayer, Is.EqualTo(Player.X));
    }

    [Test]
    public void MakeNextPlayerO()
    {
        TicTacToeOC ticTacToeOC = new TicTacToeOC();
        ticTacToeOC.PlaceMark(Position.UpperLeft);

        var _currentPlayer = ticTacToeOC.CurrentPlayer();

        Assert.That(_currentPlayer, Is.EqualTo(Player.O));
    }
```

# Walking backwards

- Start with writing the assertions in the test. It helps you focus on what you want to test and keep focus on that.
- Write more and more specific tests to drive the design forward. The design might end up completely different from what it would have done if you didn’t.
- as you add more tests, it becomes easier to see the patterns

# Don’t run to fast

- It is often tempting to do too much at once:
  - Don't refactor on red tests
  - Don't remove code when refactoring until you have written the refactored code. You might need it to see what your starting point was. (unless using resharper to extract, etc.)
  - Write the simplest implementation first. If you do not know the implementation, use fake implementation.
  - Use triangulation: don't refactor until you see the pattern, typically after 3 occurrences of the same code.
  - Remember to commit on green tests! (this is something I often forget!)

# Transformation Priority Premise

- Start simple:
- 1. Fake implementation – hard code to make test pass
- 2. Obvious implementation – when you are sure about the code you need to write
- 3. Triangulation with the next test – when you are not sure about the pattern or how to generalize a behaviour.
  - Start simple, add more tests -> this will force the code to become more generic

- Use the Transformation Priority Table if stuck!

# Using the TPP table

Often, we think to compex

- Keeping the table next to you when programming helps remind you that you should start small, and what steps you can try to use before going to more complex implementation techniques.
- In the Arabic to Roman converter kata, I probably wouldn't have thought of using a table in the way we ended up.

## Transformation Priority Cheatsheet

When coding, prioritise the simple transformations at the top of the list over the more invasive transformations at the bottom.

| | Transformation Name | Example |
|---|---|---|
| 1 | *no code* -> nil | *no code* -> nil |
| 2 | nil -> simple constant | nil -> "", 1 |
| 3 | constant -> constant+ | "", 1 -> "Australia", 42 |
| 4 | constant -> variable | "Australia" -> country |
| 5 | statement -> statements | country = "Australia" -> country = "Australia"; puts "G'Day!" |
| 6 | unconditional -> conditional | *no if* -> if, unless, ternary if statement |
| 7 | variable -> array | dog -> [dog, cat] |
| 8 | array -> collection | [dog, cat] -> {dog: 'woof', cat: 'meow'} |
| 9 | statement -> recursion | statement -> recursion(recursion) |
| 10 | if -> while/loop | if door_open? -> while(door_open?) |
| 11 | expression -> method call | Date.today - birthdate -> age_in_days(birthdate) |
| 12 | variable -> assignment | days_left -> days_left = 10 |

Transformation Priority Premise concept by @unclebobmartin. Cheatsheet design and examples by @evolve2k

# Be smart - but not too smart

- In the Arabic to Roman converter, we decided to keep IV=4 in the dictonary
- it did not make a big impact to keep numbers like that in the dictionary, and it would complicate the code a lot to try to calculate it instead.
- Normally I would probably spend a lot of time trying to get it out of there.


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| I | IV | V | IX | X | ... | ... | ... |

# If your lost

- Its ok to loose track of what you are doing when mob programming, but it is not ok to leave it like that. You should ask questions until you are up to speed with the others.
- Also, your questions might result in new ideas or that the mob realizes that they are on the wrong track.
- OR that you were just lost xD

# Thanks for listening!
# Questions?


Elisabeth Forland – elisabeth.forland@bouvet.no

Bouvet