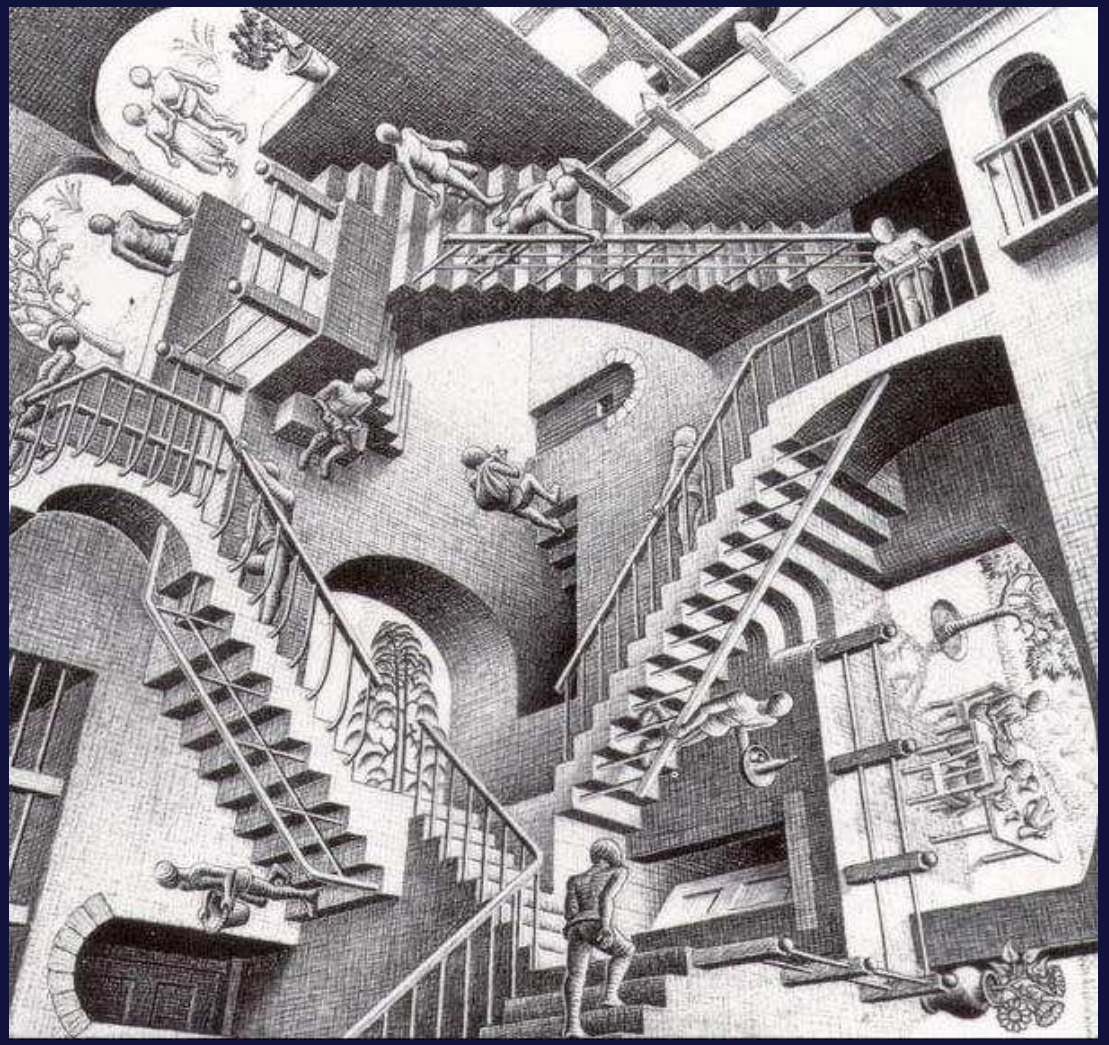
Test Driven Development

Design by choice, not by accident



Jacob Holm

Transformation Priority Premise

- Fake implementation
  - Just enough to pass the test
- Obvious (simple) implementation
  - Code evolutions
- Triangulation
  - Test-diversity

Transformation Priority Premise - What is "Obvious implementation"?

#	TRANSFORMATION	STARTING CODE	FINAL CODE
1	{} => nil		`return nil`
2	nil => constant	`return nil`	`return "1"`
3	constant => constant+	`return "1"`	`return "1" + "2"`
4	constant => scalar	`return "1" + "2"`	`return argument`
5	statement => statements	`return argument`	`return arguments`
6	unconditional => conditional	`return arguments`	`if(condition)return arguments`
7	scalar => array	`dog`	`[dog, cat]`
8	array => container	`[cog, cat]`	`{dog = "DOG", cat = "CAT"}`
9	statement => recursion	`a + b`	`a + recursion`
10	conditional => loop	`if(condition)`	`while(condition)`
11	recursion => tail recursion	`a + recursion`	`recursion`
12	expression => function	`today - birthday`	`CalculateAge()`
13	variable => mutation	`day`	`var day = 10; day = 11;`
14	switch case		

Code evolution - TicTacToe

```
public class TicTacToe
{
    private string _currentPlayer = "X";
    private int _prevX = -1;
    private int _prevY = -1;

    public string GetCurrentPlayer()
    {
        return _currentPlayer;
    }

    public void Place(int x, int y)
    {
        if (_prevX == x && _prevY == y)
        {
            return;
        }

        _prevX = x;
        _prevY = y;

        if (_currentPlayer == "O")
        {
            _currentPlayer = "X";
        }
        else
        {
            _currentPlayer = "O";
        }
    }
```

```
public class TicTacToe
{
    private string _currentPlayer = "X";
    private readonly List<string> _previousMoves = new List<string>();

    public string GetCurrentPlayer()
    {
        return _currentPlayer;
    }

    public void Place(int x, int y)
    {
        if(_previousMoves.Contains($"{x}{y}"))
        {
            return;
        }

        _previousMoves.Add($"{x}{y}");

        if (_currentPlayer == "O")
        {
            _currentPlayer = "X";
        }
        else
        {
            _currentPlayer = "O";
        }
    }
}
```

```
public class TicTacToe
{
    private string _currentPlayer = "X";
    private readonly Dictionary<string, string> _positionToPlayer = new Dictionary<string, string>();

    public string GetCurrentPlayer()
    {
        return _currentPlayer;
    }

    public void Place(int x, int y)
    {
        if (_positionToPlayer.ContainsKey($"{x}{y}"))
        {
            return;
        }

        _positionToPlayer.Add($"{x}{y}", _currentPlayer);

        if (_currentPlayer == "O")
        {
            _currentPlayer = "X";
        }
        else
        {
            _currentPlayer = "O";
        }
    }
```

Object Calisthenics

- Simple & effective
- Reduce complexity & Improve readability
  - Don’t use the ELSE keyword
  - Only one level of indentation per method
- Promote encapsulation & decoupling
  - Wrap all primitives
  - First class collections
  - No getters/setters/properties
  - Only one dot per line ~~`dog.Body.Tail.Wag()`~~ `=> dog.ExpressHappiness()`

Wrap primitives & readability

Before

```
public class TicTacToe
{
    private string _currentPlayer = "X";
    private readonly Dictionary<string, string> _positionToPlayer = new Dictionary<string, string>();

    public string GetCurrentPlayer()
    {
        return _currentPlayer;
    }

    public void Place(int x, int y)
    {
        if (_positionToPlayer.ContainsKey($"{x}{y}"))
        {
            return;
        }

        _positionToPlayer.Add($"{x}{y}", _currentPlayer);

        if (_currentPlayer == "O")
        {
            _currentPlayer = "X";
        }
        else
        {
            _currentPlayer = "O";
        }
    }
```

After

```
public sealed class TicTacToeOC
{
    private Player _currentPlayer = X;

    private readonly Board _board = new Board();

    public Player GetCurrentPlayer()
    {
        return _currentPlayer;
    }

    public void Place(Position position)
    {
        _board.MarkAt(position, _currentPlayer);

        AlternatePlayer();
    }

    private void AlternatePlayer()
    {
        if (_currentPlayer == O)
        {
            _currentPlayer = X;
            return;
        }

        _currentPlayer = O;
    }
```

Wrap primitives & rule of three

Before

```
34          public string Winner()
35          {
36              if (_positionToPlayer.ContainsKey("00") &&
37                  _positionToPlayer.ContainsKey("01") &&
38                  _positionToPlayer.ContainsKey("02") &&
39                  _positionToPlayer["00"] == _positionToPlayer["01"] &&
40                  _positionToPlayer["00"] == _positionToPlayer["02"])
41              {
42                  return _positionToPlayer["00"];
43              }
44
45              if (_positionToPlayer.ContainsKey("10") &&
46                  _positionToPlayer.ContainsKey("11") &&
47                  _positionToPlayer.ContainsKey("12") &&
48                  _positionToPlayer["10"] == _positionToPlayer["11"] &&
49                  _positionToPlayer["10"] == _positionToPlayer["12"])
50              {
51                  return _positionToPlayer["10"];
52              }
53
54              if (_positionToPlayer.ContainsKey("20") &&
55                  _positionToPlayer.ContainsKey("21") &&
56                  _positionToPlayer.ContainsKey("22") &&
57                  _positionToPlayer["20"] == _positionToPlayer["21"] &&
58                  _positionToPlayer["20"] == _positionToPlayer["22"])
59              {
60                  return _positionToPlayer["20"];
61              }
62
63              return string.Empty;
64          }
```

1
2
3

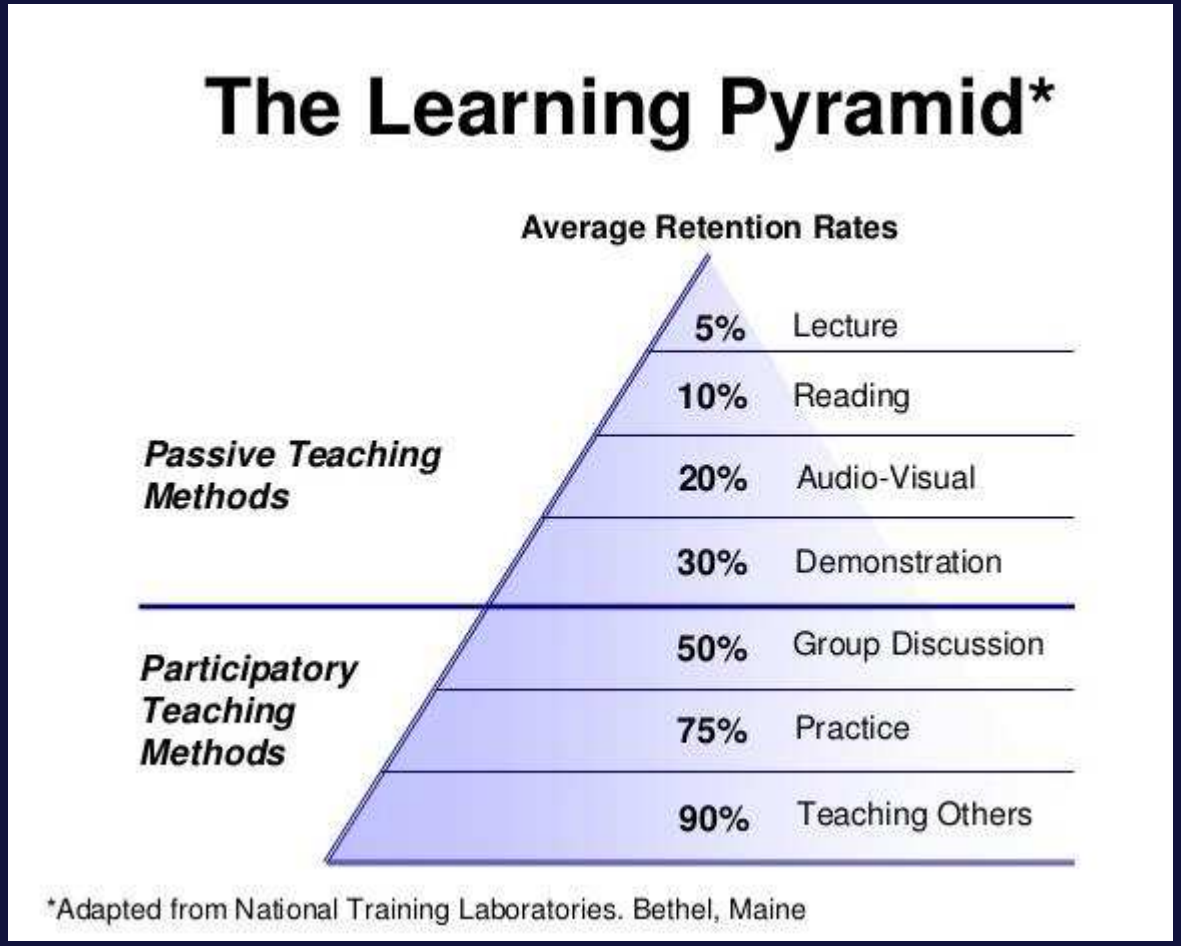
After

```
37          internal Player FindWinner()
38          {
39              var winner = WinnerInLine(TopLeft, TopMiddle, TopRight);
40              if (winner != None)
41              {
42                  return winner;
43              }
44
45              winner = WinnerInLine(MiddleLeft, MiddleMiddle, MiddleRight);
46              if (winner != None)
47              {
48                  return winner;
49              }
50
51              winner = WinnerInLine(BottomLeft, BottomMiddle, BottomRight);
52              if (winner != None)
53              {
54                  return winner;
55              }
56
57              return winner;
58          }
59
60          private Player WinnerInLine(Position first, Position second, Position third)
61          {
62              if (_positionToPlayer[first] == _positionToPlayer[second] &&
63                  _positionToPlayer[first] == _positionToPlayer[third] &&
64                  _positionToPlayer[first] != None)
65              {
66                  return _positionToPlayer[first];
67              }
68
69              return None;
70          }
```

1

Collaborative programming

- Mob/pair programming
- Retention
- Common understanding



Takeaways

- Design by choice, not by accident
  - Don’t assume too much
  - Don’t write more code than you have to
- Test behavior, not implementation
  - Test a requirement, not a new class or method
  - Behavior stays the same, implementation changes
- Readability
  - Tests should read like documentation
- Assert.AreEqual(«Improved», Alcor.GetTDDSkills(«Jacob»))
  - Passed

Thank you!

Keep in touch

jacobholm@gmail.com

Jacob Holm