So, what’s up with TDD?

By Wilhelm Vold

- Outline the specification of the feature
- Write implementation
- Write integration tests
- Does it work? Sure does

- Whats the caviats of it? I am writing answers without asking the questions

How I develop day-to-day

- Ask many questions, and answer them one by one

- Writing tests defines help you define what you are trying to achieve before achieving it
- Do you run a marathon and then spend time training for it? Probably a bad idea

TDD – a better approach

So, how does it really work?

1

Write a test that will fail – RED

2

Write enough code so it passes - GREEN

3

Improve the code, while still passing the tests – REFACTOR

Move forward efficiently

- Fake implementation
- Implement the obvious
- Add tests and generalize your code - Triangulation

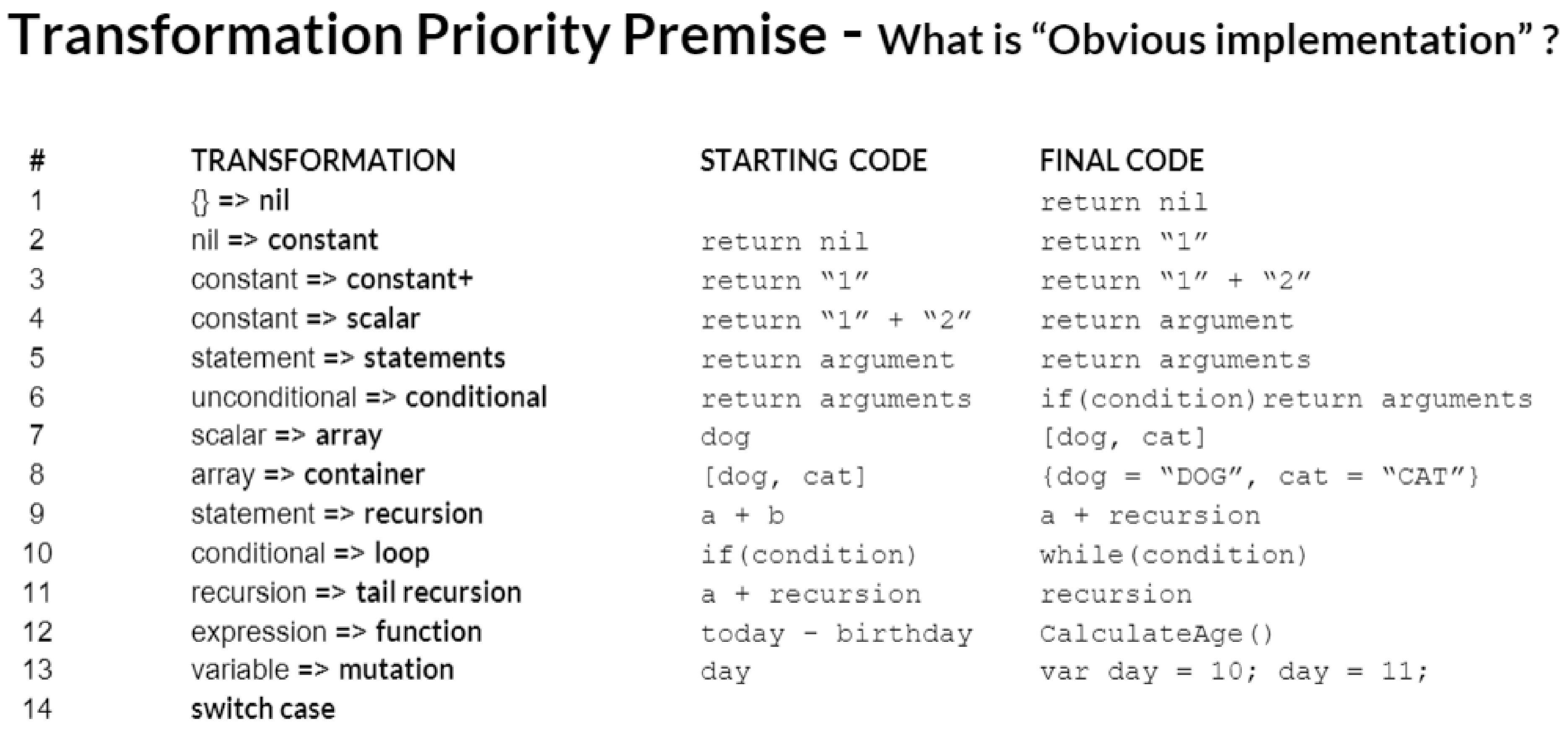
So, why should I use it?

- There are some benefits
- Evolving Design – flexible, maintainable and clean
- Documentation – you know exactly what is happening and how to do it
- The easiness of debugging – no need to Console.WriteLine() everywhere to find out what does not work. You know exactly what does not work
- Its not that scary to change code when you get feedback on what breaks

What if I get lost?

- No stress, there are paths that will lead you to success
- Transformation Priority Premise – evolution of the code from simplest to more complex
- Object Calisthenics – the rules that will make TDD for OOD easier

Transformation Priority Premise



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**	`[dog, 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**		

Object Calisthenics rules

1. Only one level of indentation per method
2. Don't use the ELSE keyword
3. Wrap all primitives and strings
4. First class collections (wrap all collections)
5. Only one dot per line `~~dog.Body.Tail.Wag()~~ => dog.ExpressHappiness()`
6. No abbreviations
7. Keep all entities small
   [10 files per package, 50 lines per class, 5 lines per method, 2 arguments per method]
8. No classes with more than two instance variables
9. No public getters/setters/properties
10. All classes must have state

Just a beginning…

NOT JUST A PRACTICE, BUT A MENTALITY

MUCH MORE TO LEARN…