Transformation Priority Premise

“As tests grow more specific, the code grows more generic.”

Uncle Bob

How to evolve the code in TDD

1) **Fake implementation**
   When you hard code exactly the value you need to pass the test

2) **Obvious implementation**
   When you are sure of the code you need to write. This is what you will be using more often to move forward quickly.

3) **Triangulation with the next test**
   When you want to generalize a behaviour but are not sure how to do it. Starting with fake implementation and then adding more tests will force the code to be more and more generic. Complete one dimension first and then move on the next one with another test case.

Transformation Priority Premises

What is “Obvious implementation” ?

- ➔ The table on the next slide shows a list of code evolutions **ordered by complexity**.
- ➔ Transformations on the **top of the list** should be preferred to those that are lower.
- ➔ When making a test pass, you should try to do so with **simpler transformations** (higher on the list) than with those that are more complex.

* Please do not take this table literally. This is a starting point. Adapt this table to your language and environment.

Transformation Priority Premises - 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` / **basic ops (split, add …)**
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 => **tail recursion**	`a + b`	`a + recursion`
10	conditional => **loop**	`if(condition)`	`while(condition)`
11	tail recursion => **full recursion**	`a + recursion`	`recursion`
12	expression => **function**	`today - birthday`	`CalculateAge()` / **algorithm, library**
13	variable => **mutation**	`day`	`var day = 10; day = 11;`
14	**switch case**	**may be better to use a simpler solution starting from top again**	

Roman numbers kata

Given a positive integer number write a function returning its Roman numeral representation as a String.

Examples:

1=> I
2=> II
3=> III
4=> IV
5=> V
6=> VI
7=> VII
8=> VIII
9=> IX
10=> X
20=> XX
30=> XXX
40=> XL
50=> L
60=> LX
70=> LXX
80=> LXXX
90=> XC
100=> C
200=> CC
300=> CCC
400=> CD
500=> D
600=> DC
700=> DCC
800=> DCCC
846=> DCCCXLVI
900=> CM
1000=> M
1999=> MCMXCIX
2008=> MMVIII