TDD and following the Dimension

Test Driven Development

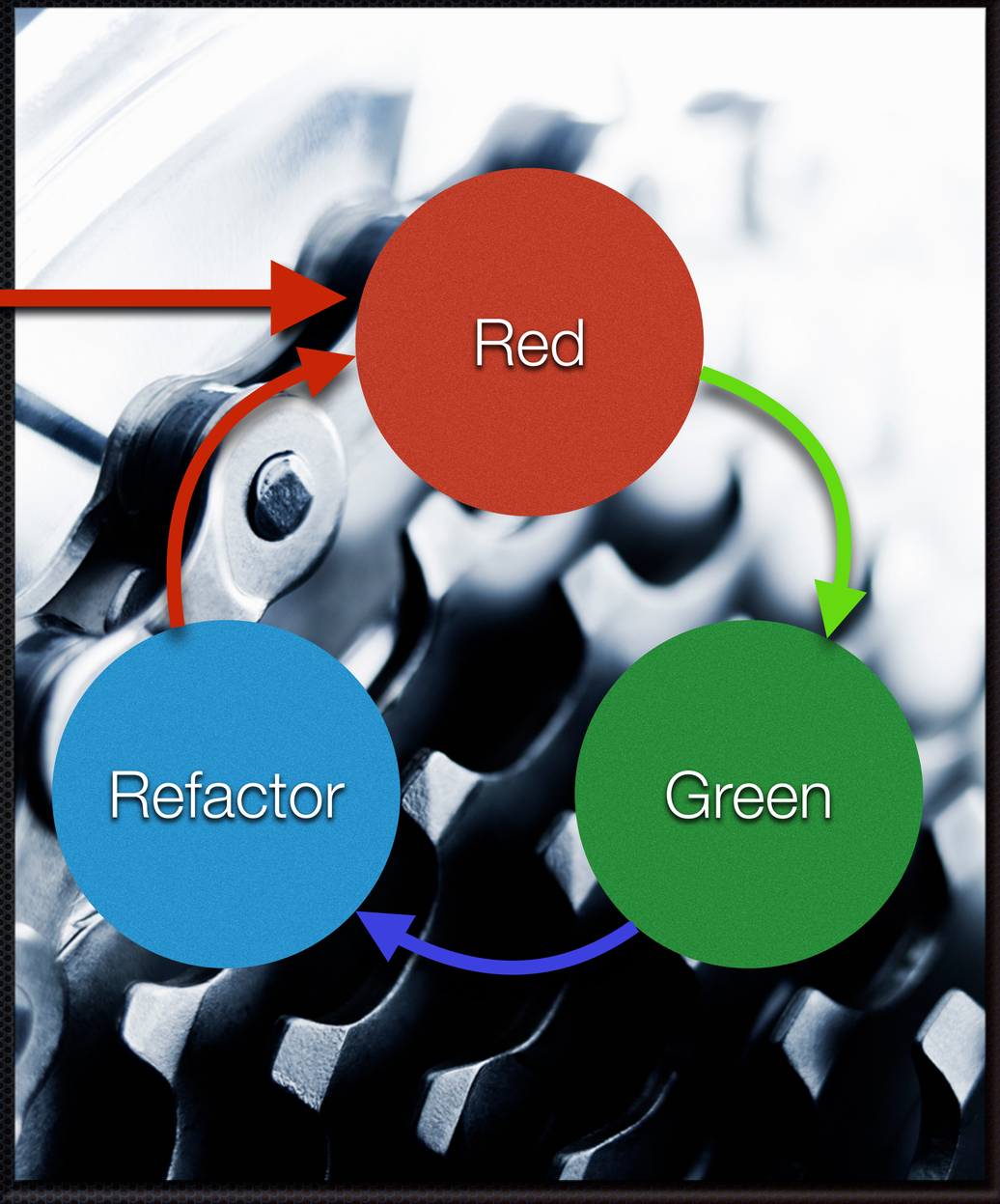
Michael Meissner 15.02.2023

Agenda

- TDD - my short extract
- Testing Strategy
  - Following one Dimension
  - Example Roman Calendar

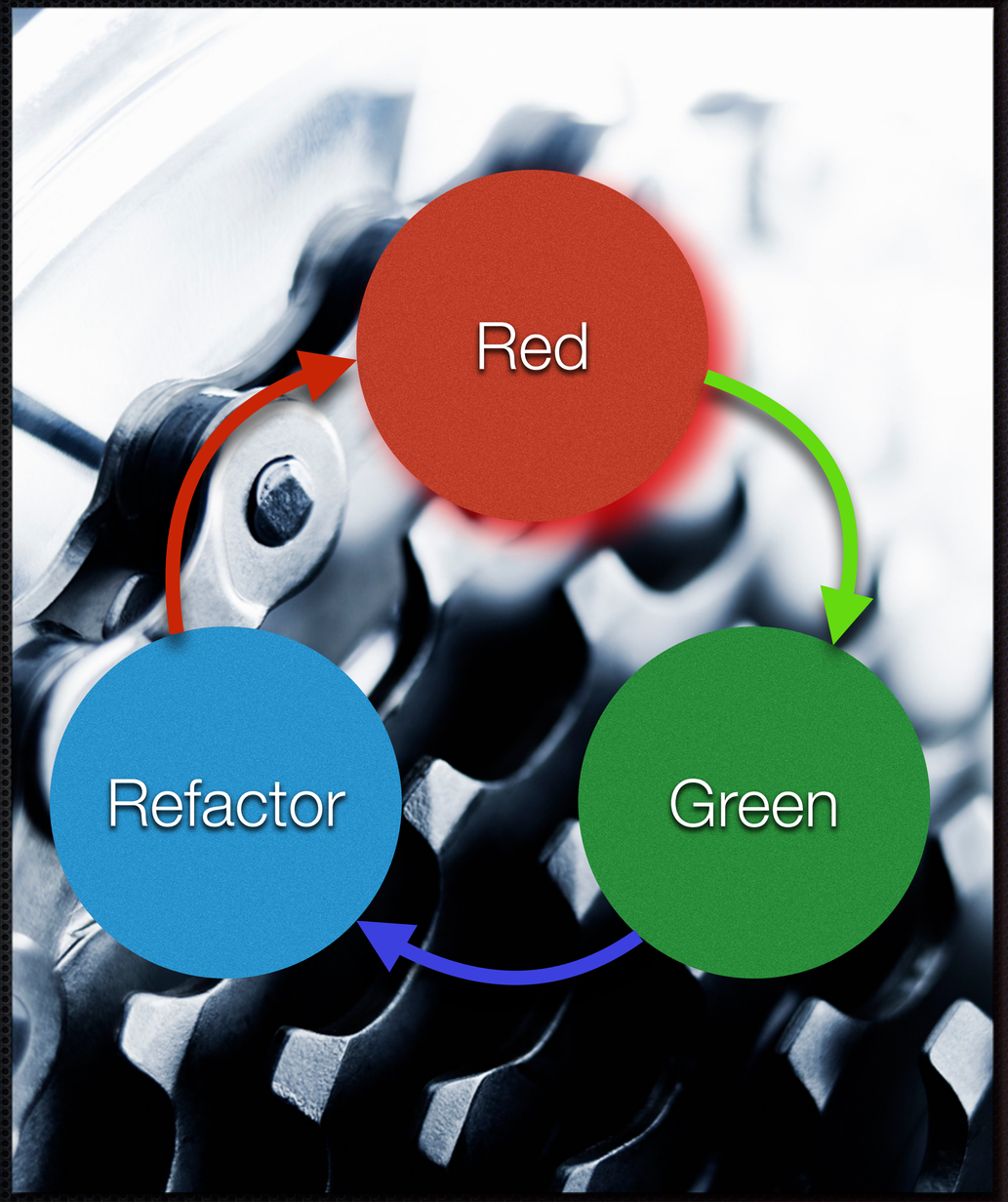
TDD : Circle

- Red - One Failing Test
- Make It Green
- Refactor



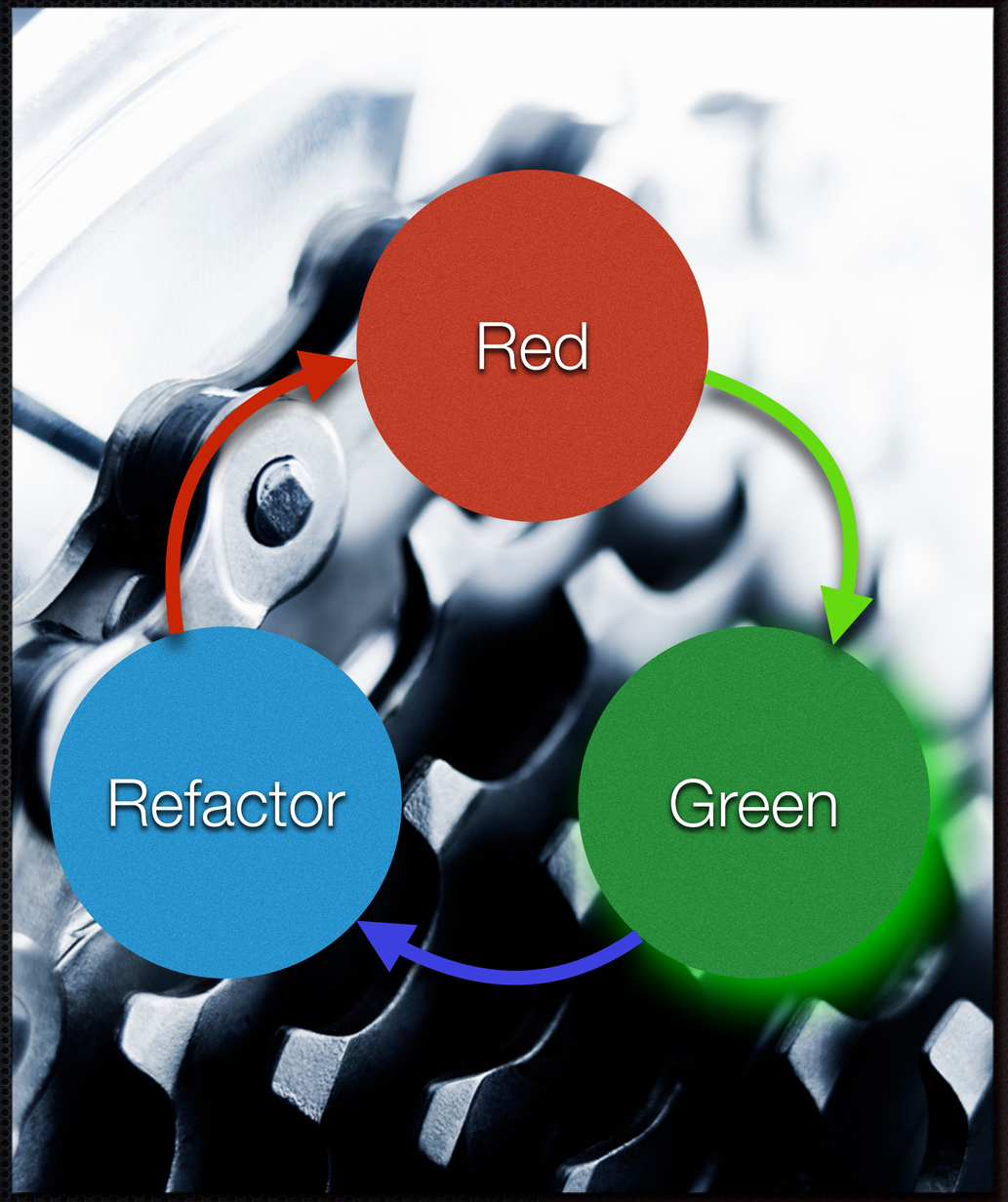
Red -
One Failing Test

- write one simple Test failing
  for the right Reason
- test the behavior and
  not the implementation
- priorisize the happy path
- start with the assertion
- structure the test: arrange, Act and Assert
  Naming Convention
  <Class>Should： Test: <verb><Expectation>



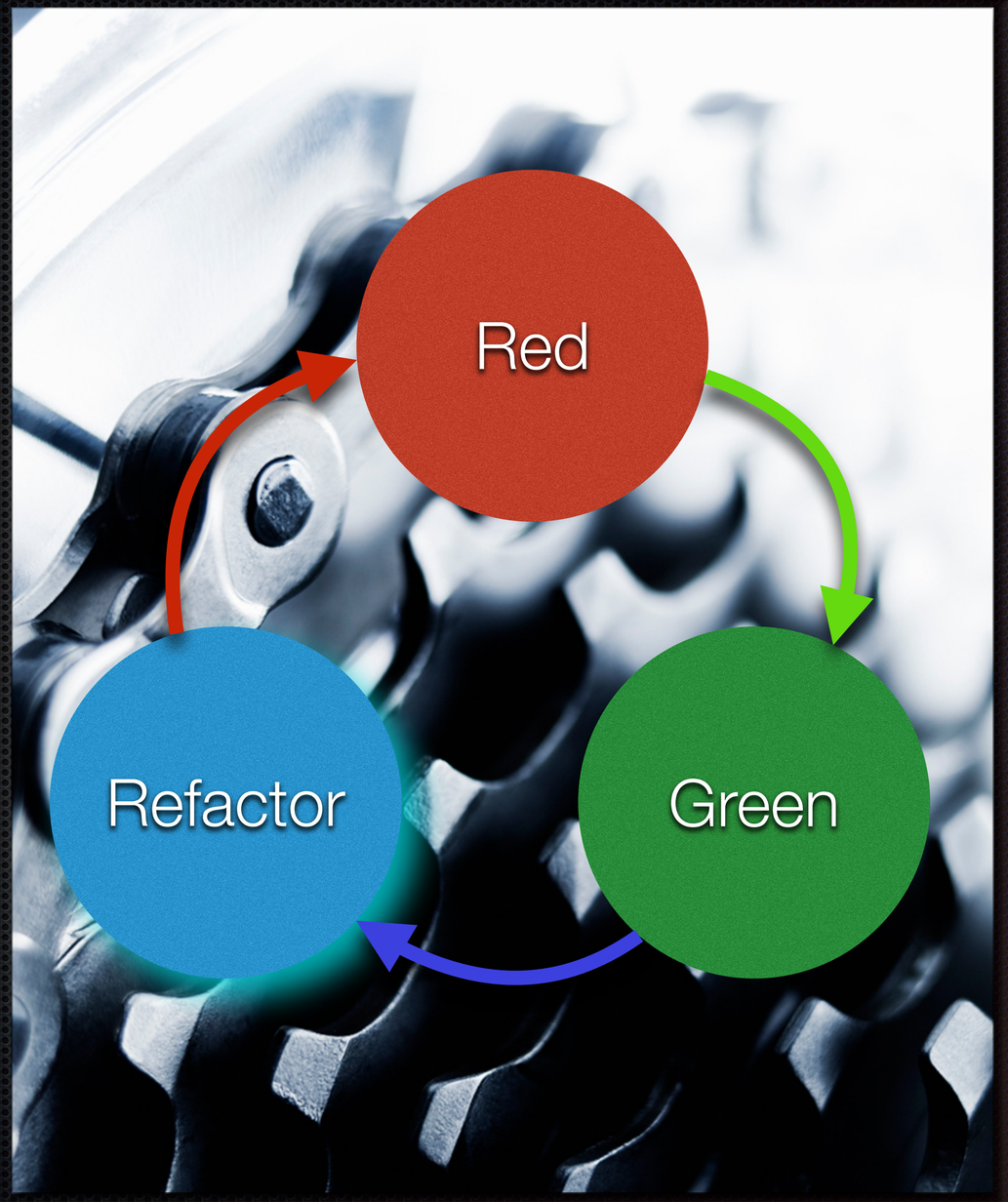
Make It Green

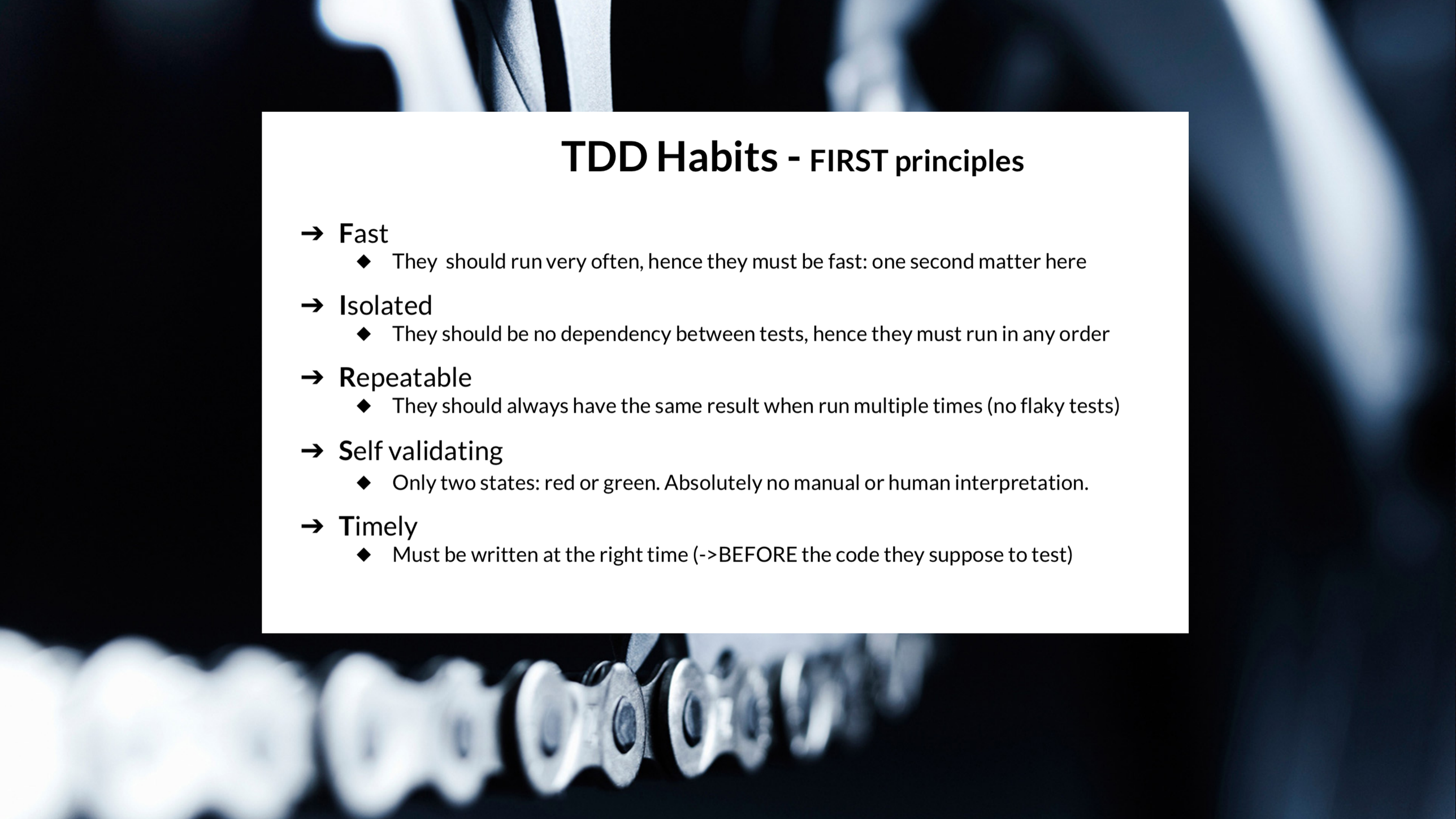
- Write just simplest code
  to pass all tests
  - Fake implementation
  - Triangulation
  - Obvious Implementation
    - Transformation Priority Premise
- Commit with name of test



Refactor

- Refactor Test and Production Code
  - better naming
  - remove Code duplications „Rule of Three“
  - extract methods
- run Green and commit





TDD Habits - FIRST principles

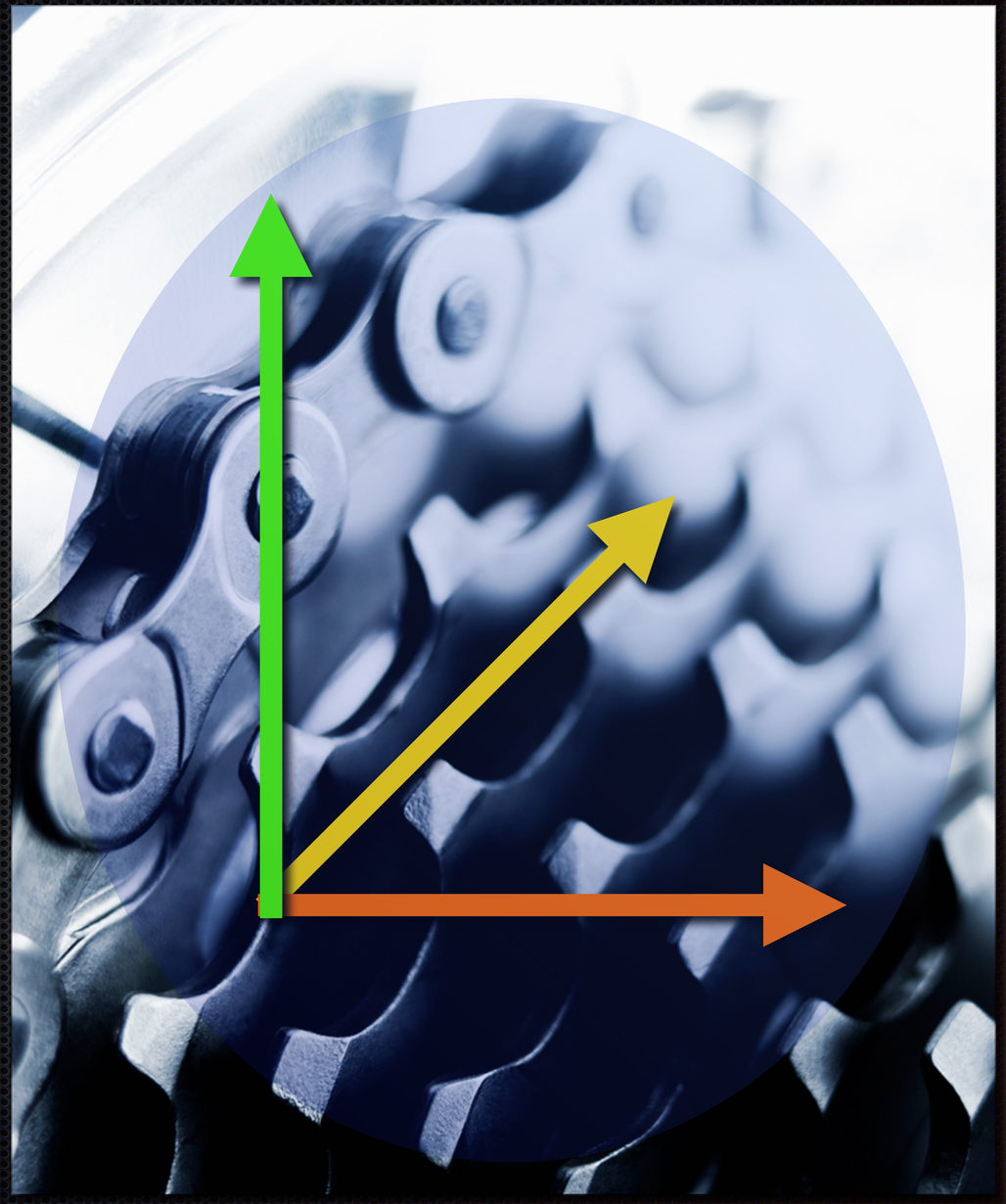
- **F**ast
  - They should run very often, hence they must be fast: one second matter here
- **I**solated
  - They should be no dependency between tests, hence they must run in any order
- **R**epeatable
  - They should always have the same result when run multiple times (no flaky tests)
- **S**elf validating
  - Only two states: red or green. Absolutely no manual or human interpretation.
- **T**imely
  - Must be written at the right time (->BEFORE the code they suppose to test)

Testing Strategie

It’s impossible to test the complete system.
So we test until we have sufficient confidence.

Follow the Dimension

Test one dimension of freedom,
so that you have enough confidence
for the production code ,
before switching
to the next dimension



Example Roman Calendar

Dimensions:

1. Additive Symbols I,X,C,M
2. Special Cases

1. Repetition of I, X, C

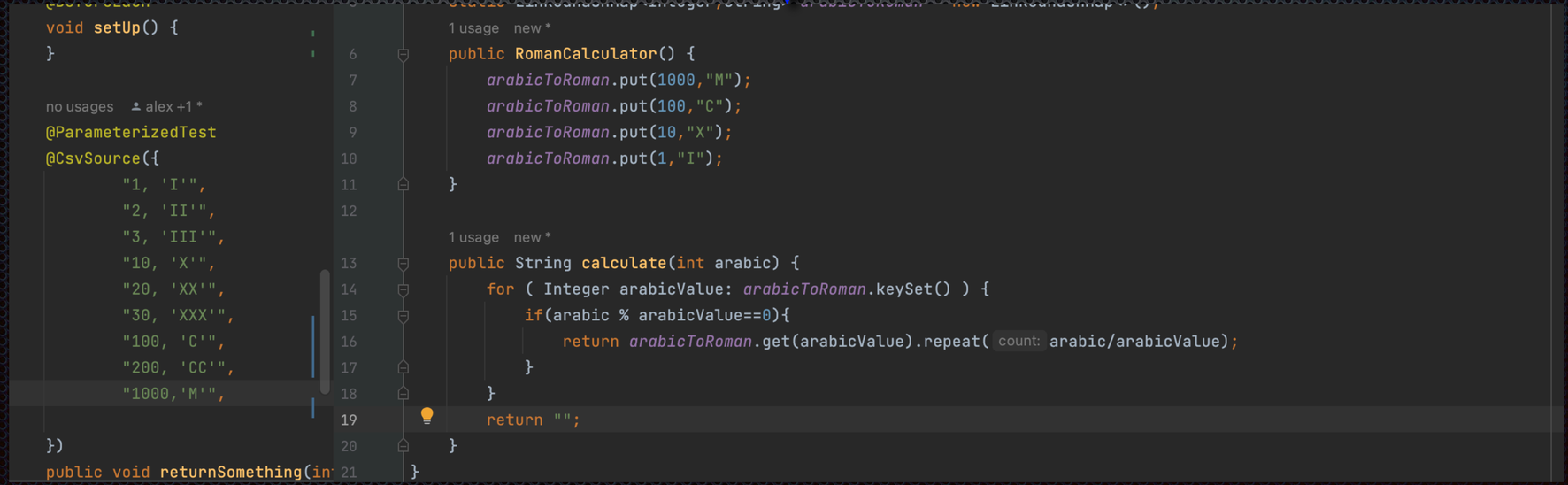
```
void setUp() {
}

no usages   alex +1 *
@ParameterizedTest
@CsvSource({
        "1, 'I'",
        "2, 'II'",
        "3, 'III'",
        "10, 'X'",
        "20, 'XX'",
        "30, 'XXX'",
        "100, 'C'",
        "200, 'CC'",
        "1000,'M'",
```

```
public class RomanCalculator {
    1 usage   new *
    public String calculate(int arabic) {
        if(arabic % 1000==0){
            return "M".repeat( count: arabic/1000);
        }
        if(arabic % 100==0){
            return "C".repeat( count: arabic/100);
        }
        if(arabic % 10==0){
            return "X".repeat( count: arabic/10);
        }
        return "I".repeat(arabic);
    }
}
```

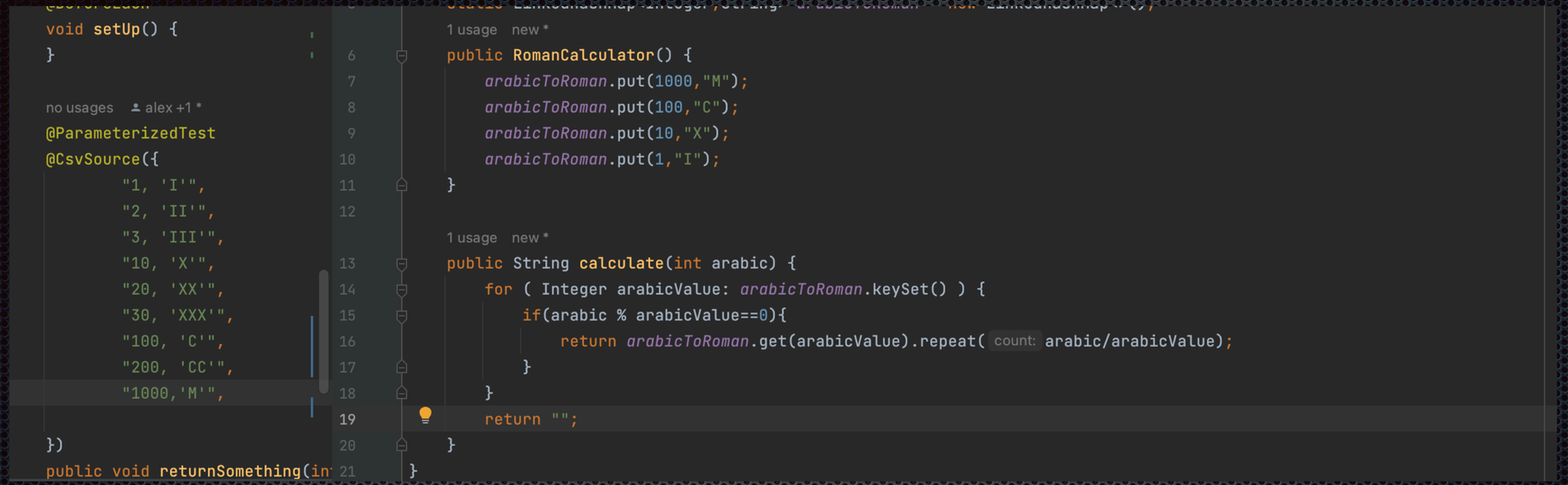
1. Repetition of I, X, C

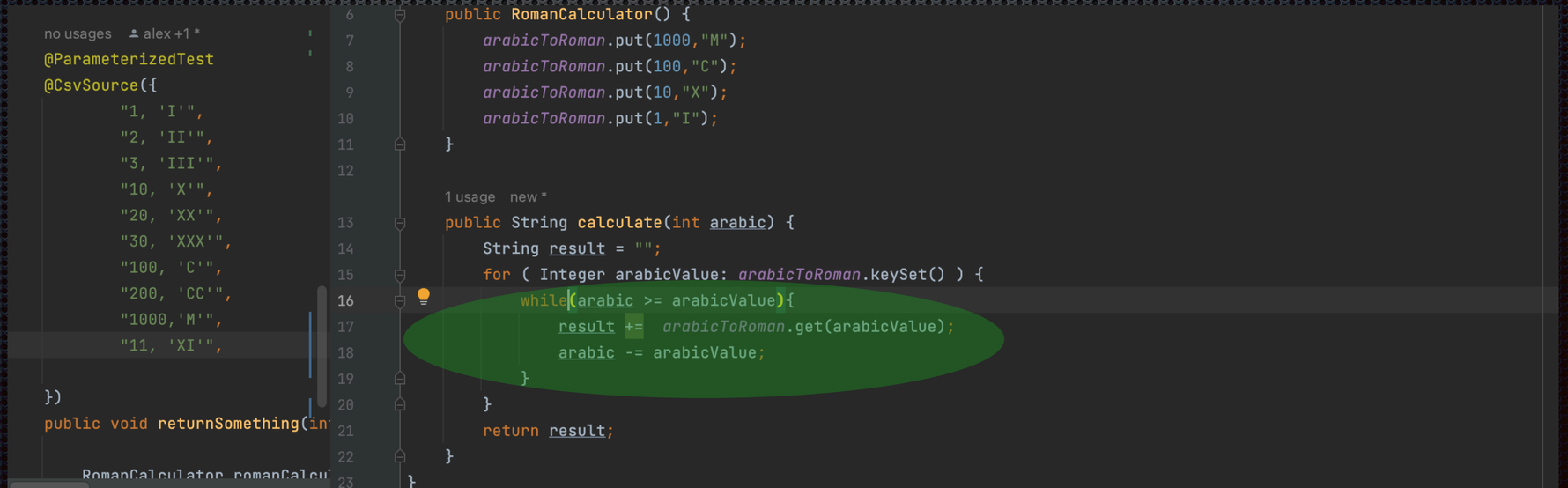
```
void setUp() {
}

no usages  alex +1 *
@ParameterizedTest
@CsvSource({
        "1, 'I'",
        "2, 'II'",
        "3, 'III'",
        "10, 'X'",
        "20, 'XX'",
        "30, 'XXX'",
        "100, 'C'",
        "200, 'CC'",
        "1000,'M'",
```

```
public class RomanCalculator {
    1 usage  new *
    public String calculate(int arabic) {
        if(arabic % 1000==0){
            return "M".repeat( count: arabic/1000);
        }
        if(arabic % 100==0){
            return "C".repeat( count: arabic/100);
        }
        if(arabic % 10==0){
            return "X".repeat( count: arabic/10);
        }
        return "I".repeat(arabic);
    }
}
```



2. additive numbers of different characters I, X, C





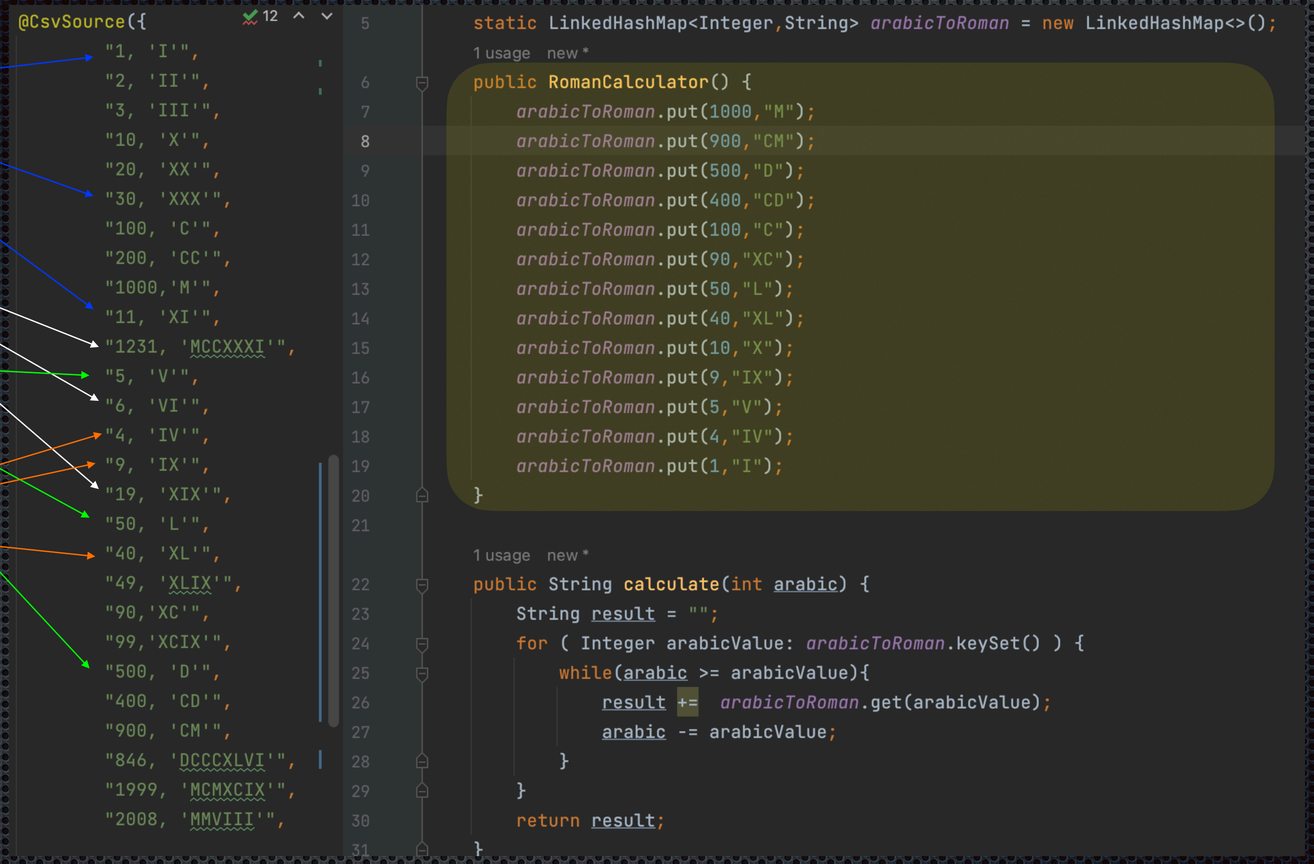
3. special cases

Symbol Repetition and Addition

for Confidence

new single Symbols

new Symbols special case



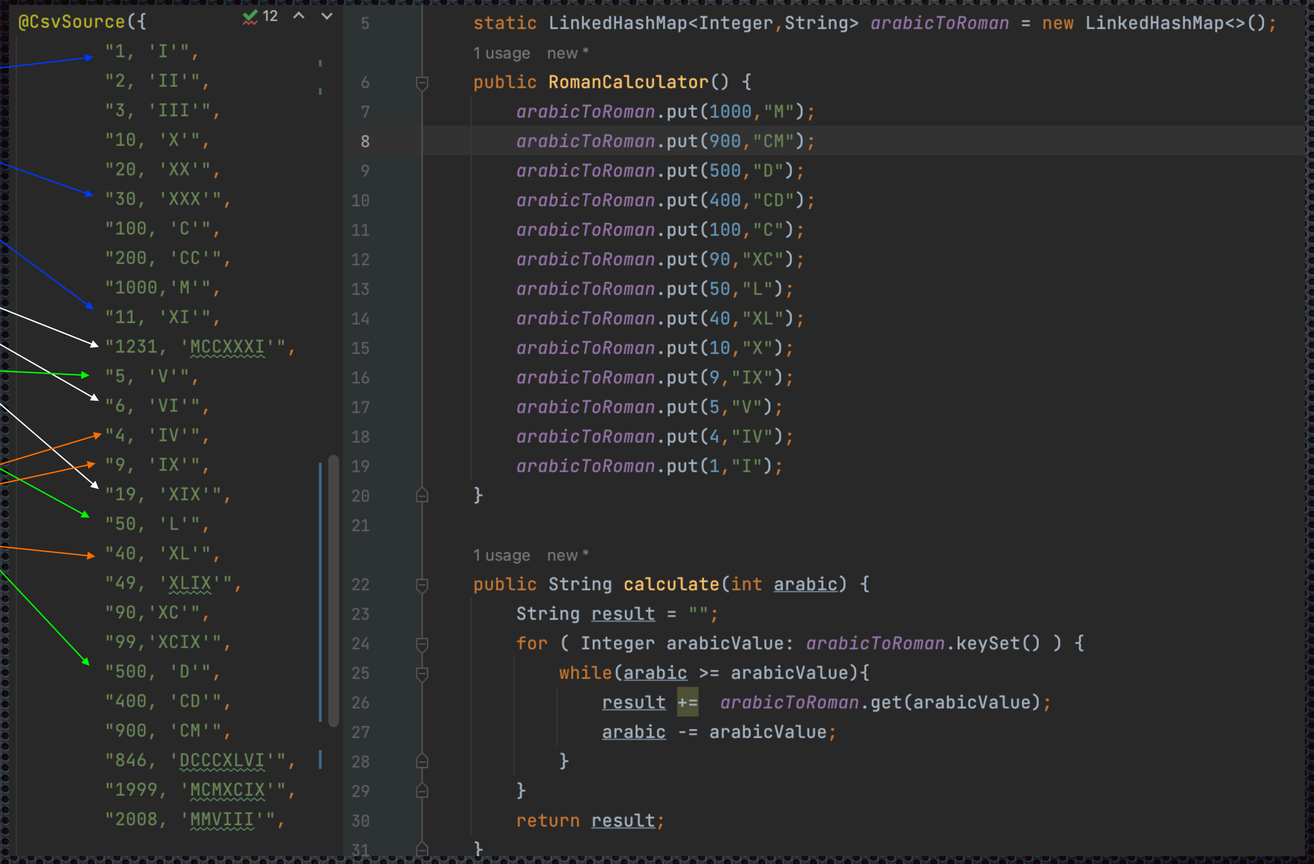
3. special cases

Symbol Repetition and Addition

for Confidence

new single Symbols

new Symbols special case

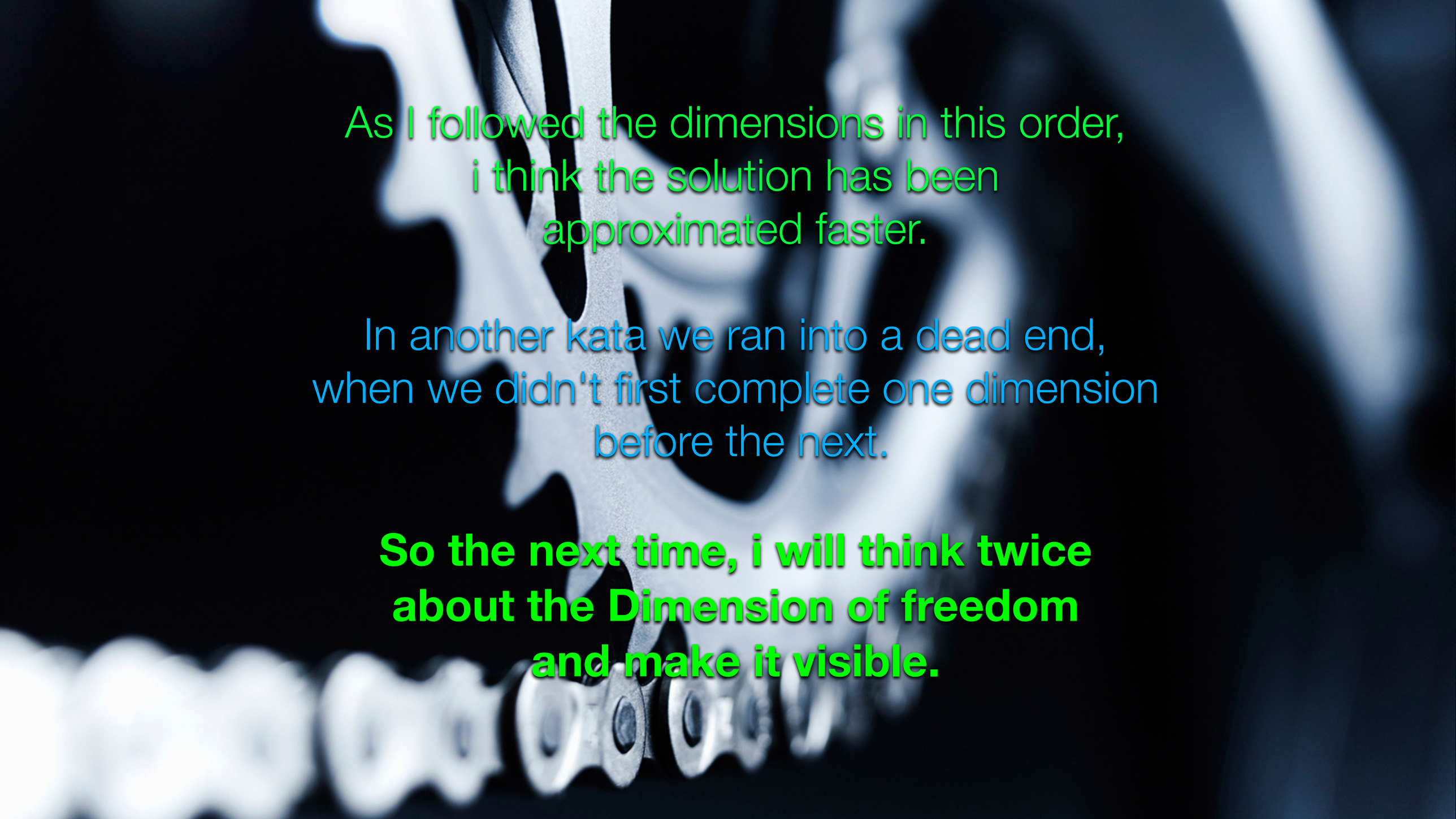


```
@CsvSource({
        "1, 'I'",
        "2, 'II'",
        "3, 'III'",
        "10, 'X'",
        "20, 'XX'",
        "30, 'XXX'",
        "100, 'C'",
        "200, 'CC'",
        "1000,'M'",
        "11, 'XI'",
        "1231, 'MCCXXXI'",
        "5, 'V'",
        "6, 'VI'",
        "4, 'IV'",
        "9, 'IX'",
        "19, 'XIX'",
        "50, 'L'",
        "40, 'XL'",
        "49, 'XLIX'",
        "90,'XC'",
        "99,'XCIX'",
        "500, 'D'",
        "400, 'CD'",
        "900, 'CM'",
        "846, 'DCCCXLVI'",
        "1999, 'MCMXCIX'",
        "2008, 'MMVIII'",
```

```
    static LinkedHashMap<Integer,String> arabicToRoman = new LinkedHashMap<>();
    1 usage  new *
    public RomanCalculator() {
        arabicToRoman.put(1000,"M");
        arabicToRoman.put(900,"CM");
        arabicToRoman.put(500,"D");
        arabicToRoman.put(400,"CD");
        arabicToRoman.put(100,"C");
        arabicToRoman.put(90,"XC");
        arabicToRoman.put(50,"L");
        arabicToRoman.put(40,"XL");
        arabicToRoman.put(10,"X");
        arabicToRoman.put(9,"IX");
        arabicToRoman.put(5,"V");
        arabicToRoman.put(4,"IV");
        arabicToRoman.put(1,"I");
    }

    1 usage  new *
    public String calculate(int arabic) {
        String result = "";
        for ( Integer arabicValue: arabicToRoman.keySet() ) {
            while(arabic >= arabicValue){
                result +=  arabicToRoman.get(arabicValue);
                arabic -= arabicValue;
            }
        }
        return result;
    }
```

I’ve also mixed the Dimension of special cases:

- Single Symbols, which are not used multiplicative (V,L,…)
- Values which are subtraktive used (IV,IX,XC,…)



As I followed the dimensions in this order,
i think the solution has been
approximated faster.

In another kata we ran into a dead end,
when we didn't first complete one dimension
before the next.

**So the next time, i will think twice
about the Dimension of freedom
and make it visible.**

Questions ?

Appendix

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**		