Test Driven Development (TDD)

Rudi Stene

From “test first” to “test driven”

- Difficult to create a complete test
- You must start thinking about implementation details first
- Big and complex tests.
- Testing multiple tings in one test
- Often you end up refactoring or fixing the test when implementing.
- Testing classes and methods over focus on behavior.
- Tests tightly coupled to implementation.

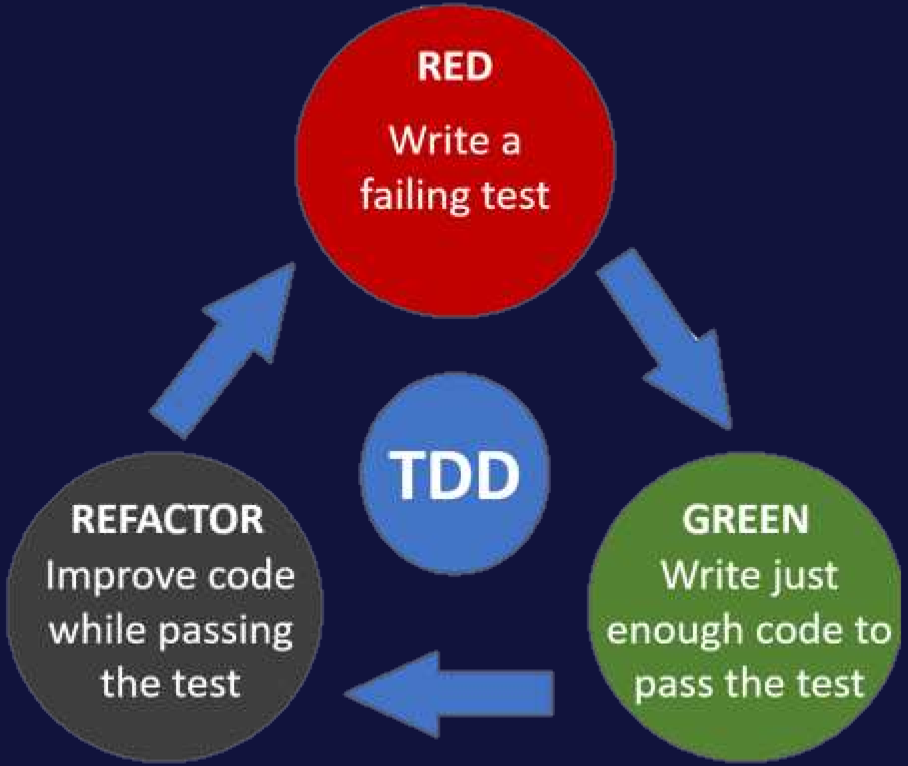
Classic TTD

The three laws of TDD:

1. You are not allowed to write any production code unless it is for making failing unit test pass

2. You are not allowed to write any more of a unit test than is sufficient to fail

3. You are not allowed to write any more production code that is sufficient to pass the one failing unit test.


RED
Write a failing test
TDD
GREEN
Write just enough code to pass the test
REFACTOR
Improve code while passing the test

Source: Alcor Academy, Walking, lesson 1, page 14

Classic TTD

Baby steps:

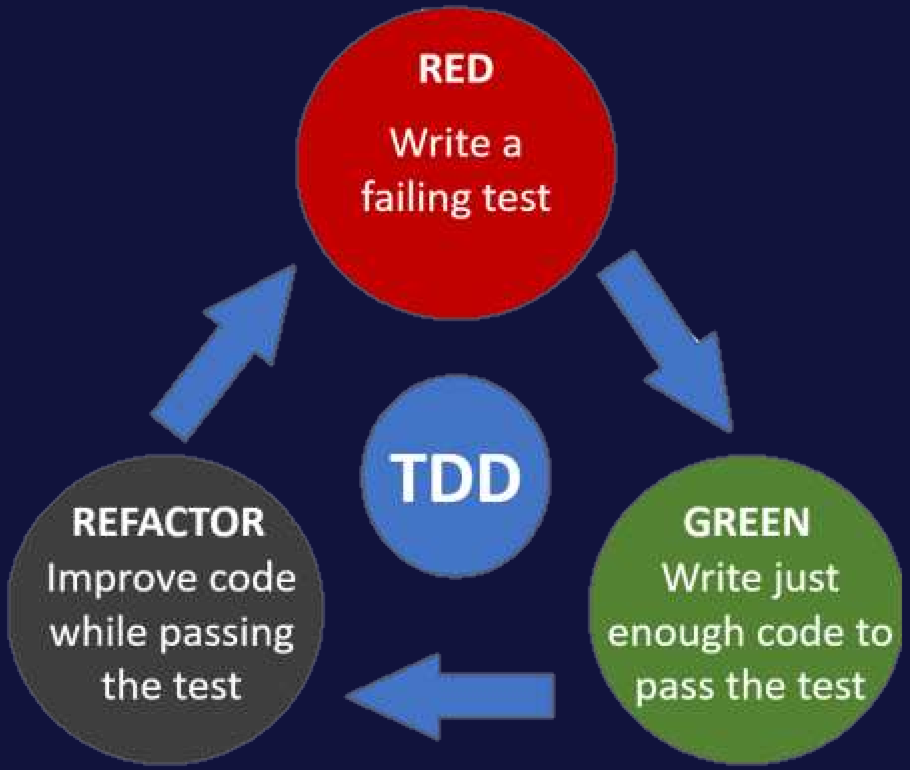
- Fake implementation

- Obvious implementation (use Transformation Priority Premise)

- Triangulation with the next test

We refactor only when tests passes.

- Extract duplication only when you see it for the third time
  (Rule of Three)


RED
Write a failing test
TDD
GREEN
Write just enough code to pass the test
REFACTOR
Improve code while passing the test

Source: Alcor Academy, Walking, lesson 1, page 14

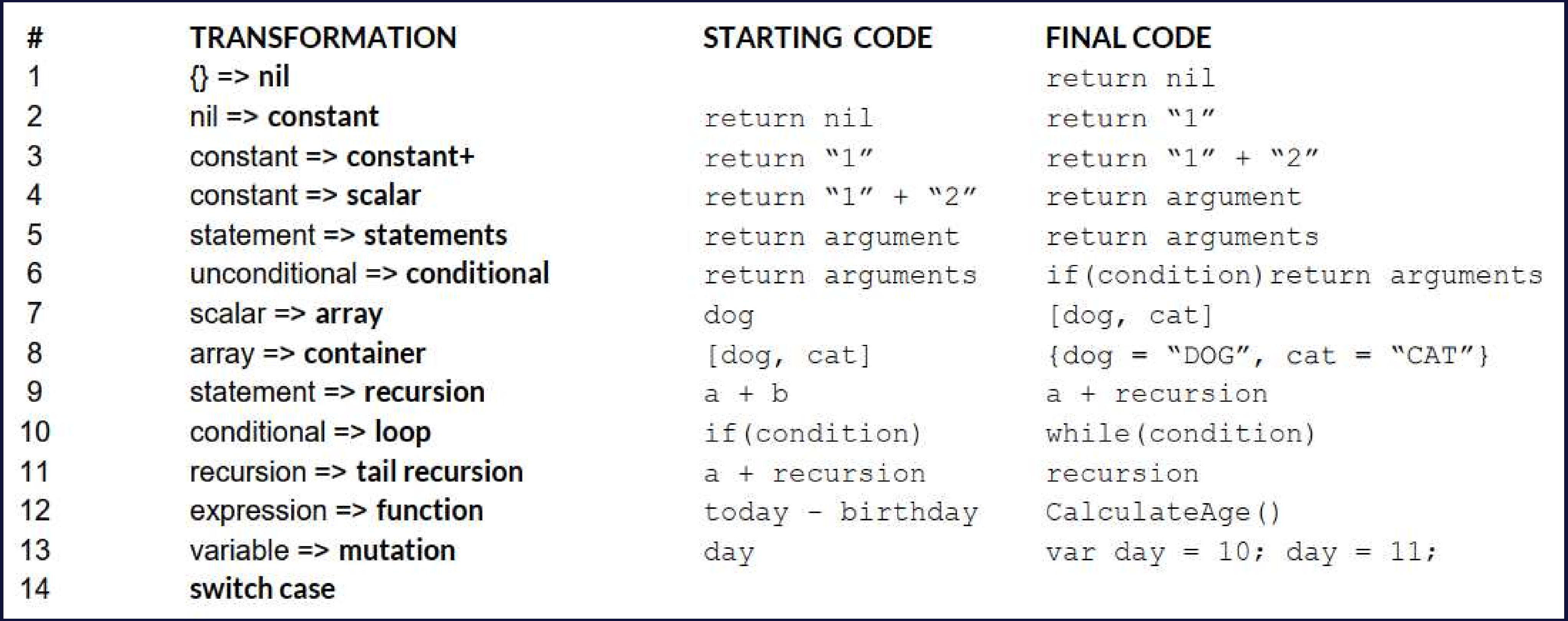
Classic TDD

- Test behavior, not implementation details.
- Name tests so it gives a description of the behavior.
- Tests can be used as documentation when named properly.
- Explore one degree of freedom at a time

- Organize unit test into three blocks:
    1. Arrange
    2. Act
    3. Assert

TPP – Transformation Priority Premise

- Prefer transformation from the top of the following list.
- Transformations ordered by complexity.



#	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**		

Source: Alcor Academy, Walking, lesson 3, page 4 bouvet

Thanks for your attention

Any questions?

Contact information:

✉ rudi.stene@bouvet.no

in rudistene