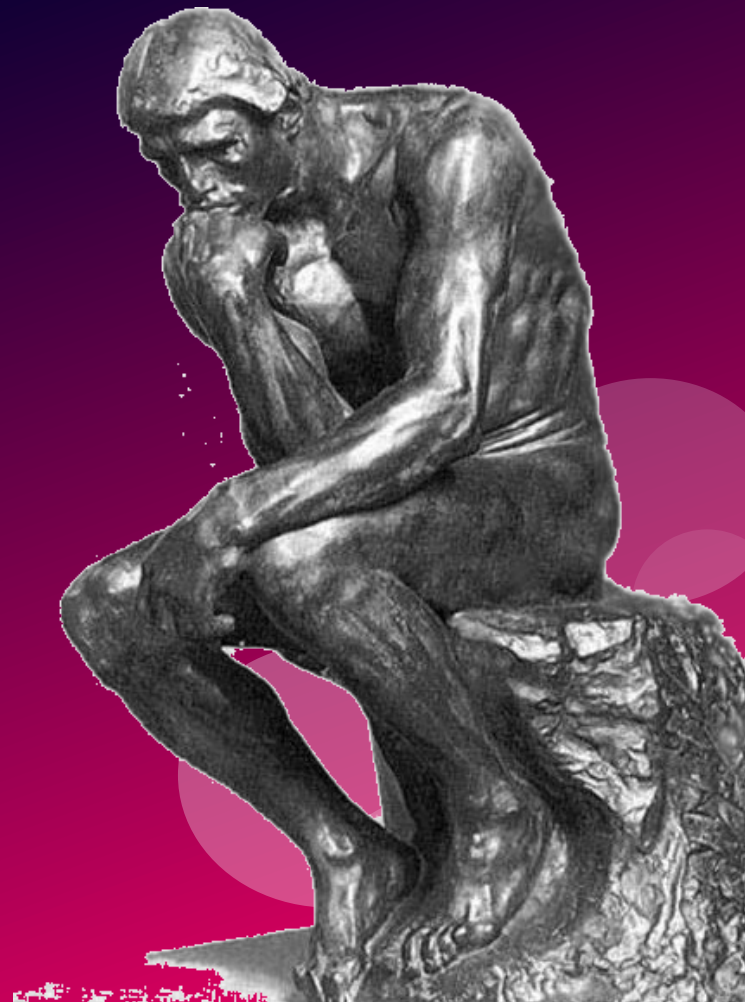
TDD impressions

TDD. Yes, but what do we test
BEHAVIOUR not implementation



Transformation Priority Premise

-
- Fake implementation →
  Hardcode exactly the value.
-
- Triangulation with next test →
  Use it on the way to a more generic solution until the implementation gets obvious. Go from one dimension to another.
-
- Obvious implementation
-

Code evolution

#	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 (map)	[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	int day = 10; day = 11;
14	switch case		

Parameterized tests

```
@ParameterizedTest
@CsvSource({
        "'1,\n2'",
        "'1\n,2'",
        "'1,,2'"
})
public void throw_exception_when_input_is_not_valid(String invalidInput) {
    Assertions.assertThrowsExactly(IllegalArgumentException.class, ()
            -> calculator.add(invalidInput));
}
```

Object Calisthenics rules

-
-
- Wrap all primitives and strings
- So we have an explicit type with a name. The value control and access can be easily managed.
-
- Wrap all collections
- Collection specific behaviour is on a single place. The internal representation is not effected by the rest.

Limits of TDD

- Algorithms
- I tried to calculate all permutation of a set of chars and failed. ChatGPT helped me with the algorithm. The written unit tests helped me to validate the code.

Efficiency (memory and cpu)

- The drawback of the obvious code can be reduces performance. Anyway, in most cases this does not matter and is not worth the optimisation. (e.g. A map is less efficient than an array)

- Forget a fake implementation in the productive code
- It can happen that a fake implementation is forgotten in the code as we always commit after a passing test.

- Start too fast with implementation. No big picture.
- More refactoring steps are needed because of intentionally wrong interface methods.

Conclusion (so far)

- + Higher code readability
- + Confidence about functionality
- + Methods to solve complex challenges
  (Baby steps, Triangulation)
- + Decreased code complexity
- + Encapsulated responsibility
- + Reduces noise
- - Slow process
- - Missing big picture

Thank you

Questions?