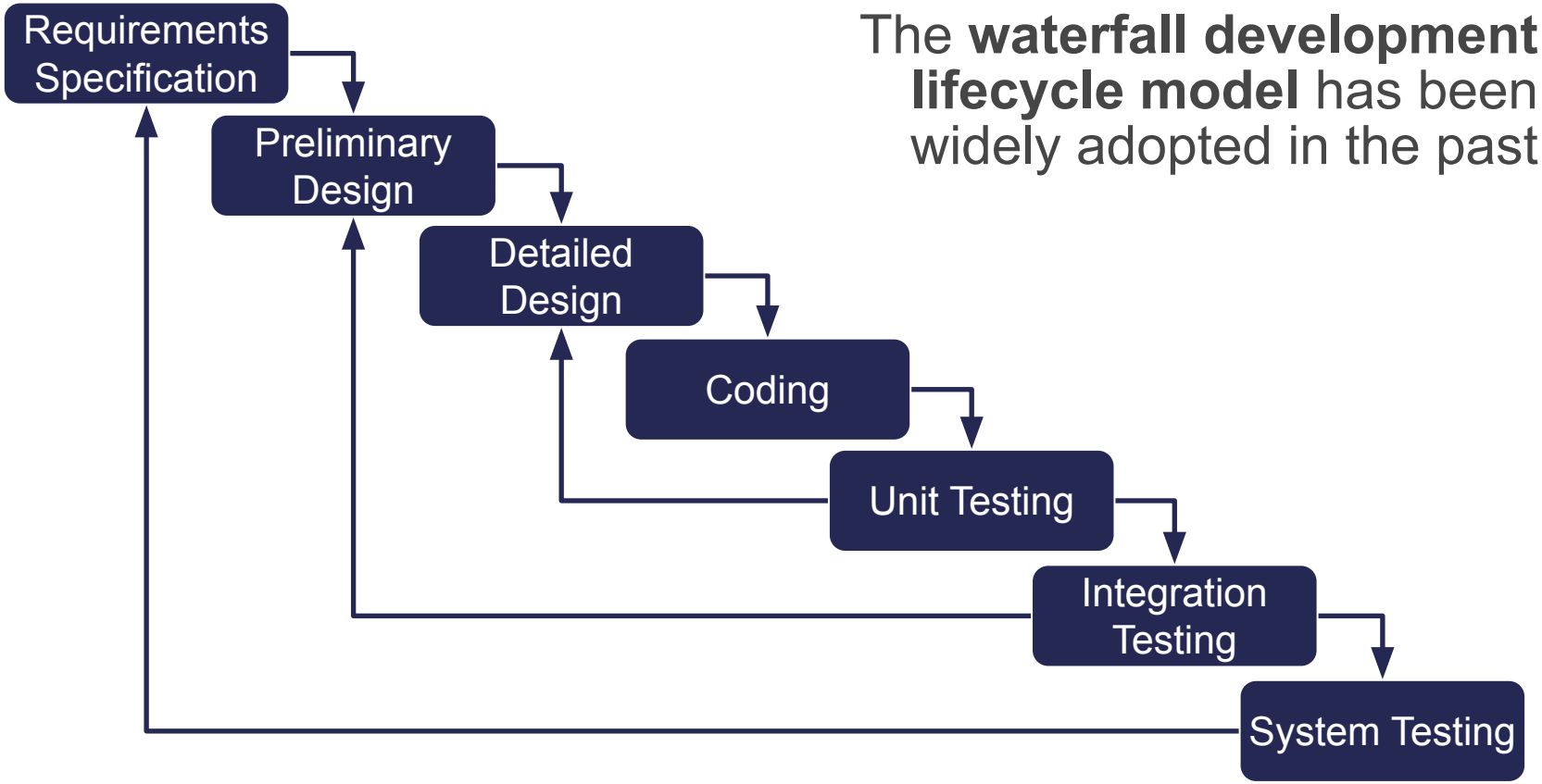
TDD

Giuseppe Scanniello

Simone Romano

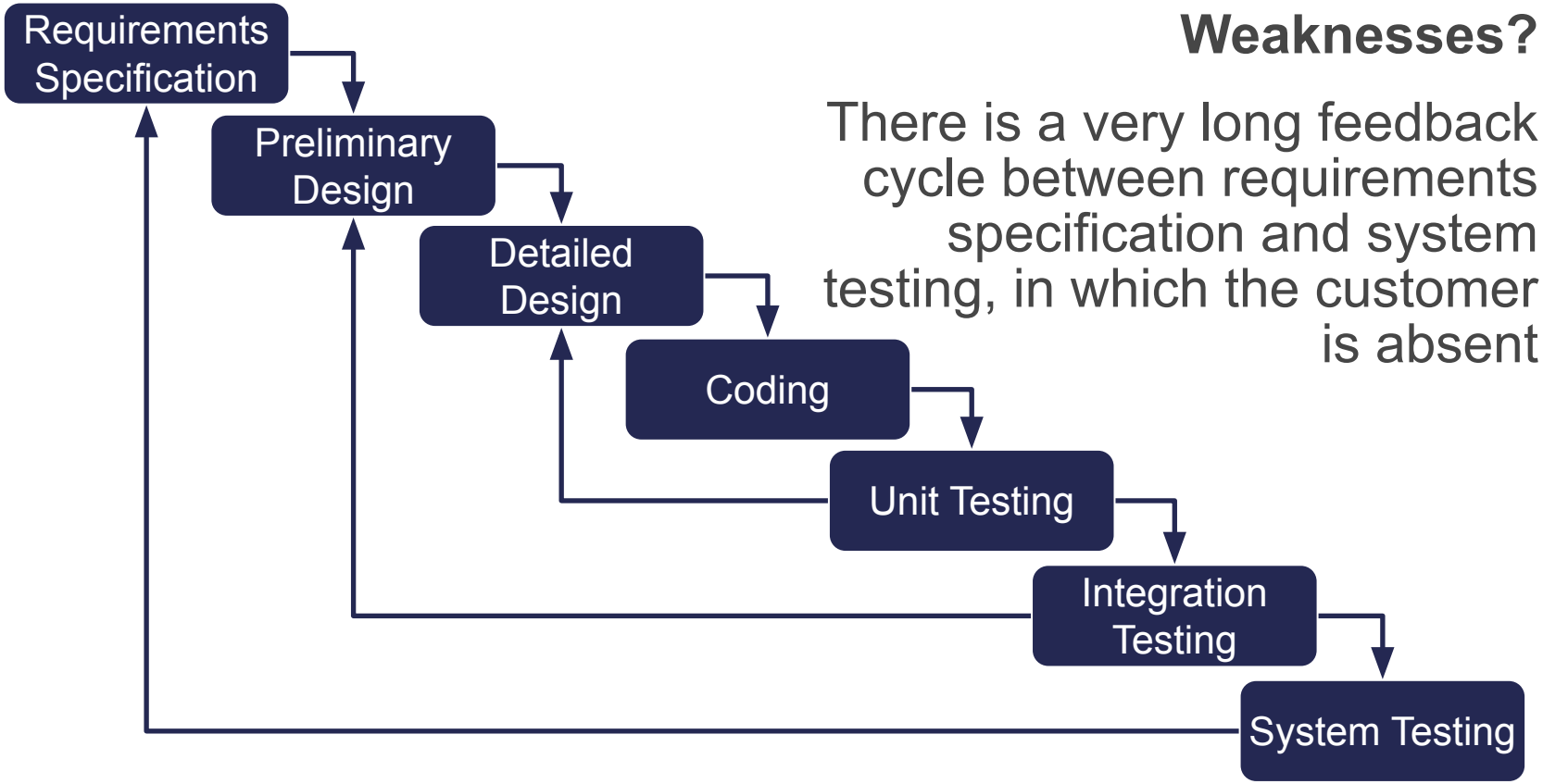
Michelangelo Esposito

Traditional development



The **waterfall development lifecycle model** has been widely adopted in the past

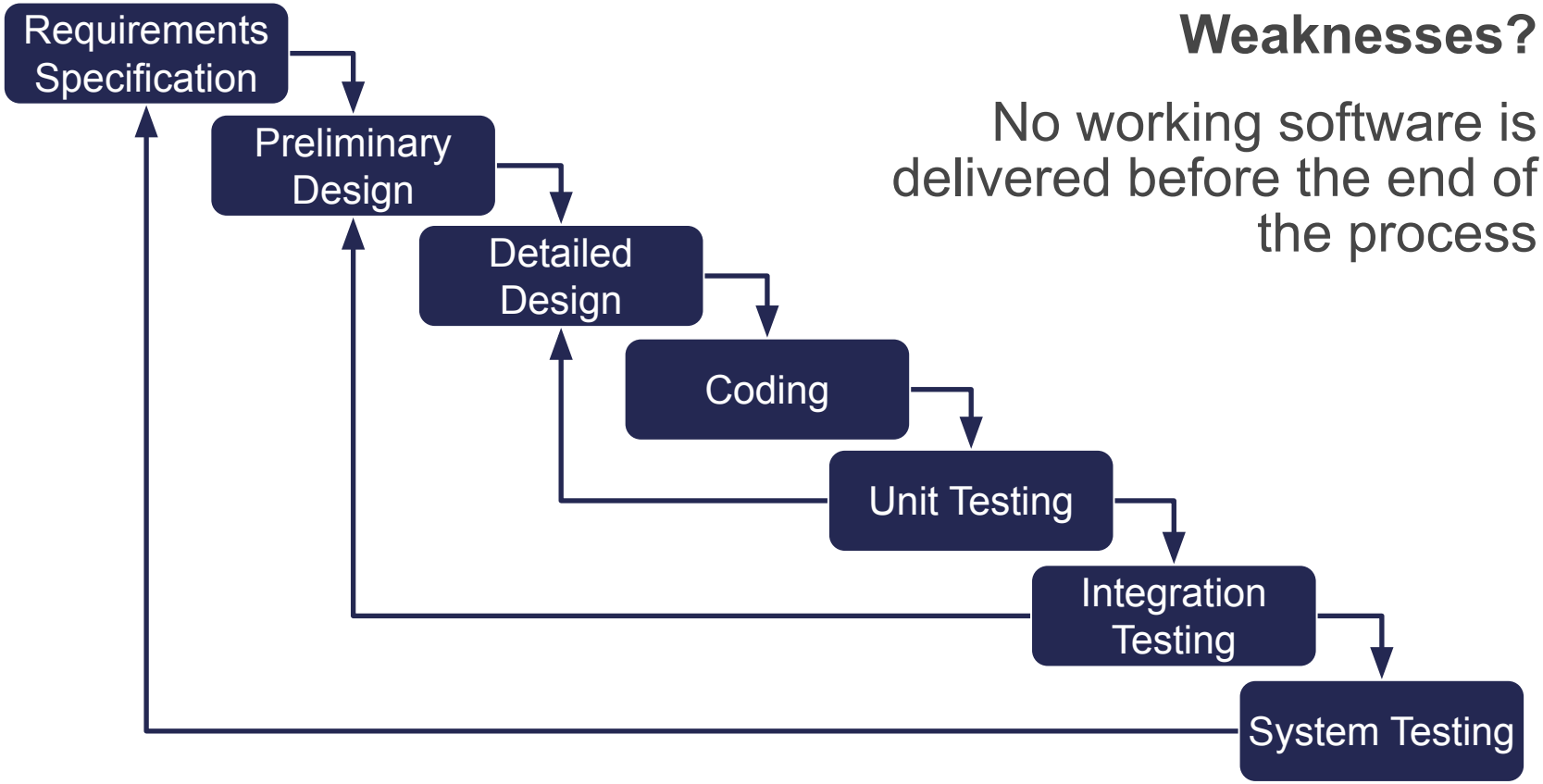
Traditional development



Weaknesses?

There is a very long feedback cycle between requirements specification and system testing, in which the customer is absent

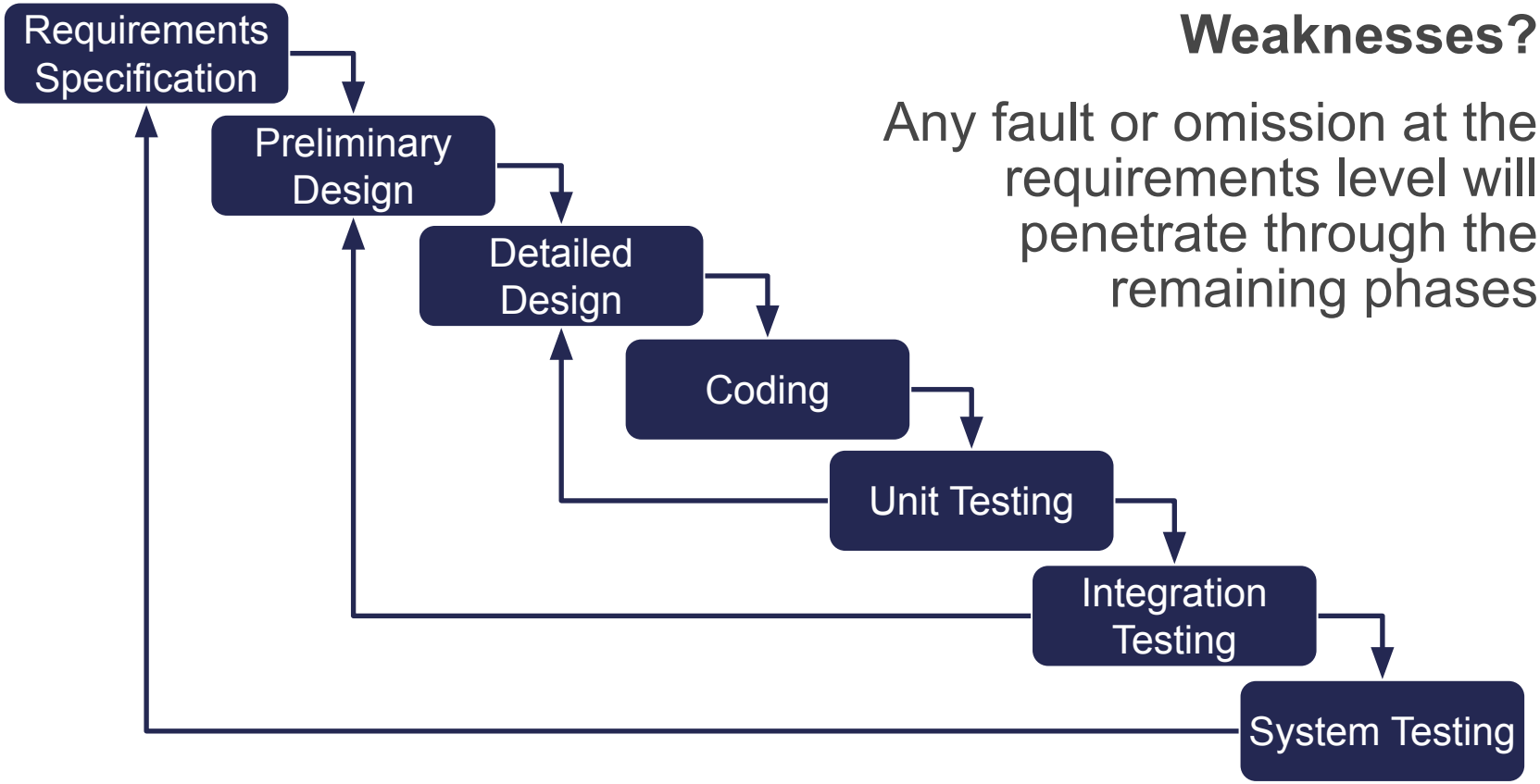
Traditional development



Weaknesses?

No working software is delivered before the end of the process

Traditional development



Weaknesses?

Any fault or omission at the requirements level will penetrate through the remaining phases

A way to overcome the limitations of traditional development?

Agile Development

Agile development

Based on the values and principles derived from the **Agile Manifesto**

There are several variants of agile software development (some websites list up to 40 variants)

- Extreme programming (XP)
- **Test-driven development** (**TDD**)
- Feature-driven development (FDD)
- Scrum
- …

Agile development

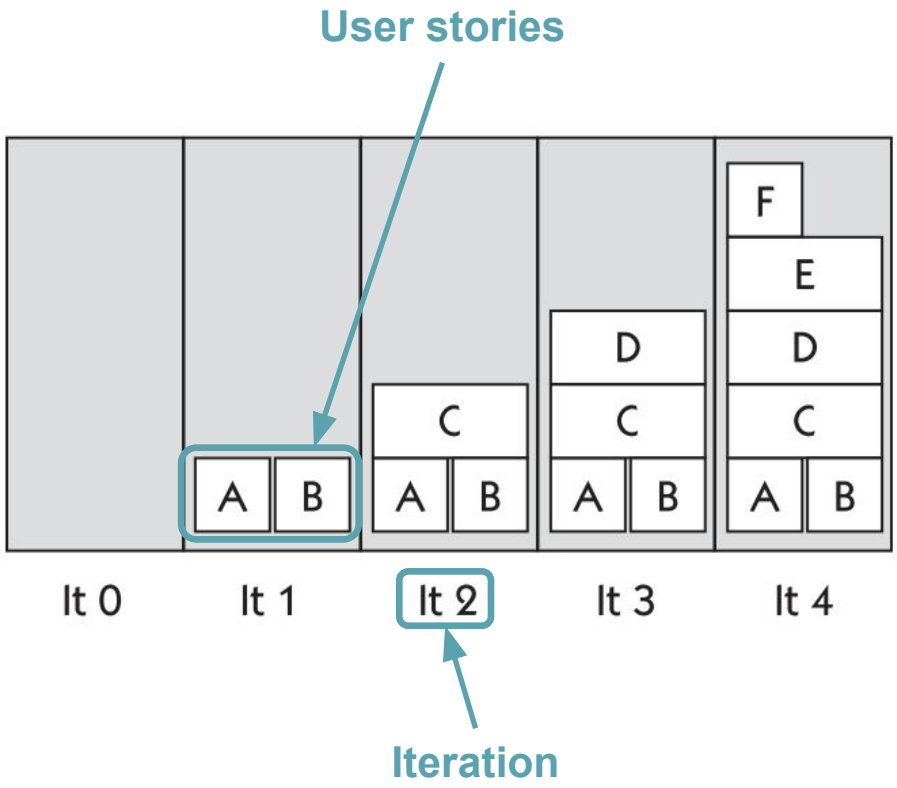
A **time boxed**, **iterative approach** to software development in which software systems are built and delivered **incrementally** from the beginning of the project, rather than trying to deliver it all at once near the end

It works by breaking down the software systems' functionality into little bits called **user stories**, prioritizing them, and continuously delivering them in short cycles called **iterations**

Testing in agile development

Each increment of coding is tested as soon as it is finished

A story is not "done" until it has been tested and the tests have passed

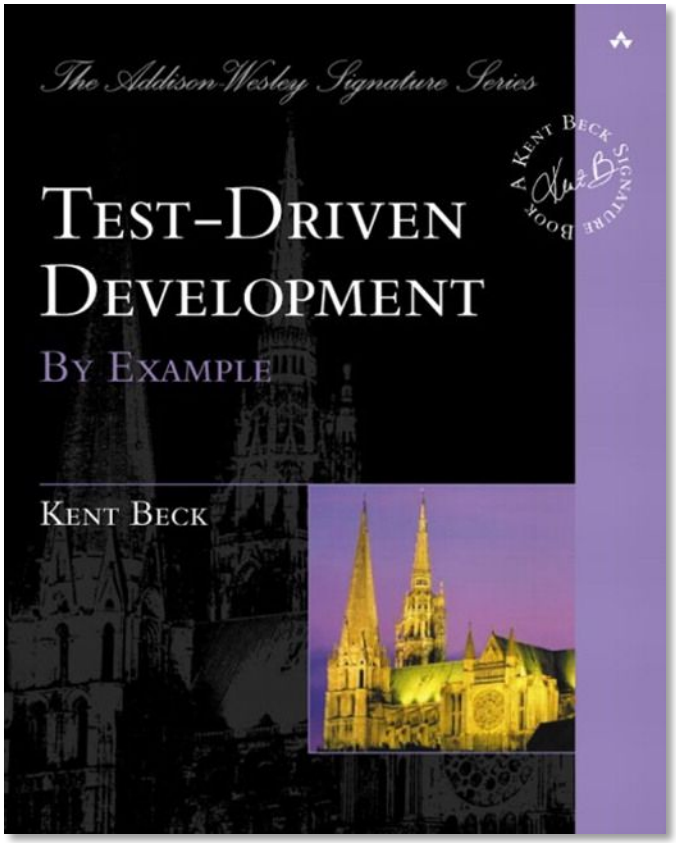


TDD

It has been proposed (or popularized) by Kent Beck in his 2002 book *Test-Driven Development by Example*

An approach to software development in which developers interleave testing, development, and refactoring

- Developers (not testers) first write **automated unit** tests and then the associated production code

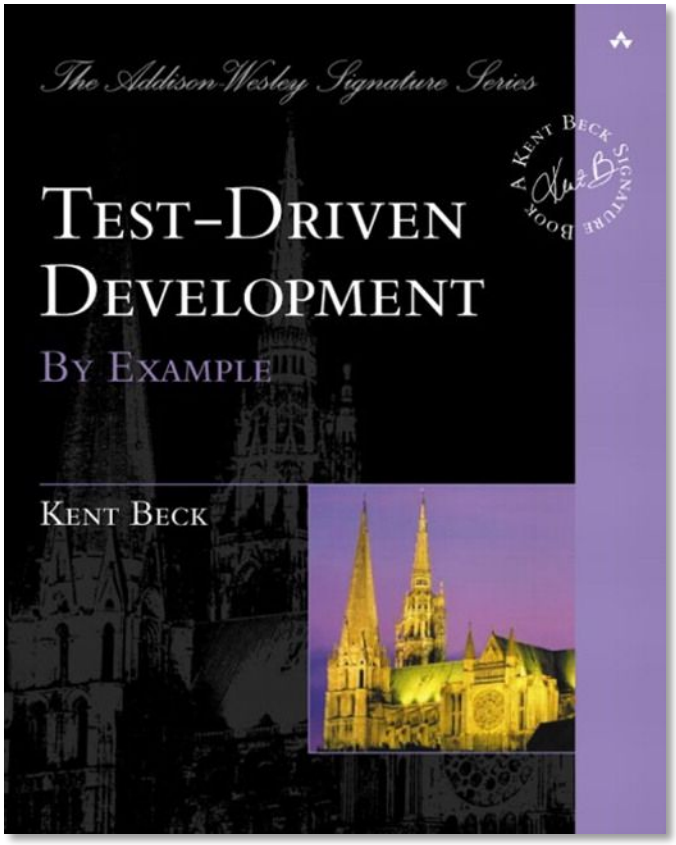


TDD

It is an extreme case of agility

- Developers focus on **smaller** increments of coding that they perform in **shorter** iterations (at most 10-15 minute long) w.r.t. to other software development variants

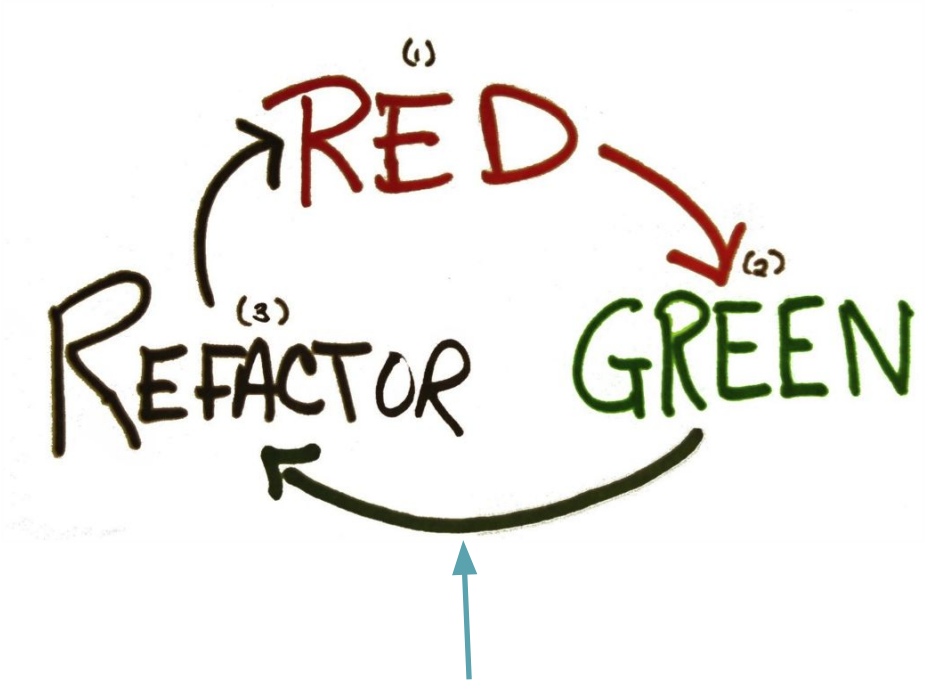
TDD is also used within other agile development variants like XP or Scrum



TDD mantra

Very short cycles of three phases:

1. **Red**
2. **Green**
3. **Refactor**



A cycle should last at most 10-15 minutes (less is even better)

Red phase

Write a failing unit test for a **small** chunk of functionality, which is not implemented yet

- Perhaps the test does not even compile at first

Run the test

Watch the test fail

Green phase

Make the unit test pass **quickly** (i.e., write the minimal amount of code to make the test pass), committing whatever sin necessarily in the process

Run the test (as well as any other test)

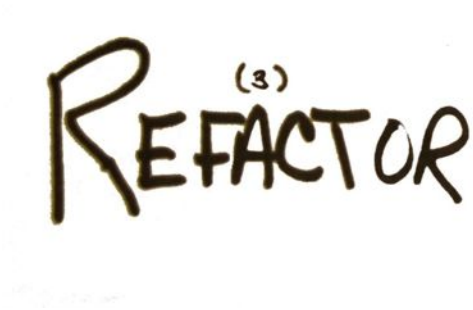
Watch the test pass (as well as any other test)

Refactor phase

Eliminate all duplications and smells created in just getting the test to pass

Run all tests

Watch them pass



Best practice

Make it green, then make it clean!

It is similar to a game, where your goal is to play with a unit test until it becomes green

The three laws of TDD

TDD practitioners follow three laws [Martin 2007]:

1. You may not write production code unless you have first written a failing unit test
2. You may not write more of a unit test than is sufficient to fail
3. You may not write more production code than is sufficient to make the failing unit test pass

Fundamental principles

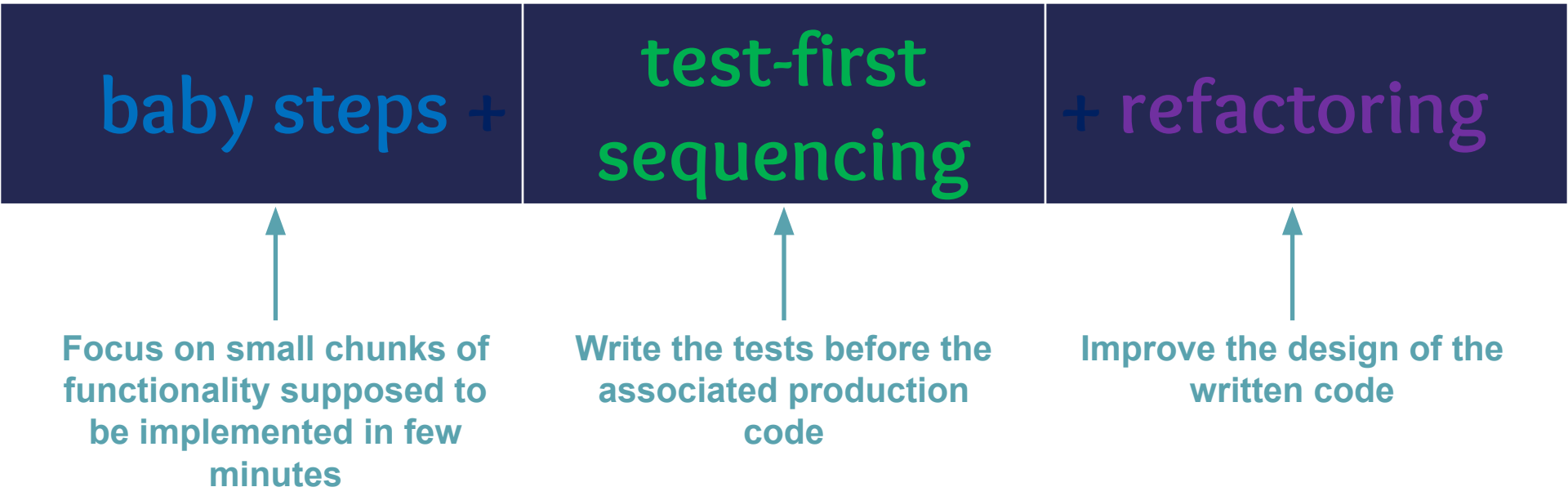
Think about what you are trying to do

Follow the TDD mantra, the best practice, and the three laws

Continually make small, incremental changes

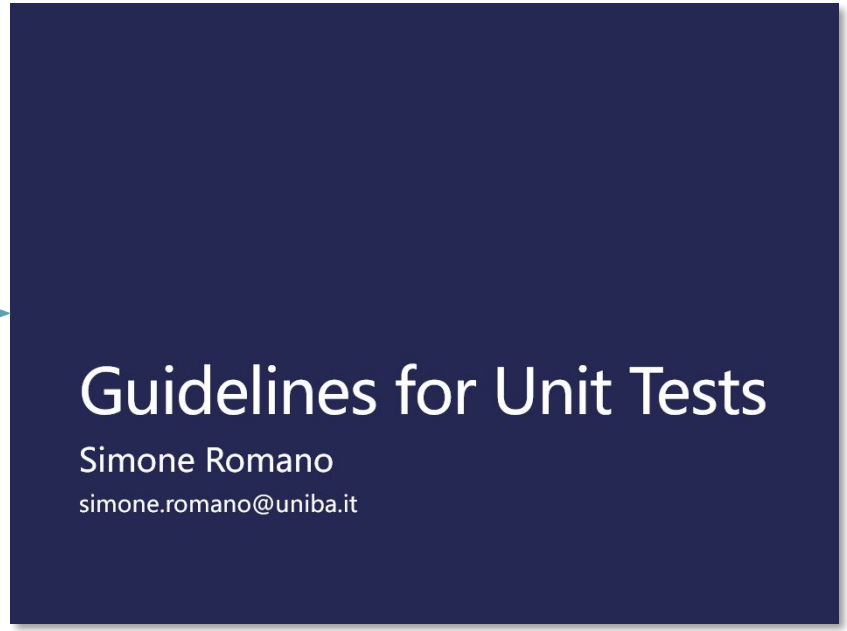
Keep the system running at all times---failures must be addressed immediately

TDD can be seen as…



Do I need guidelines for writing unit tests in the TDD contest?

Sure, you should already know them!!!



TDD in action

Let us apply TDD to the Fibonacci numbers example

$$f_0 = 0 \; if \; n = 0$$
$$f_1 = 1 \; if \; n = 1$$
$$f_n = f_{n-1} + f_{n-2} \; if \; n > 1$$

For simplicity, we will not consider invalid input values

TDD in action – cycle #1

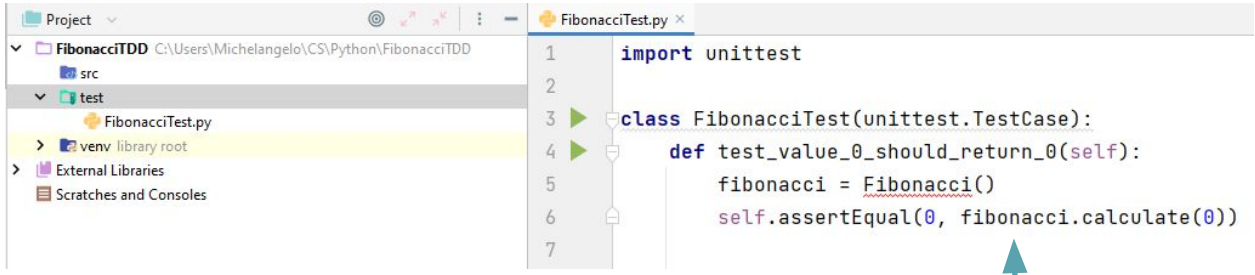
Red phase

Write a failing unit test for a **small** chunk of functionality, which is not implemented yet

- Perhaps the test does not even compile at first

Run the test

Watch the test fail



$$f_0 = 0 \; if \; n = 0$$
$$f_1 = 1 \; if \; n = 1$$
$$f_n = f_{n-1} + f_{n-2} \; if \; n > 1$$

Chunk of functionality #1

TDD in action – cycle #1

Red phase

Write a failing unit test for a **small** chunk of functionality, which is not implemented yet

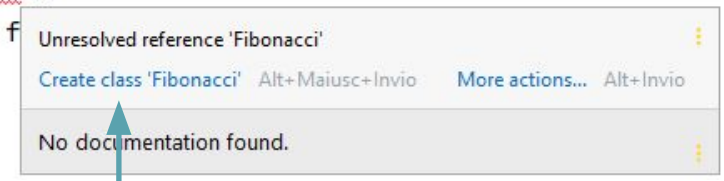
- Perhaps the test does not even compile at first

Run the test

Watch the test fail

```
import unittest

class FibonacciTest(unittest.TestCase):
    def test_value_0_should_return_0(self):
        fibonacci = Fibonacci()
        self.assertEqual(0, f
```



Make it compile---create the class Fibonacci

TDD in action – cycle #1

Red phase

Write a failing unit test for a **small** chunk of functionality, which is not implemented yet

Perhaps the test does not even compile at first

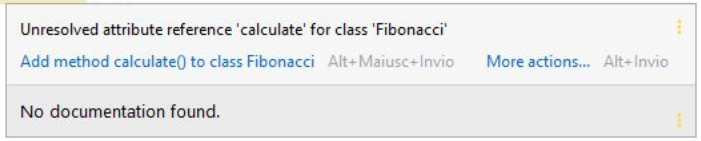
Run the test

Watch the test fail

```
import unittest

from src.Fibonacci import Fibonacci

class FibonacciTest(unittest.TestCase):
    def test_value_0_should_return_0(self):
        fibonacci = Fibonacci()
        self.assertEqual(0, fibonacci.calculate(0))
```



Make it compile---create the method calculate(int)

TDD in action – cycle #1

Red phase

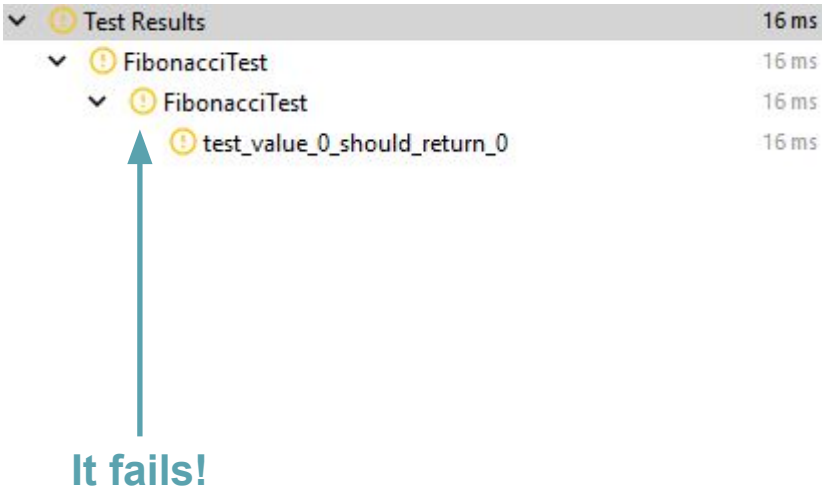
Write a failing unit test for a **small** chunk of functionality, which is not implemented yet

- Perhaps the test does not even compile at first

Run the test

Watch the test fail

```
class Fibonacci:
    def calculate(self, n: int) -> int:
        pass
```



TDD in action – cycle #1

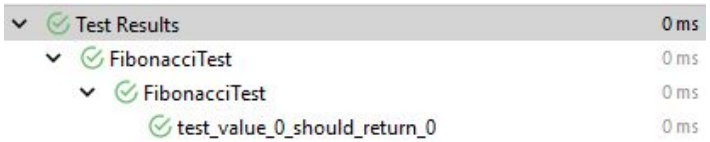
Green phase

Make the test pass **quickly** (i.e., write the minimal amount of code to make the test pass), committing whatever sins necessarily in the process

Run the test (as well as any other test)

Watch the test pass (as well as any other test)

```
class Fibonacci:
    def calculate(self, n: int) -> int:
        return 0
```



Use a FAKE response to make the test pass quickly (if possible)

TDD in action – cycle #1

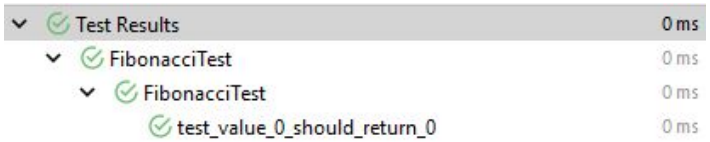
Refactor phase

Eliminate all duplications and smells created in just getting the test to pass

Run all tests

Watch them pass

```
class Fibonacci:
    def calculate(self, n: int) -> int:
        return 0
```



No refactoring opportunities? Ok, I am allowed to skip the refactor phase

TDD in action – cycle #2

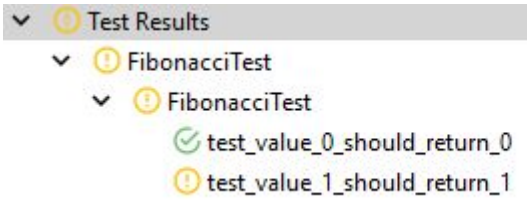
Red phase

Write a failing unit test for a **small** chunk of functionality, which is not implemented yet

Perhaps the test does not even compile at first

Run the test

Watch the test fail




```python
import unittest

from src.Fibonacci import Fibonacci

class FibonacciTest(unittest.TestCase):
    def test_value_0_should_return_0(self):
        fibonacci = Fibonacci()
        self.assertEqual(0, fibonacci.calculate(0))

    def test_value_1_should_return_1(self):
        fibonacci = Fibonacci()
        self.assertEqual(1, fibonacci.calculate(1))
```

$$f_0 = 0 \; if \; n = 0$$
$$\boldsymbol{f_1 = 1 \; if \; n = 1}$$
$$f_n = f_{n-1} + f_{n-2} \; if \; n > 1$$

Chunk of functionality #2

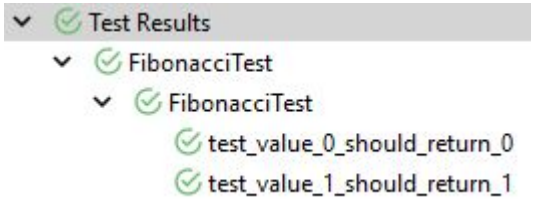
TDD in action – cycle #2

Green phase

Make the test pass **quickly** (i.e., write the minimal amount of code to make the test pass), committing whatever sins necessarily in the process

Run the test (as well as any other test)

Watch the test pass (as well as any other test)




```
class Fibonacci:
    def calculate(self, n: int) -> int:
        return n
```

Replace the fake response with an ABSTRACTION for the considered behaviors

TDD in action – cycle #2

Refactor phase

Eliminate all duplications and smells created in just getting the test to pass

Run all tests

Watch them pass

```python
import unittest

from src.Fibonacci import Fibonacci

class FibonacciTest(unittest.TestCase):
    def setUp(self) -> None:
        self.fibonacci = Fibonacci()

    def test_value_0_should_return_0(self):
        self.assertEqual(0, self.fibonacci.calculate(0))

    def test_value_1_should_return_1(self):
        self.assertEqual(1, self.fibonacci.calculate(1))
```

Test code is code!
Refactor it!

TDD in action – cycle #3

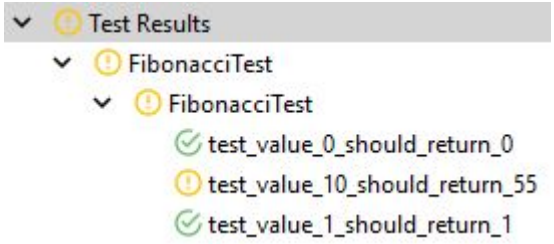
Red phase

Write a failing unit test for a **small** chunk of functionality, which is not implemented yet

- Perhaps the test does not even compile at first

Run the test

Watch the test fail




```python
import unittest

from src.Fibonacci import Fibonacci

class FibonacciTest(unittest.TestCase):
    def setUp(self) -> None:
        self.fibonacci = Fibonacci()

    def test_value_0_should_return_0(self):
        self.assertEqual(0, self.fibonacci.calculate(0))

    def test_value_1_should_return_1(self):
        self.assertEqual(1, self.fibonacci.calculate(1))

    def test_value_10_should_return_55(self):
        self.assertEqual(55, self.fibonacci.calculate(10))
```

$$f_0 = 0 \; if \; n = 0$$
$$f_1 = 1 \; if \; n = 1$$
$$\boldsymbol{f_n = f_{n-1} + f_{n-2} \; if \; n > 1}$$

Chunk of functionality #3

TDD in action – cycle #3

Green phase

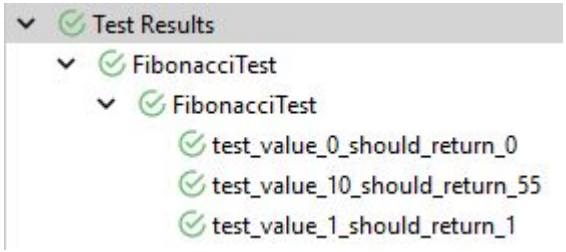
Make the test pass **quickly** (i.e., write the minimal amount of code to make the test pass), committing whatever sins necessarily in the process

Run the test (as well as any other test)

Watch the test pass (as well as any other test)

```python
class Fibonacci:
    def calculate(self, n: int) -> int:
        if n == 0 or n == 1:
            return n

        return self.calculate(n - 1) + self.calculate(n - 2)
```



Are you still unsure about the correctness of your ABSTRACTION?

TRIANGULATION: add another test (e.g., check boundary conditions)!

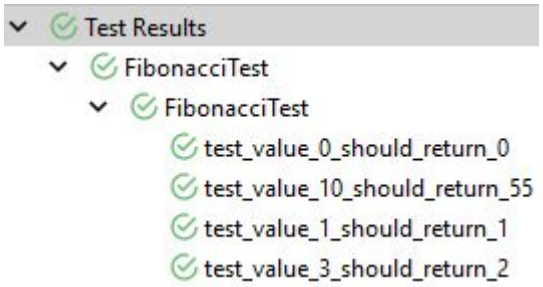
TDD in action – cycle #3

Green phase

Make the test pass **quickly** (i.e., write the minimal amount of code to make the test pass), committing whatever sins necessarily in the process

Run the test (as well as any other test)

Watch the test pass (as well as any other test)




```python
import unittest

from src.Fibonacci import Fibonacci

class FibonacciTest(unittest.TestCase):
    def setUp(self) -> None:
        self.fibonacci = Fibonacci()

    def test_value_0_should_return_0(self):
        self.assertEqual(0, self.fibonacci.calculate(0))

    def test_value_1_should_return_1(self):
        self.assertEqual(1, self.fibonacci.calculate(1))

    def test_value_3_should_return_2(self):
        self.assertEqual(2, self.fibonacci.calculate(3))

    def test_value_10_should_return_55(self):
        self.assertEqual(55, self.fibonacci.calculate(10))
```

TRIAGULATION

TDD in action – cycle #3

Green phase

Make the test pass **quickly** (i.e., write the minimal amount of code to make the test pass), committing whatever sins necessarily in the process

Run the test (as well as any other test)

Watch the test pass (as well as any other test)

```python
import unittest

from src.Fibonacci import Fibonacci

class FibonacciTest(unittest.TestCase):
    def setUp(self) -> None:
        self.fibonacci = Fibonacci()

    def test_value_0_should_return_0(self):
        self.assertEqual(0, self.fibonacci.calculate(0))

    def test_value_1_should_return_1(self):
        self.assertEqual(1, self.fibonacci.calculate(1))

    def test_value_3_should_return_2(self):
        self.assertEqual(2, self.fibonacci.calculate(3))

    def test_value_10_should_return_55(self):
        self.assertEqual(55, self.fibonacci.calculate(10))
```

No refactoring opportunities? Ok, I am allowed to skip the refactor phase

TDD benefits

- Testable code
- Problem understanding
- Code coverage
- Early fault detection
- Regression testing
- Simplified debugging
- System documentation

TDD benefits

Testable code

It forces developers to write tests before the associated production code

TDD benefits

Problem understanding

It helps developers clarify their ideas of what a code segment is actually supposed to do because you have to first write a test for that segment

TDD benefits

Code coverage

Any code segment that you write should have at least one associated test, thus you can be confident that all the code in the system has been exercised

TDD benefits

Early fault detection

Code is tested as it is written so faults are discovered early in the development process

TDD benefits

Regression testing

A test suite is developed incrementally as a system is developed so regression tests ensure that changes to the system have not introduced new faults

TDD benefits

Simplified debugging

When a test fails, it should be obvious where the fault lies, namely the newly written code is the cause of that failure

TDD benefits

System documentation

The tests themselves act as a form of documentation that describe what the tested code should do, thus reading the tests can make it easier to understand the code