Unit Testing

Cyriel Mallart Ludovic Deneuville Rémi Pépin

1 Testing

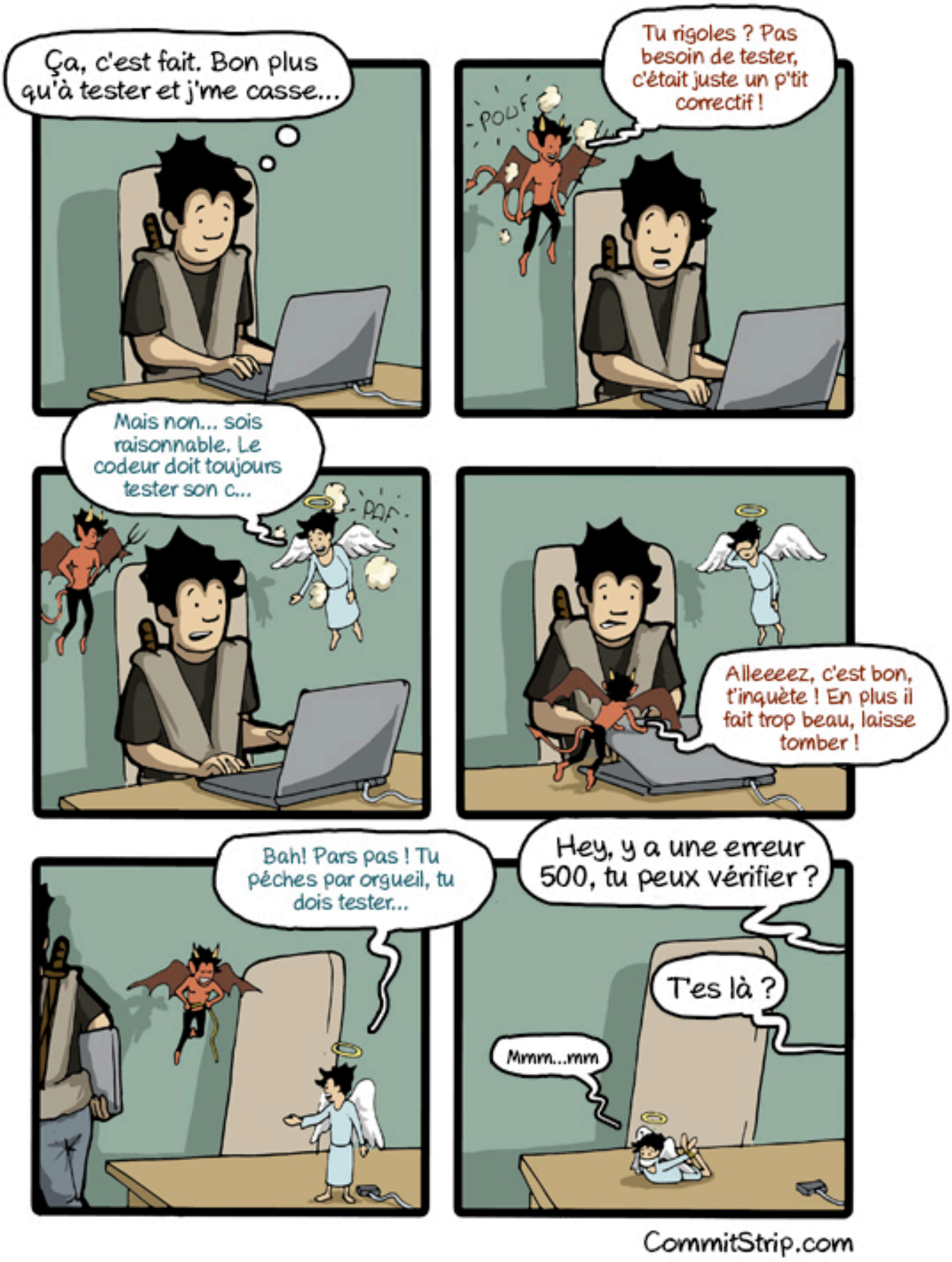

Figure 1: Source : CommitStrip

1.1 Why Test?

- Car crash tests
- Compliance tests
- Quality tests in industry
- Statistical tests

1.2 In Computer Science

- To verify that your program works
- To detect errors
- To avoid regressions (when you modify code)

> **i** Note
>
> No test is perfect, but it still helps eliminate many errors.

1.3 Definition

A test resembles a scientific experiment.

It examines a hypothesis expressed in terms of three elements:

- Input data
- The object to test
- Expected results

This examination is conducted

- under controlled conditions
- with the goal to draw conclusions
- and ideally, be reproducible.

1.4 Test Coverage

- Percentage of functions tested
- **ONE** quality indicator
- Trend, rather than a reliable value

Be lazy : fewer tests, but useful ones!

1.5 Types of Tests

There are many different types of tests, here are the main ones:

- **Unit test**
- Functional test
- Load test
- Integration test
- Penetration test

2 Unit Testing

We will use the `pytest` package to perform our tests in Python.

2.1 A Good Unit Test

- Tests a single functionality
- Isolated
- Reproducible
- Deterministic

2.2 Method to Test

```
class MathOperations:
    """Mathematical Operations"""
    def divide_five_by(self, nb) -> float:
        """Divides the number 5 by a given number.
        Parameters
        ----------
        nb : float or int
            The number by which 5 will be divided.
        Returns
        -------
        float or None
            The result of dividing 5 by the given number.
            If the number is equal to 0, the method returns None.
        """
        if nb != 0:
            return 5 / nb
        else:
            return None
```

2.3 Test Class

Let's create a test class.

To test the nominal case (="normal" case), we:

- Choose an input number
- Call the `divide_five_by()` method
- Verify that the returned value is equal to the expected value

2.4 Nominal Case

```
import pytest
from mathematiques.operations_mathematiques import MathOperations
class TestMathOperations():
    def test_divide_five_by_non_null_nb(self):
        # GIVEN
        nombre = 2
        # WHEN
        resultat = MathOperations().divide_five_by(nombre)
        # THEN
        assert resultat == 2.5
```

2.5 Other Cases

But this is not sufficient!

- The method also has another possible return: `None`
- We also need to test this case

```
    def test_divide_five_by_zero(self):
        # GIVEN
        nombre = 0
        # WHEN
        resultat = MathOperations().diviser_cinq_par(nombre)
        # THEN
        assert resultat is None
```

2.6 What If...

We call the method with this parameter: `divide_five_by("a")`?

You can also write a test to verify that your method indeed returns a `TypeError` exception in this case.

```
    def test_divide_five_by_string(self):
        # GIVEN
        nombre = "a"
        # WHEN / THEN
        with pytest.raises(TypeError):
            MathOperations().diviser_cinq_par(nombre)
```

2.7 Key Takeaways

Unit tests:

- Verify that a method does what it is supposed to do
- Test nominal cases, but also edge cases and errors
- A unit test tests ONE and ONLY ONE thing
- As many unit tests as there are possible returns

3 Mock

- Simulated object that replaces a real component during tests
- Isolating external dependencies = test code independently
- Simulate complex scenarios like network errors

3.1 Mock - Example

```
class PlayerService:
    def create(self, pseudo, mdp, age, mail, fan_pokemon) -> Player:
        new_player = Player(
            pseudo=pseudo,
            mdp=hash_password(mdp, pseudo),
            age=age,
            mail=mail,
```

```
            fan_pokemon=fan_pokemon,
        )

        create_ok = PlayerDao().creer(new_player)

        if create_ok:
            return new_player
        else:
            return None
```

3.2 Mock - Example

```
from unittest.mock import MagicMock
def test_create_ok():
    """Successful creation of Player"""
    # GIVEN
    pseudo, mdp, age, mail, fan_pokemon = "jp", "1234", 15, "z@mail.oo", True
    PlayerDao().creer = MagicMock(return_value=True)

    # WHEN
    player = PlayerService().creer(pseudo, mdp, age, mail, fan_pokemon)

    # THEN
    assert isinstance(player, Player)
    assert player.pseudo == pseudo
```

4 Test-Driven Development (TDD)

4.1 When to Test?

At the beginning!

> Tip
>
> The earlier you test, the more effective and less costly the tests are!

4.2 Test-Driven Development

The best practice:

1. Create the tests
2. Code the function

4.3 Logic

It may seem a bit strange !?

But...

When you code a function, you know before you start:

- What the input parameters will be

- What results you expect as output
- So you already know what to test!

4.4 TDD Practice

- ✅ Improvement of code quality
- ✅ Reduction of bugs
- ❌ then ✅ Time
- ❌ Maintenance of tests

> ! Important
>
> Advantages >>> Disadvantages

4.5 Project tip

Designate the Test Police (or the Quality Police)

Why ?

- Ensure all important methods are tested
- Maintain consistency in testing and documentation
- Promote the test-first thinking

4.6 Benefits on team spirit

- Contribute to code quality without coding A-Z
- One person has an idea of the whole architecture and the pain points
- Easier to review when it’s not your code