RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

cnes
CENTRE NATIONAL D'ÉTUDES SPATIALES

SW TESTING : PRACTICES, AUTOMATION AND AI ASSISTANCE

CNES

DTN/QE/NEO

ANTONY RICARD & FRANCK MAFFENINY

SW PA CONFERENCE - ESTEC

22-25 09 2025

CONTENTS

01 SOFTWARE TESTING PRACTICES for better code
- Code Coverage
- Test first

02 TESTS AUTOMATION
- JIRA XRay plugin – Cucumber in a CI/CD environment

03 AI ASSISTANCE IN DEFINING SW TESTS
- Approach & AI selection : Codestral and Copilot
- Measurement Criterias for a test
- Results

04 CONCLUSION

2019
Study :
« State of the art » in software testing

2020
Study :
White book on software testing part1

2021
Study :
White book on software testing part2

2022
Study :
White book on software testing part3

2024
Study :
Artificial intelligence and Software testing

POC JIRA XRay in the CNES SW Factory

CODE COVERAGE

Software testing is crucial for developing high-quality software, enabling early detection of issues.

Among the most common practices, **code coverage** measures the proportion of code tested, helping teams identify untested part

Hypothesis : JAVA application

"Our policy on automated testing is clear : we want a minimum of 80% of code coverage, aiming for 100% in a near future to make sure we are not having any bug"

- In many cases, tests can be ineffective and not relevant !
  - ✓ Only getters and setters are tested
  - ✓ Coverage does not reveal dead code
  - ✓ Defensive code and logs add complexity but all the code has to be tested
  - ✓ One test can execute a large portion of the code

- Summary
  - ✓ **Focus attention and energy on business code**
  - ✓ **Delete without mercy all forms of dead code**
  - ✓ **Use business, not coverage, to drive tests**
  - ✓ **Even the best tests can fail at detecting bugs**
  - ✓ **All branches should be covered**
  - ✓ **Each piece of business code is tested by one and only one test**

Coverage alone doesn't guarantee test quality and should be complemented by other testing methodologies

TEST FIRST

- What if we start doing things in the right order ?
  - ✓Specify what the software must do (Write test)
  - ✓Make the specification executable (Write code)

test = executable specification

right approach => Test First

- A test leads to no interpretation : **it's red, or green**
- Always up to date, as always executed

How can you **handle frequent** changes, when you are **not confident** that your changes **d'ont break anything** ?

Better approach => **Test First**

(➔ also suitable for refactoring existing code)

TEST FIRST

Test Driven Development

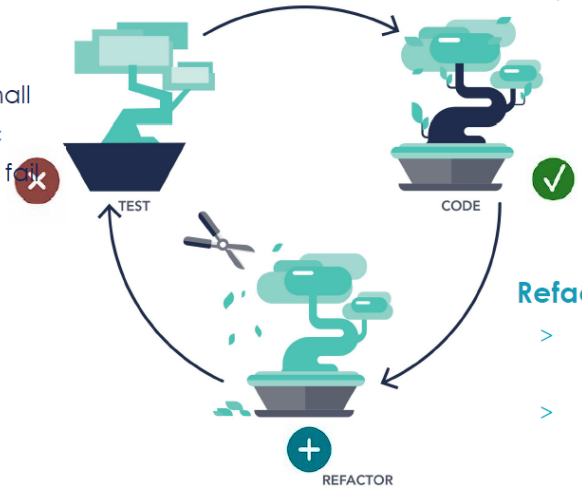
red – green – refactor

is a methodology based on 3 steps

TDD is a methodology based on 3 steps :

Writing test :

- Write a test covering a small part of your business logic
- Test must build, but will fail because you didn't implement the logic



Code :

- Do the minimal piece of code needed to make the test pass

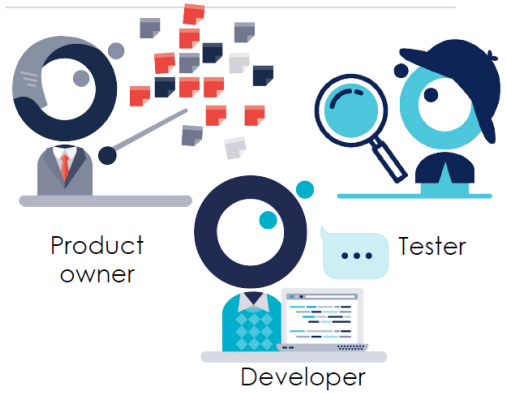
Refactor :

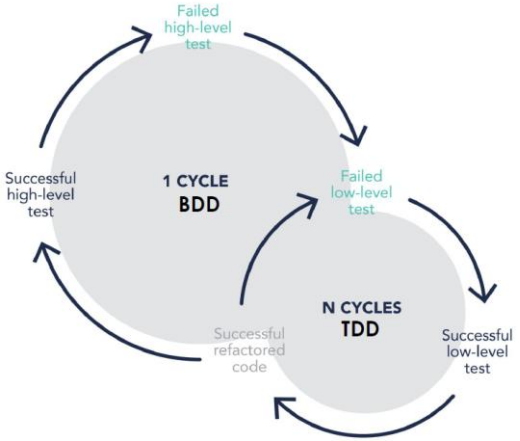
- Now that you have a test covering your code, you can refacto
- Refactoring also includes preparing your code for the next test you will write

Behaviour Driven Development

A peaceful relationship between business and development !

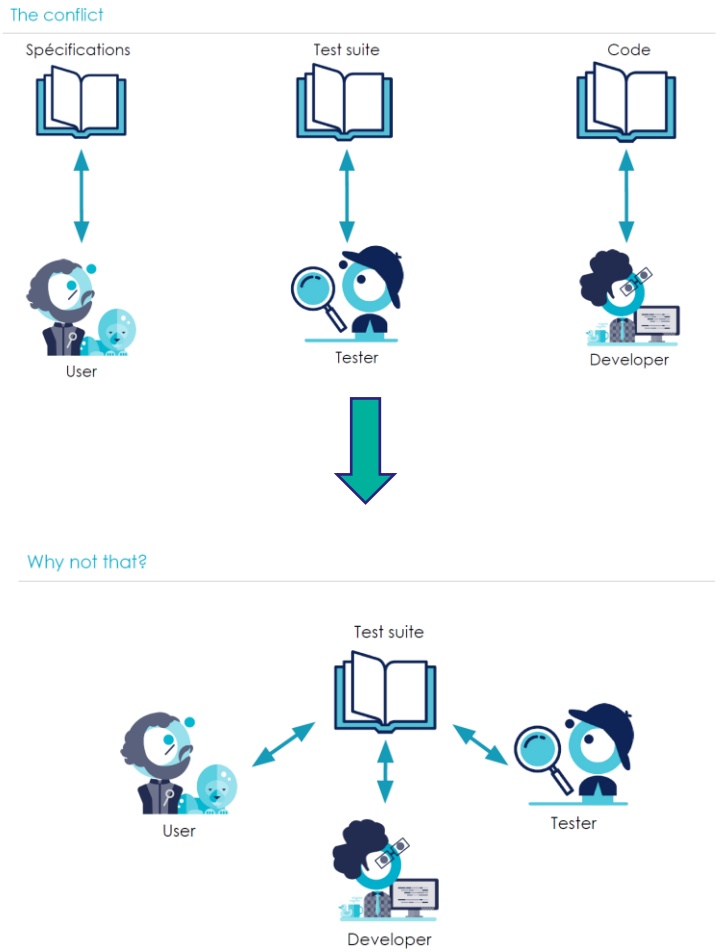
- The most important is **to create a conversation** between all participants
- BDD is very good at providing high-level tests that reflect the business need
  - ✓TDD is very good at being very precise and exhaustive (BDD Loves TDD)





Advantages of TEST FIRST

- Eases the development process :
  - ✓ It reduces the mental load as you work on small part of the code
  - ✓ You can easily handle changes in the code
  - ✓ The feedback is immediate
- Helps to produce clean simpler code
- Not being afraid of change
- Places business at the center
- A peaceful relationship between business and development
- Clarifies the specification earlier, Specification done in a collaborative way
- Living specification
  - ✓ Evolve the test first, then the code to make the new test work
  - ✓ Always up to date



It takes time to be trained to TDD !

TEST FIRST

BDD uses a shared language to describe system behaviors

Gherkin

➔ **the promise to write tests without developers**

```
Feature: Big Fat New Feature
  As an automation panda,
  I want to write a helpful blog about software testing,
  So that others can learn from my experiences.


  # Basic test scenario
  @automation @panda
  Scenario: Verify Google shows pandas
    Given the web browser is at the Google home page
    When the user searches for "panda"
    Then the results page includes results for "panda"


  # More comprehensive test scenario
  @automation @panda @elephant @tiger
  Scenario Outline: Verify Google shows pandas
    Given the web browser is at the Google home page
    When the user searches for "<phrase>"
    Then the results page includes results for "<phrase>"
```

Cucumber

Pros

- More readable by non-devs
- Can produce documentation automatically
  - ✓ + Very useful if required by contract

Recos

- Let your dev team decide to use it or not
- If documentation is required by contract, consider the advantage of a single source of truth

JIRA XRAY PLUGIN IN CI/CD ENVIRONMENT

In SW dev and Agile/Devops context, **tests are costly !**

CNES study objectives :

- Implement a tool to easily manage test scenarios and test campaigns
- Automate test execution in CI (with Gitlab-CI)
- Measure test coverage of requirements, test progress and implementation of requirement

The Criteria :

Ease to use, ease to integrate with a SW Factory, easy to integrate with JIRA, license costs, user support, test management, automation, report generation, requirements coverage

The winners are...

- Cucumber Open
  - ✓ For BDD (test execution in CI/CD environment)
- JIRA Xray
  - ✓ For tests management (creation, storage, report generation, etc.)

- the results of the POC are **conclusive**
- the users (developers and testers) are **enthusiastic**
- Cucumber Open and JIRA Xray plugin tools are **currently being deployed at CNES**

APPROACH AND AI SELECTION : : CODESTRAL AND COPILOT

Hypothesis :

Generating tests with AI on a CNES JAVA open source Application (criticality C, good quality, 96% code coverage)

1. Simple test on utility classes
2. Extensive testing on functional classes
3. Comparison with existing tests and tests generated

AI Tools Selection :

1. Codestral

- LLM specialises in code generation, developed by a French company, guarantees robust data, flexible and customizable, respects EU standards; open source, trained on French data

2. Copilot Assistant - Copilot Message

- Robust model : Based on OpenAI Codex, Easy to configure and use, learning continuous : GitHub is constantly improving Copilot by taking into accountuser feedback, RGPD. Copilot is available at an affordable price

MEASUREMENT CRITERIAS FOR A TEST

1

Test name clear

The name of the test must be descriptive and explicit, clearly indicating the functionality being tested and the conditions being verified, thus facilitating understanding and maintenance

2

Compliance with a 3-part parts

The test must follow a standard structure, such as Gherkin(Given, When, Then) or AAA (Arrange, Act, Assert), to ensure readability and good organisation.

3

FIRST

A good test must be fast and independent, repeatable, self-validating and exhaustive, to guarantee efficient execution and verified automatic results.

4

Coverage & Mutation testing

The test must aim for high code coverage and be robust against mutations, effectively detecting changes in behaviour.

5

Compilation / Assert

The proposed code must compile and the tests must pass. This metric will tell you how many tests will require additional work.

Observations

- On utility class
  - ✓ The AIs proposed more tests than the existing tests
  - ✓ Even if the coverage and naming of the tests are better, fewer mutants have been killed
  - ✓ Further work was required to refine and validate these proposals.
- On functional class
  - ✓ Prompts must be well designed to be as effective as existing tests
  - ✓ The simple prompts generated few tests, but rivalled the existing tests (which had more tests)
  - ✓ More complex rework was required to obtain satisfactory results
- Overall
  - ✓ Generations of tests have proposed tests that complement the existing tests
  - ✓ But they are not enough to make all the production code reliable
  - ✓ The suggestions made by Codestral and Copilot Message proved to be more relevant overall.

Conclusions

- The AIs generated tests that required numerous corrections to enable the classes to be compiled and tests to be validated
- The AIs respected the format that we had requested on the names of the tests and their structuring in the prompt.
- Test generation was fairly robust on the 'surface', and implemented boundary cases by managing return exceptions
- The AI was unable to generate sufficiently satisfactory tests on its own.
- However, its assistance to the developer brings productivity gains, particularly in the case of error management

CONCLUSION

- **Modern software testing practices, automation, and AI are revolutionizing how teams ensure software quality**
- Test Driven Development and Behaviour Driven Development provide a foundation for quality tests
- Tools like JIRA XRay, GitLab CI, and Cucumber make testing faster and more reliable
- AI tools like Copilot and Codestral enhance test efficiency by making test definition smarter and more precise
- All **these integrated approaches** ensure that software is functional, robust, efficient, and production-ready


THANK YOU
QUESTIONS ?
© CNES/DERAMECOURT Arnaud, 2017