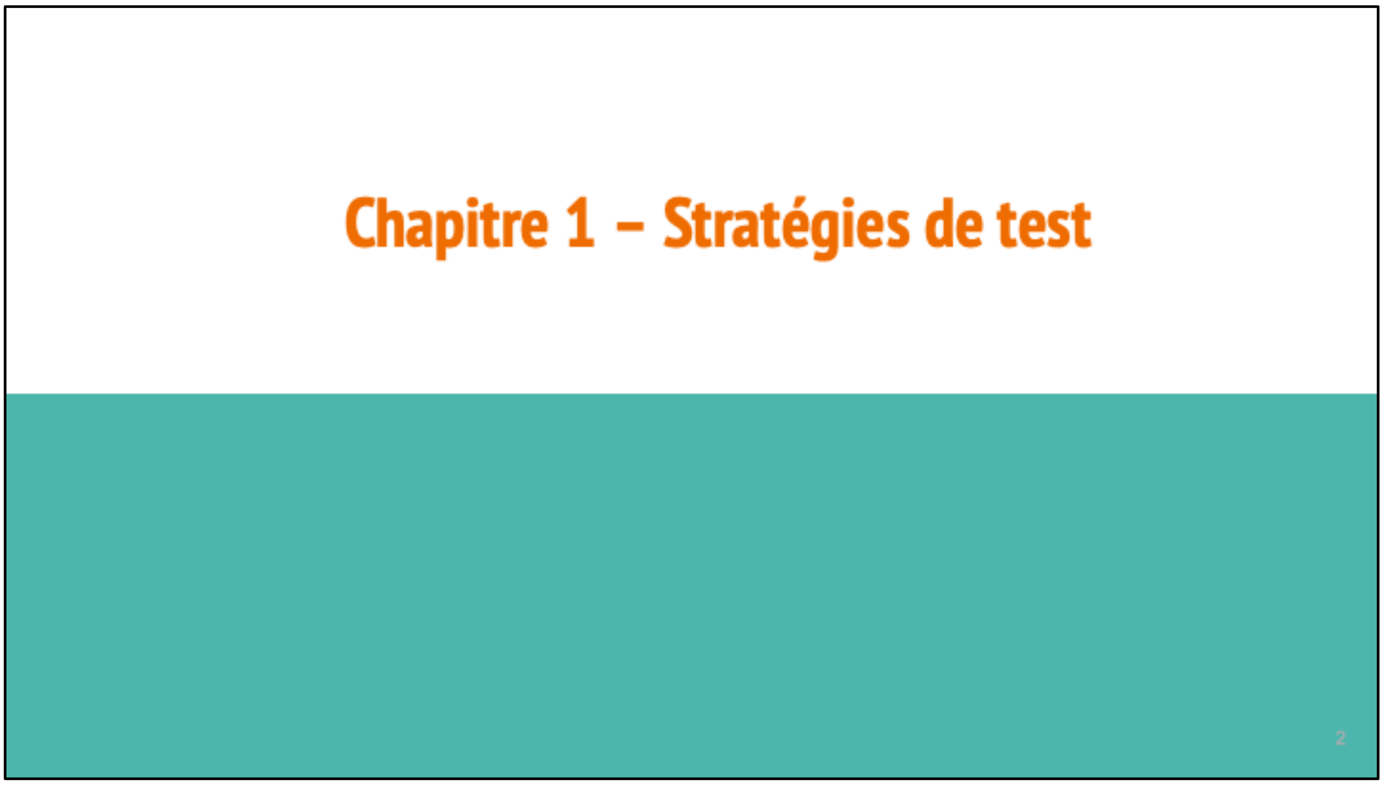
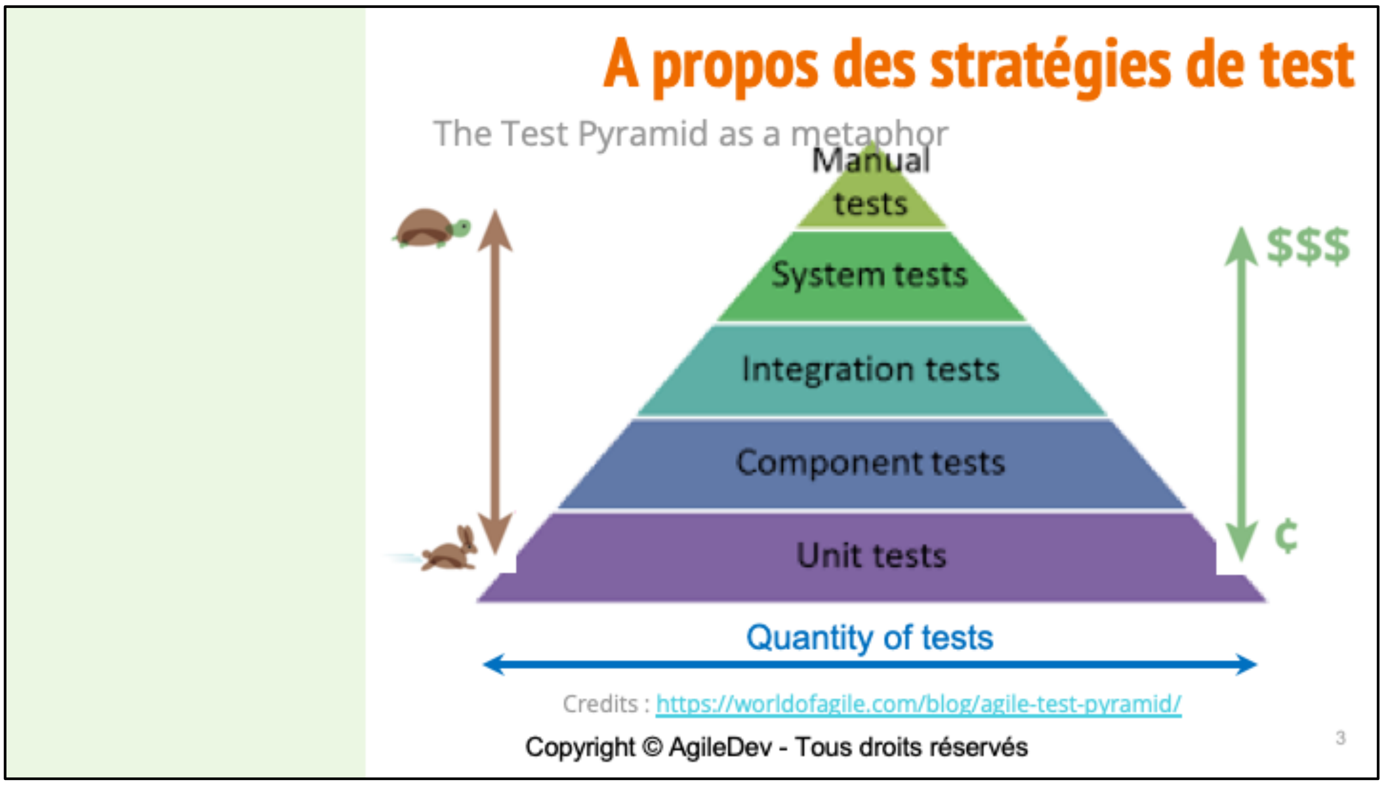
Mastère CTO & Tech Lead - MT4

Tests Unitaires

Stratégies de test et tests unitaires

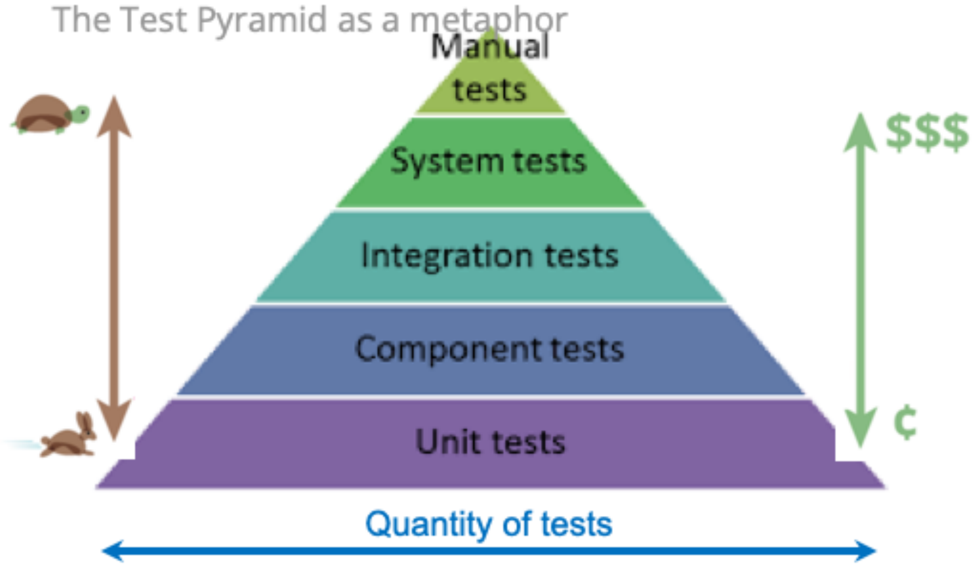


Chapitre 1 – Stratégies de test



A propos des stratégies de test

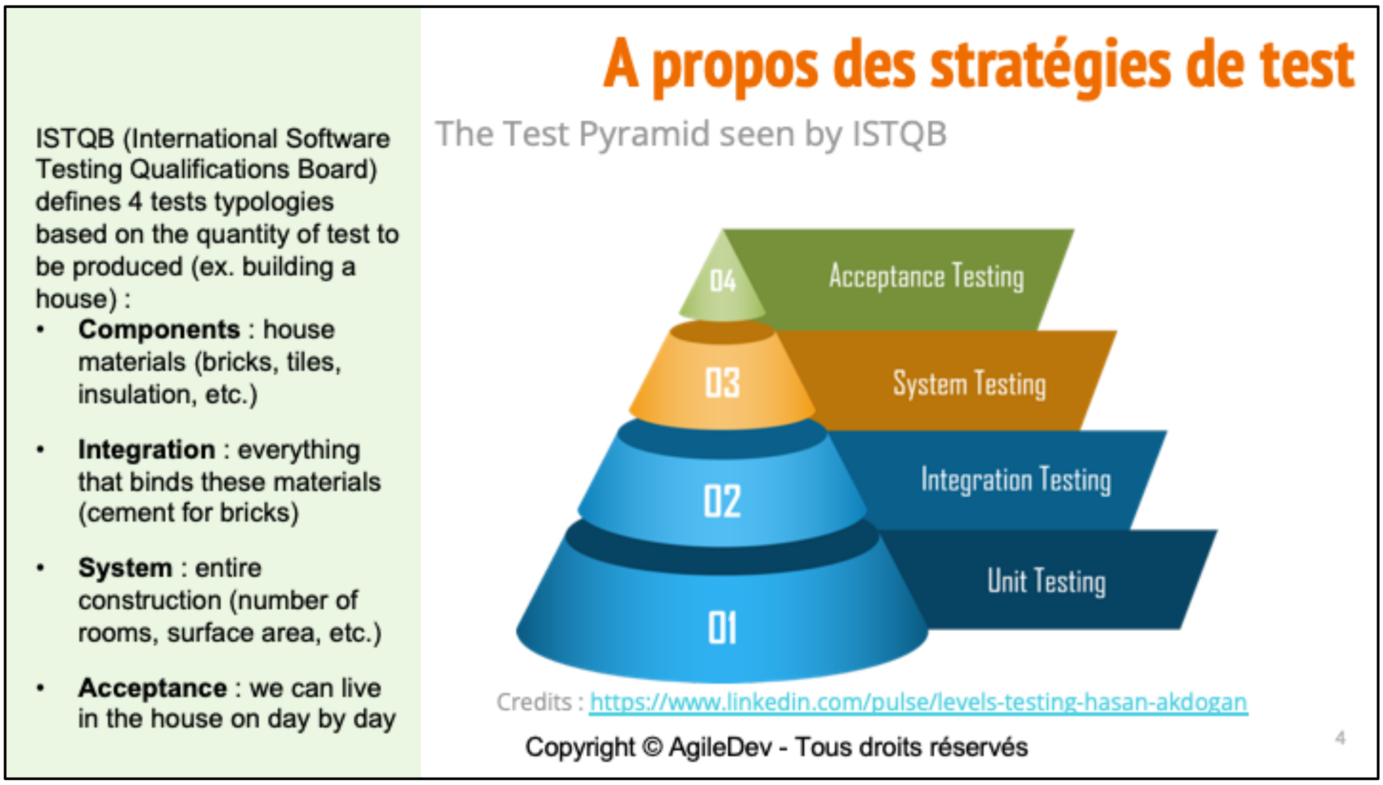
The Test Pyramid as a metaphor



Credits : https://worldofagile.com/blog/agile-test-pyramid/

Credits : https://worldofagile.com/blog/agile-test-pyramid/

Putting different testing practices on a pyramid by assuming that :
- the width stands for the quantity of a given testing practice compared to another
- the more you go to the top of the pyramid, the more executing a test takes time, and writing / maintaining them has a cost
- we can also figure out a level of isolation : high at the bottom and low at the top
- we can also identify a level of accuracy : again high at the bottom and low at the top

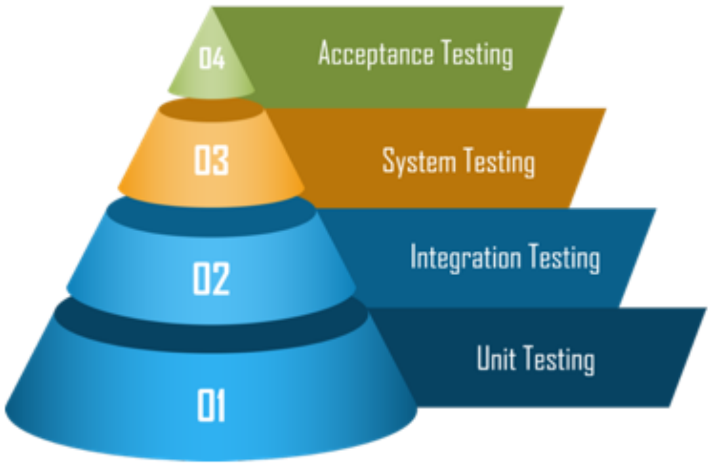


A propos des stratégies de test

ISTQB (International Software Testing Qualifications Board) defines 4 tests typologies based on the quantity of test to be produced (ex. building a house) :

- **Components** : house materials (bricks, tiles, insulation, etc.)
- **Integration** : everything that binds these materials (cement for bricks)
- **System** : entire construction (number of rooms, surface area, etc.)
- **Acceptance** : we can live in the house on day by day

The Test Pyramid seen by ISTQB

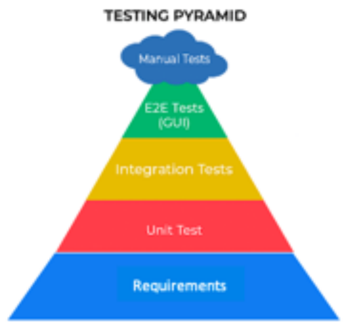


Credits : https://www.linkedin.com/pulse/levels-testing-hasan-akdogan

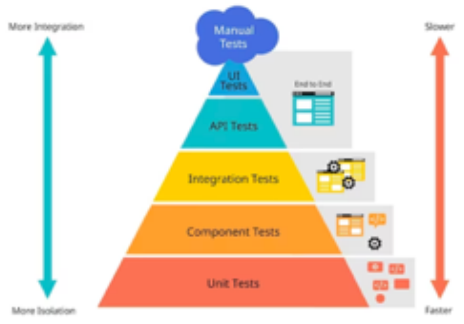
Credits : https://www.linkedin.com/pulse/levels-testing-hasan-akdogan

A propos des stratégies de test

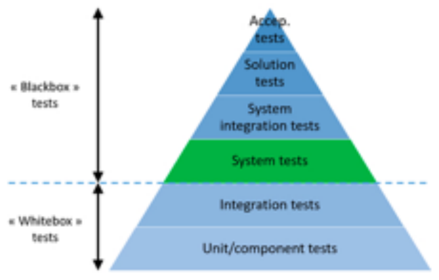
Further pyramid representations



Credits

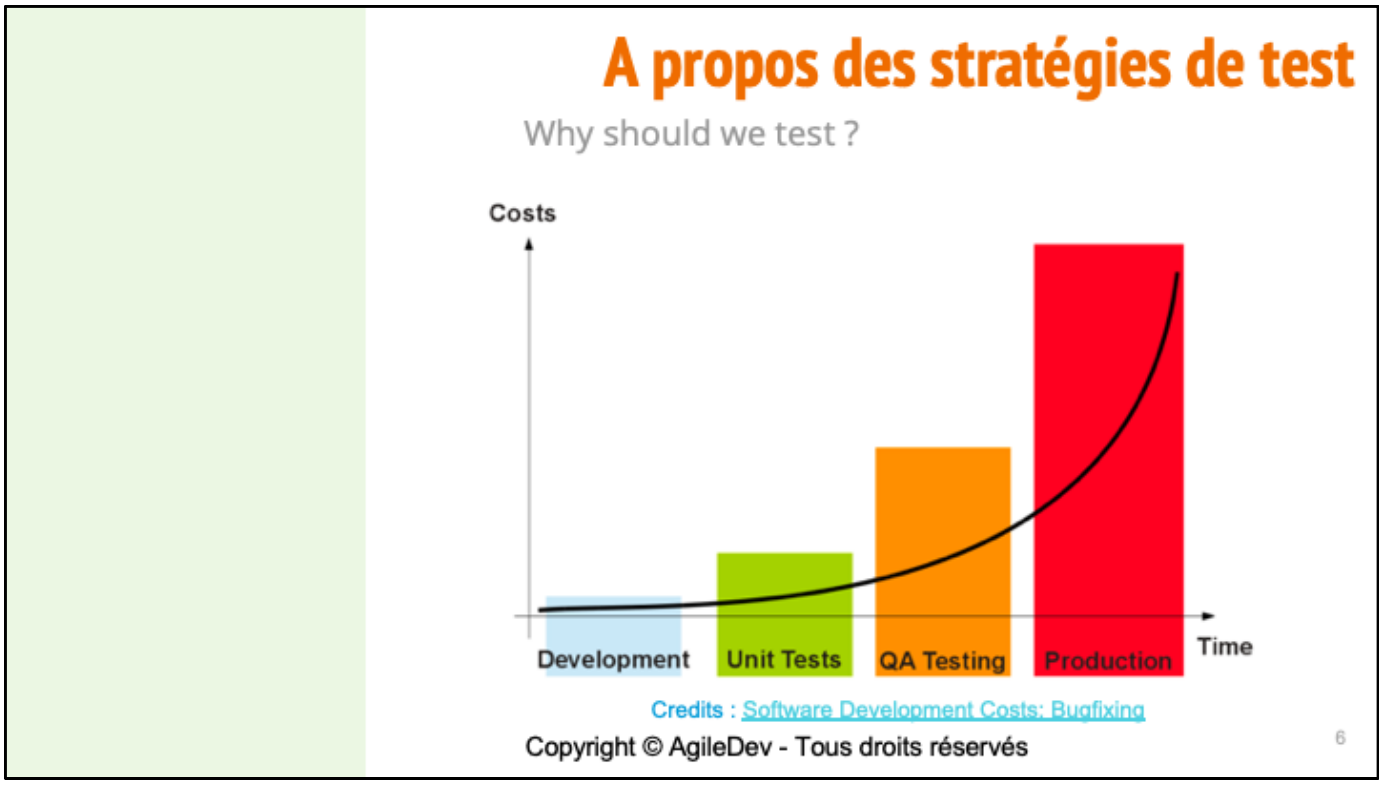


Credits



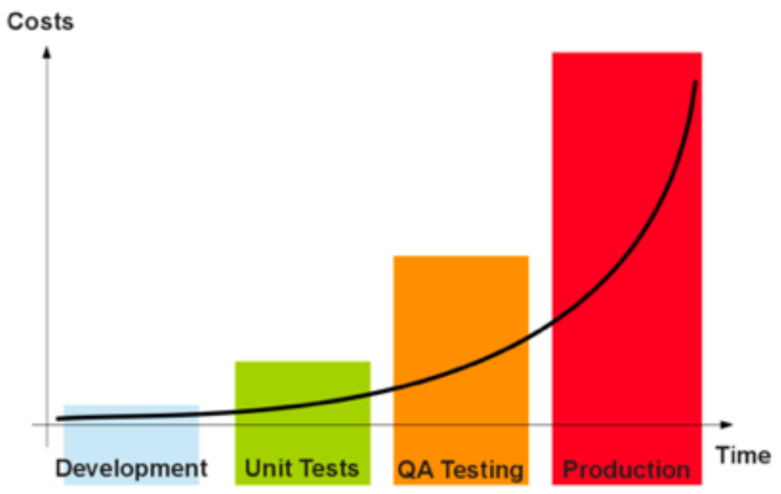
Credits

- Showing manual tests at the top of the pyramid, so to tell they don't really follow the whole trend (Credits : https://www.compass-testservices.com/shift-left-testing-with-early-model-based-testing-embt/)
  - Still having high cost and taking a lot of time to execute
  - But not automated and their quantity depends on the maturity of the team / organization
- The second one focuses on isolation / integration dimension, as well as speed. It also gives some details about System / Acceptance test layer (Credits : https://fr.freepik.com/vecteurs-premium/pyramide-tests-tests-interface-utilisateur-tests-integration-tests-unitaires_50393283.htm)
- The last one makes appear another categorization of test families : whitebox and blackbox (Credits : https://www.neosoft.fr/nos-publications/blog-tech/cleancode-videostore-refacto-oncle-bob-tests/)



A propos des stratégies de test

Why should we test ?



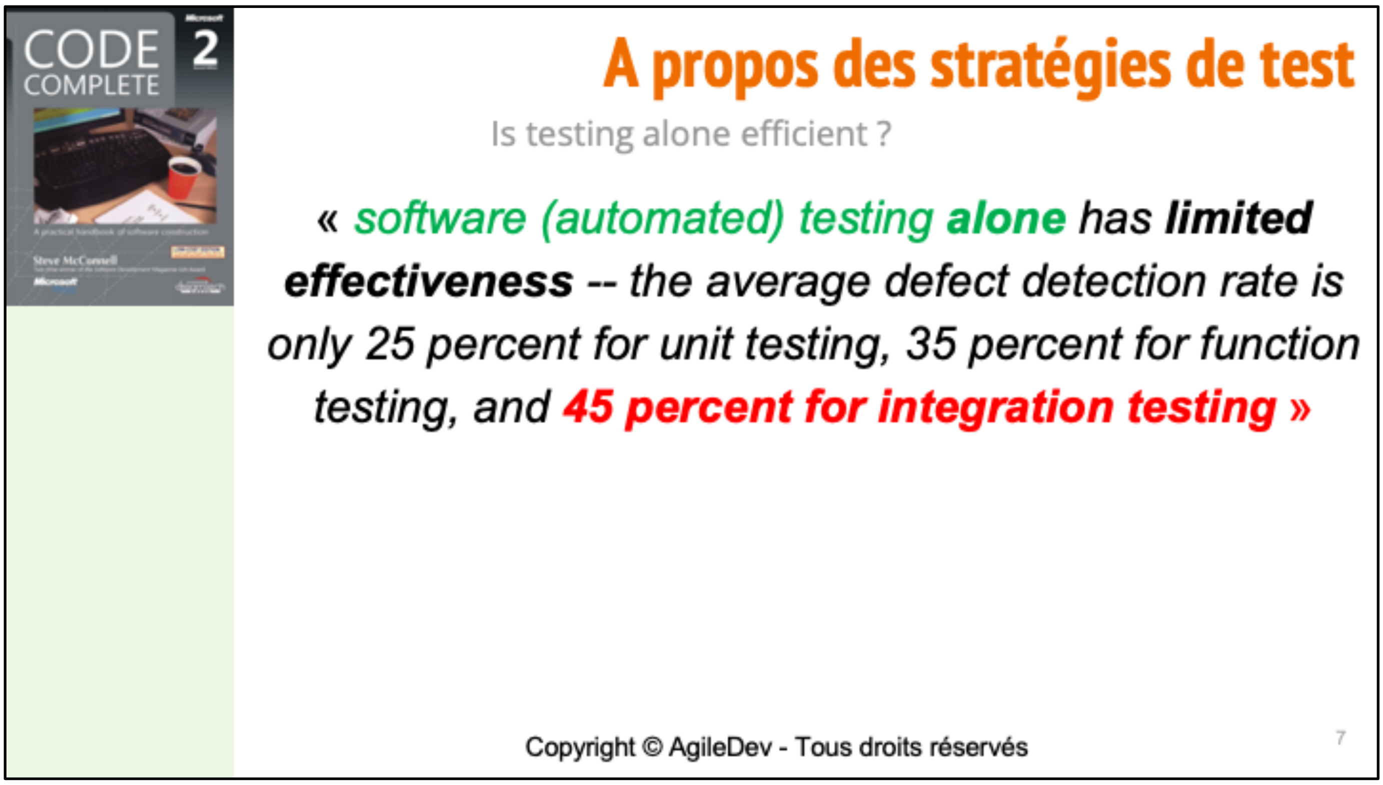
Credits : Software Development Costs: Bugfixing

Credits : https://blog.pdark.de/2012/07/21/software-development-costs-bugfixing/

Here we can see some correlation between the cost of fixing a bug and the phase in development lifecycle which enlighten the disorder.
The curve applies for a given feature which development started a time 0, each rectangle width is here to illustrate that the cost is increasing as soon as we started the activity

It is particularly relevant for the production part, meaning that the cost of remediation has a given value the day we put the feature in production, and this cost will increase dramatically with the amount of time the feature is in production (ex. after 5 releases it will be more difficult to identify root cause if a user encounters a faillure related to something put in production months ago).

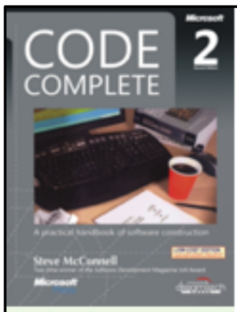
This representation is not to say that we should avoid QA Testing or Unit Tests and allocate more time during development phase, because it does not tell us how many issues are raised during each phase.



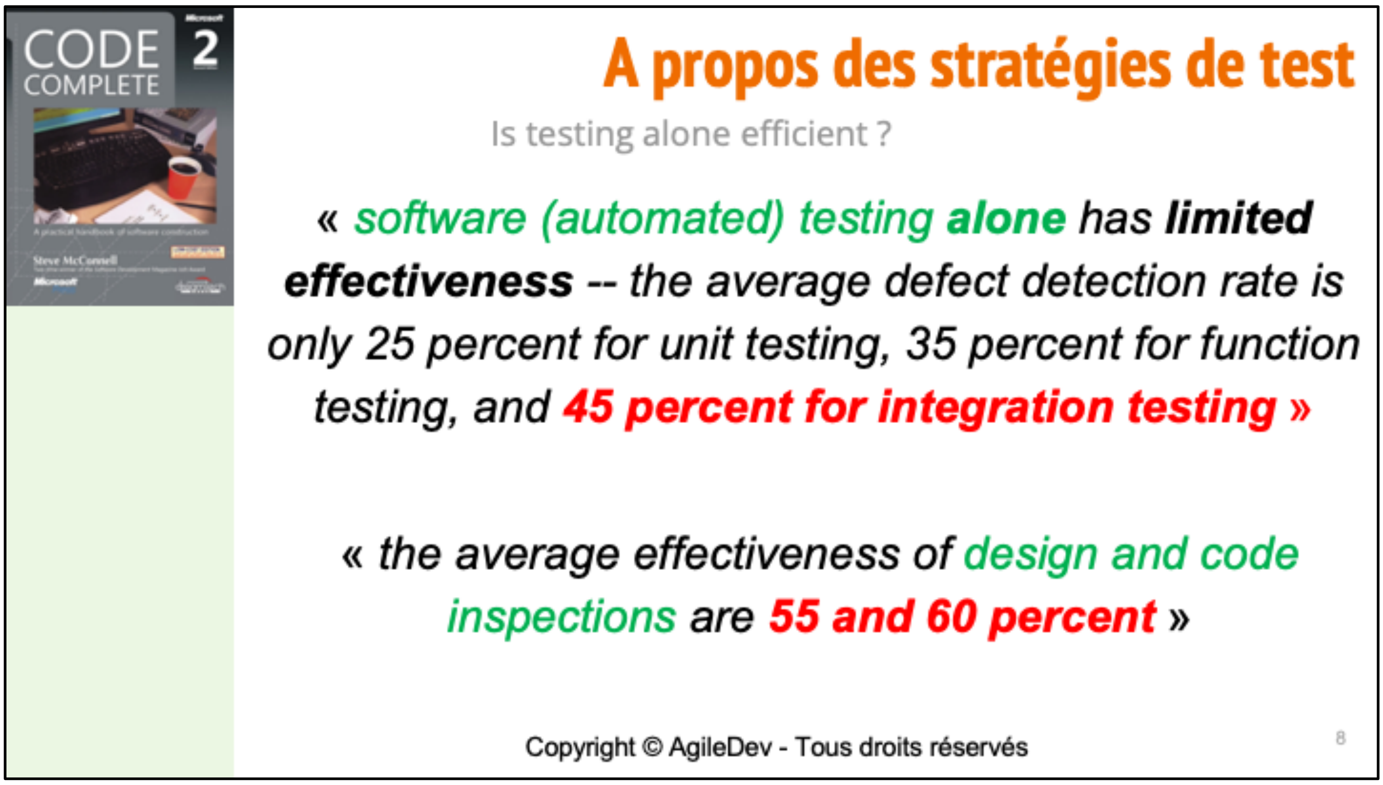
A propos des stratégies de test

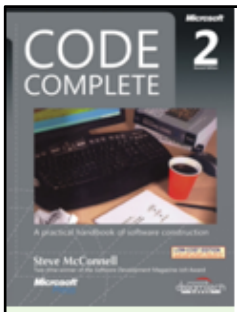
Is testing alone efficient ?

« software (automated) testing **alone** has **limited effectiveness** -- *the average defect detection rate is only 25 percent for unit testing, 35 percent for function testing, and* ***45 percent for integration testing*** »



Code complete was written in 1993 by Steve McConnell, american engineer born in 1963.
Steve was designated in 1998 amongst the 3 most influent people in software industry (with Linus Torvalds and Bill Gates)





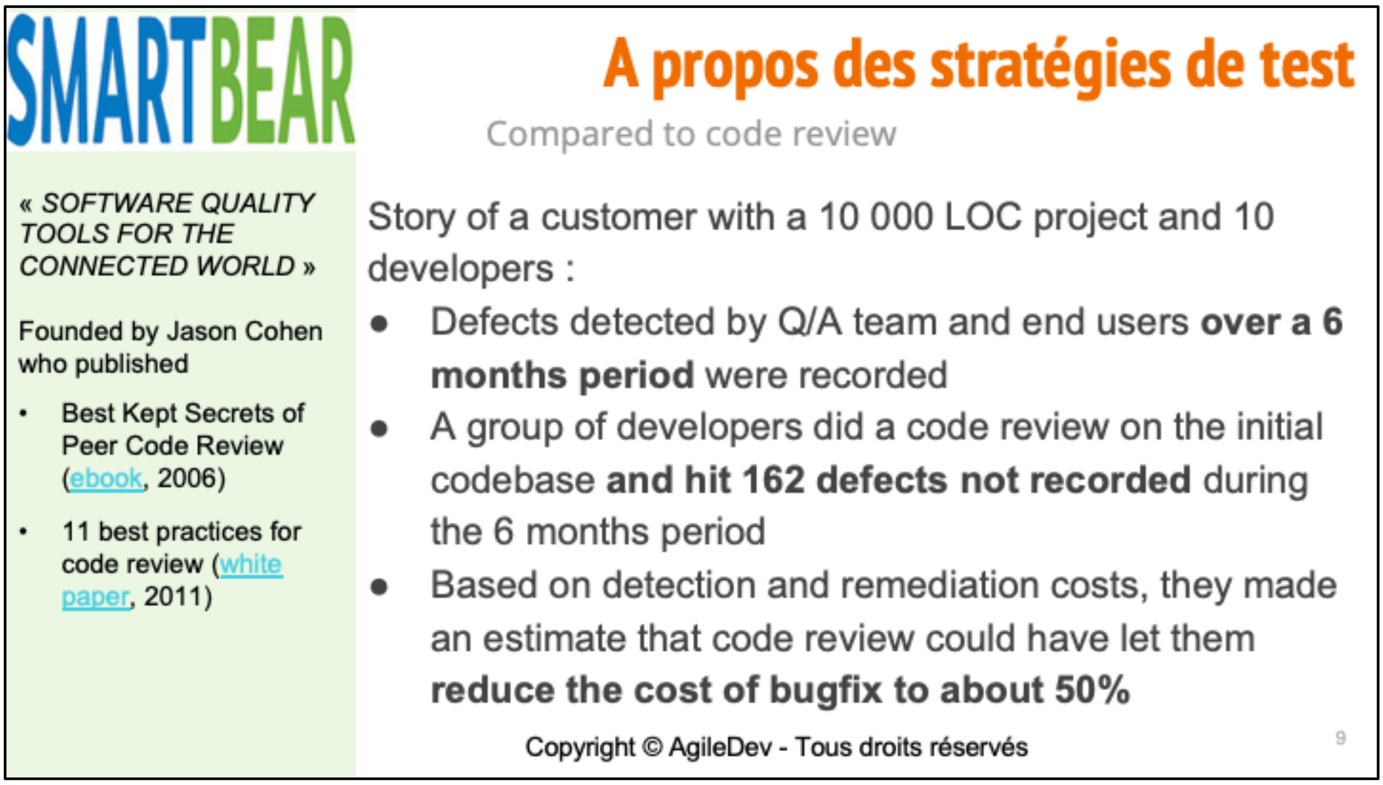
A propos des stratégies de test

Is testing alone efficient ?

« *software (automated) testing **alone** has **limited effectiveness** -- the average defect detection rate is only 25 percent for unit testing, 35 percent for function testing, and **45 percent for integration testing** »*

*« the average effectiveness of design and code inspections are **55 and 60 percent** »*

- Writing automated test might only help reach at best 50% of defect detections
- Code review (and design / architecture reviews) can reach up to 60% of defect detection



SMARTBEAR

« SOFTWARE QUALITY TOOLS FOR THE CONNECTED WORLD »

Founded by Jason Cohen who published

- Best Kept Secrets of Peer Code Review (ebook, 2006)
- 11 best practices for code review (white paper, 2011)

A propos des stratégies de test

Compared to code review

Story of a customer with a 10 000 LOC project and 10 developers :

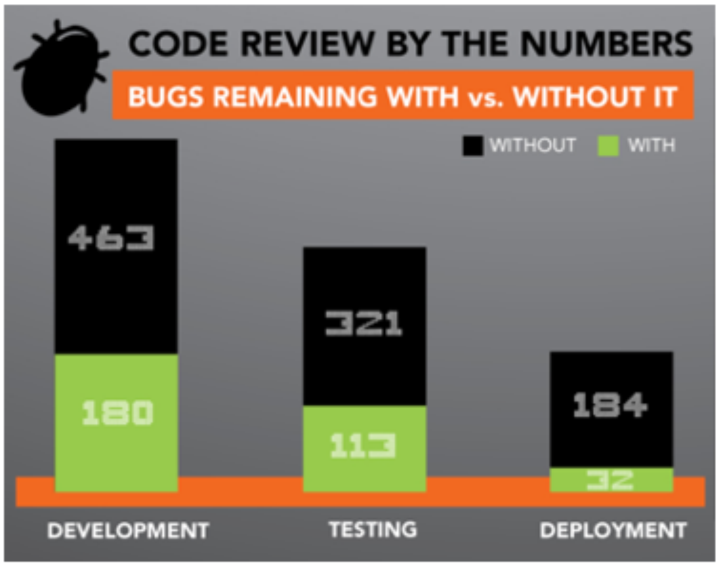
- Defects detected by Q/A team and end users **over a 6 months period** were recorded
- A group of developers did a code review on the initial codebase **and hit 162 defects not recorded** during the 6 months period
- Based on detection and remediation costs, they made an estimate that code review could have let them **reduce the cost of bugfix to about 50%**

Ebook : https://smartbear.com/resources/ebooks/best-kept-secrets-of-code-review/
(direct access : https://static1.smartbear.co/smartbear/media/pdfs/best-kept-secrets-of-peer-code-review_redirected.pdf)
White paper :
http://viewer.media.bitpipe.com/1253203751_753/1284482743_310/11_Best_Practices_for_Peer_Code_Review.pdf

SMARTBEAR

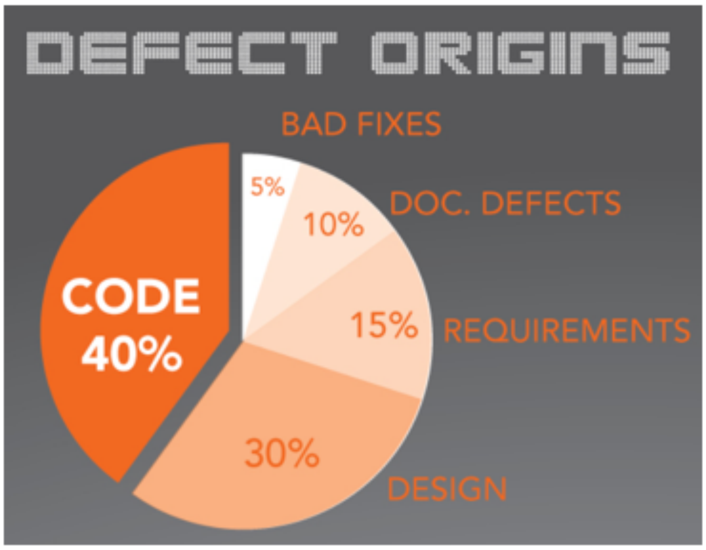
A propos des stratégies de test

On behalf code review



A propos des stratégies de test

Where do defects come from ?



If you put together requirements and design you reach 45% which is more than the 40% coming from code => sounds interesting to hear that we should put more effort to (requirements + design) defect reduction than for code.

A propos des stratégies de test

Let us put code review aside (but keep it in mind … if not already)

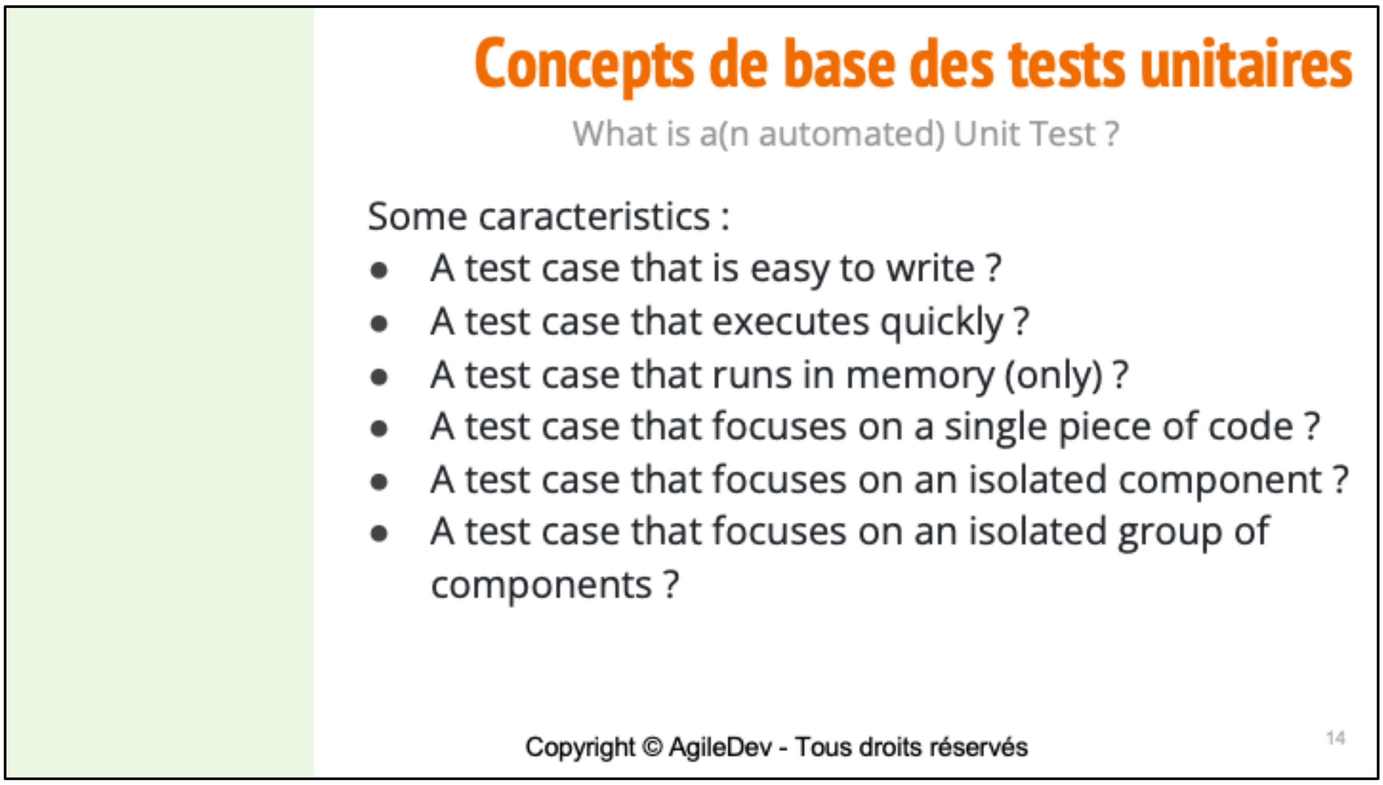
La relecture de code : avant tout des pratiques !

Eric SIBER - @esiber – 25/04/2019

https://www.youtube.com/watch?v=FzaLio0mP1Y

You can also find the slideshow on https://www.slideshare.net/ericsiber

Chapitre 2 – Concepts de base des tests unitaires



Concepts de base des tests unitaires

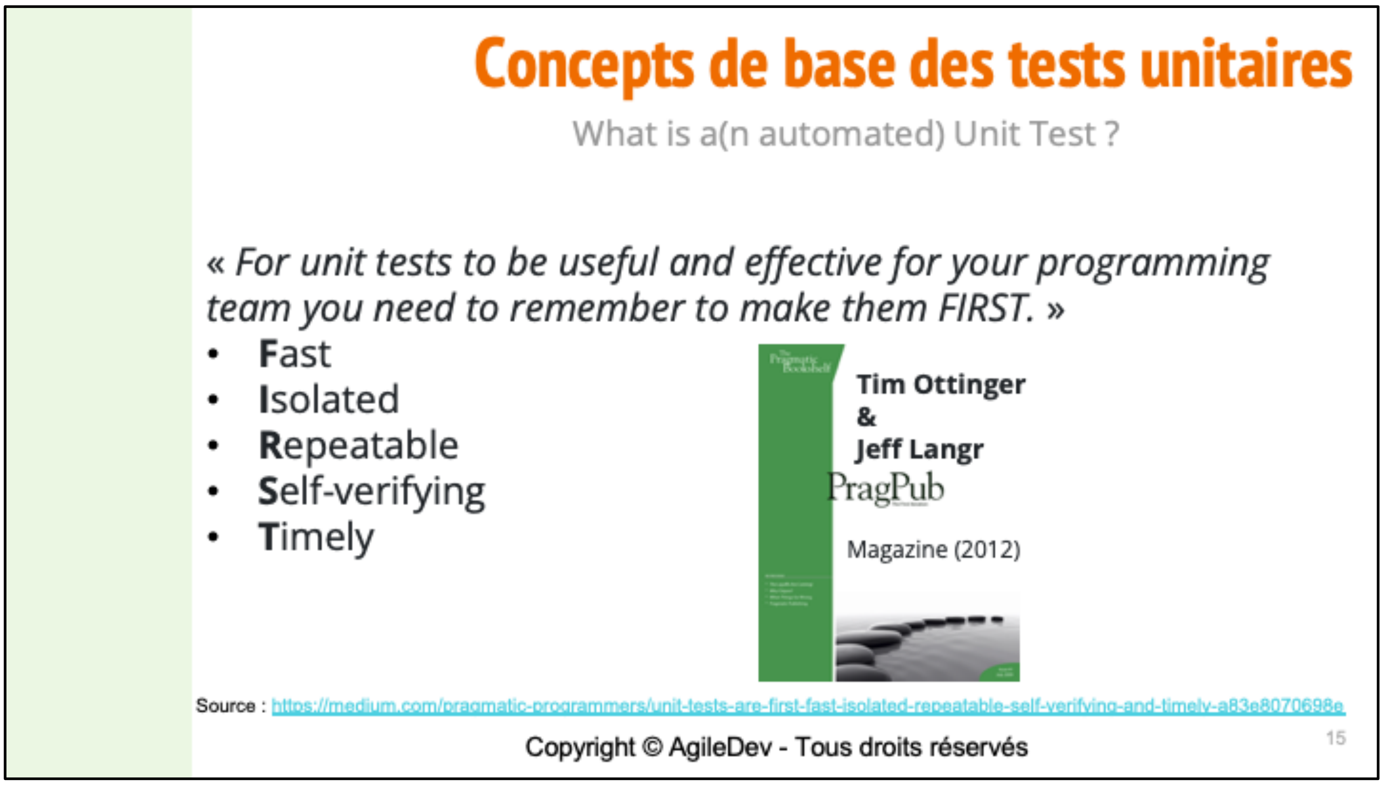
What is a(n automated) Unit Test ?

Some caracteristics :

- A test case that is easy to write ?
- A test case that executes quickly ?
- A test case that runs in memory (only) ?
- A test case that focuses on a single piece of code ?
- A test case that focuses on an isolated component ?
- A test case that focuses on an isolated group of components ?

Based on the pyramid, we should first focus on Unit Tests
… depending on what we define as being a component …

Definition given by Smartbear



Concepts de base des tests unitaires

What is a(n automated) Unit Test ?

« For unit tests to be useful and effective for your programming team you need to remember to make them FIRST. »

- **F**ast
- **I**solated
- **R**epeatable
- **S**elf-verifying
- **T**imely

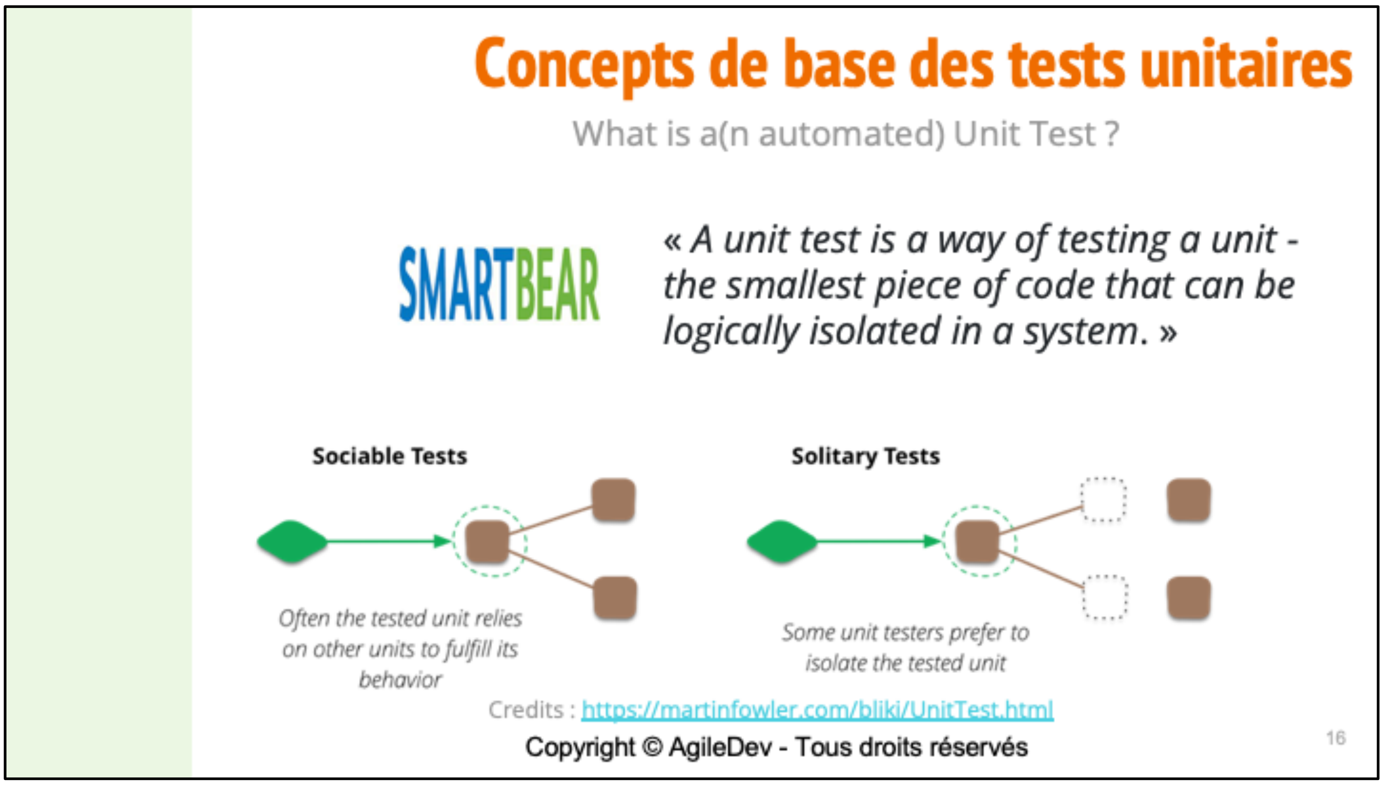
Tim Ottinger
&
Jeff Langr

Magazine (2012)

Source : https://medium.com/pragmatic-programmers/unit-tests-are-first-fast-isolated-repeatable-self-verifying-and-timely-a83e8070698e

Source : https://medium.com/pragmatic-programmers/unit-tests-are-first-fast-isolated-repeatable-self-verifying-and-timely-a83e8070698e

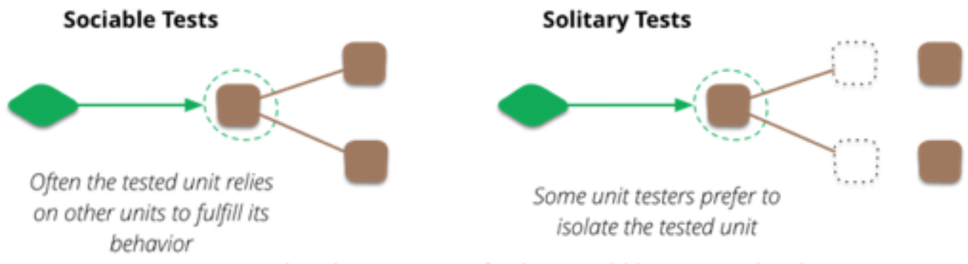
S => don't fall in the trap to avoid asserting anything …
T => favor Test First against Test After



Concepts de base des tests unitaires

What is a(n automated) Unit Test ?

« A unit test is a way of testing a unit - the smallest piece of code that can be logically isolated in a system. »


Sociable Tests
Often the tested unit relies on other units to fulfill its behavior
Solitary Tests
Some unit testers prefer to isolate the tested unit

Credits : https://martinfowler.com/bliki/UnitTest.html

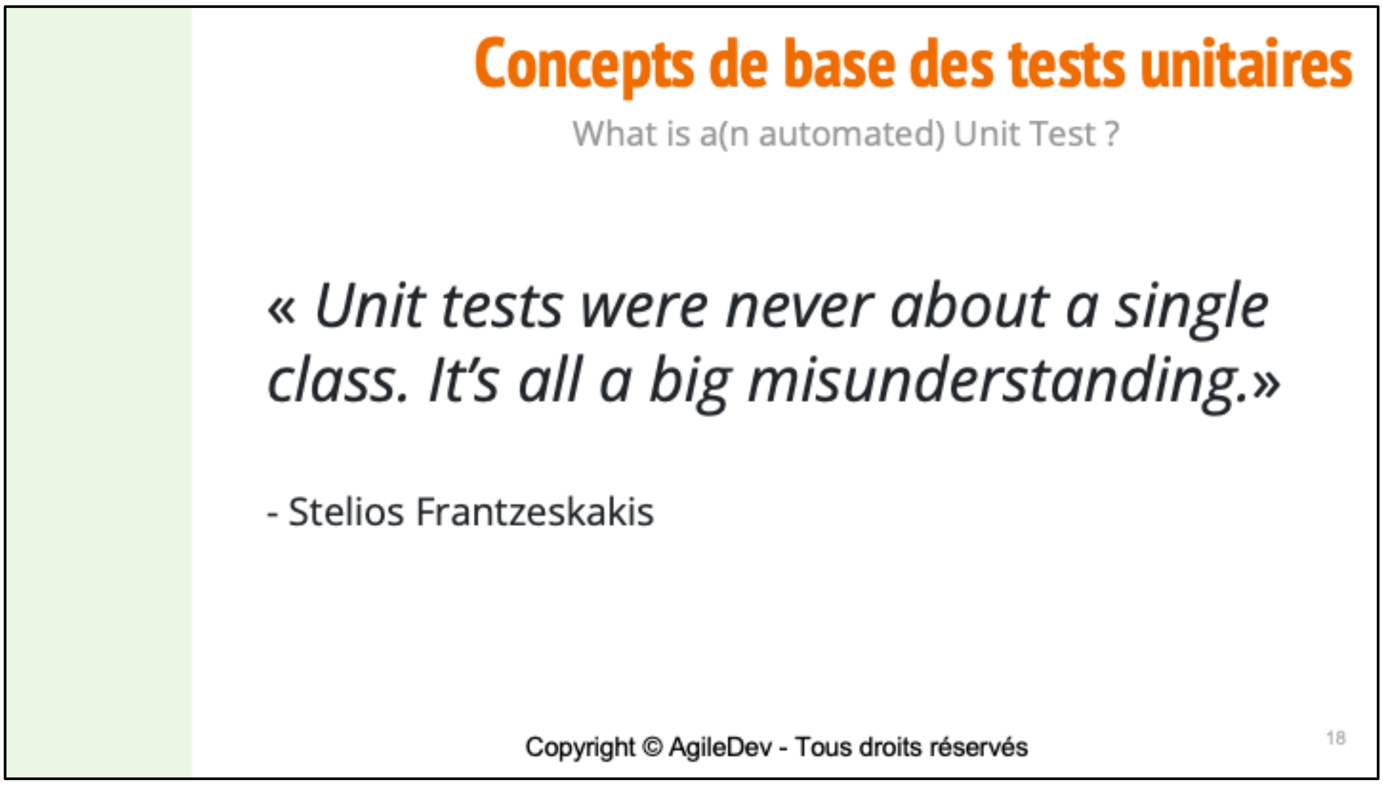
Credits : https://martinfowler.com/bliki/UnitTest.html

Concepts de base des tests unitaires

What is a(n automated) Unit Test ?

« Tests that can run ***independent*** *of one another. The unit of isolation is the test and not the classes under test. »*

« A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work. »



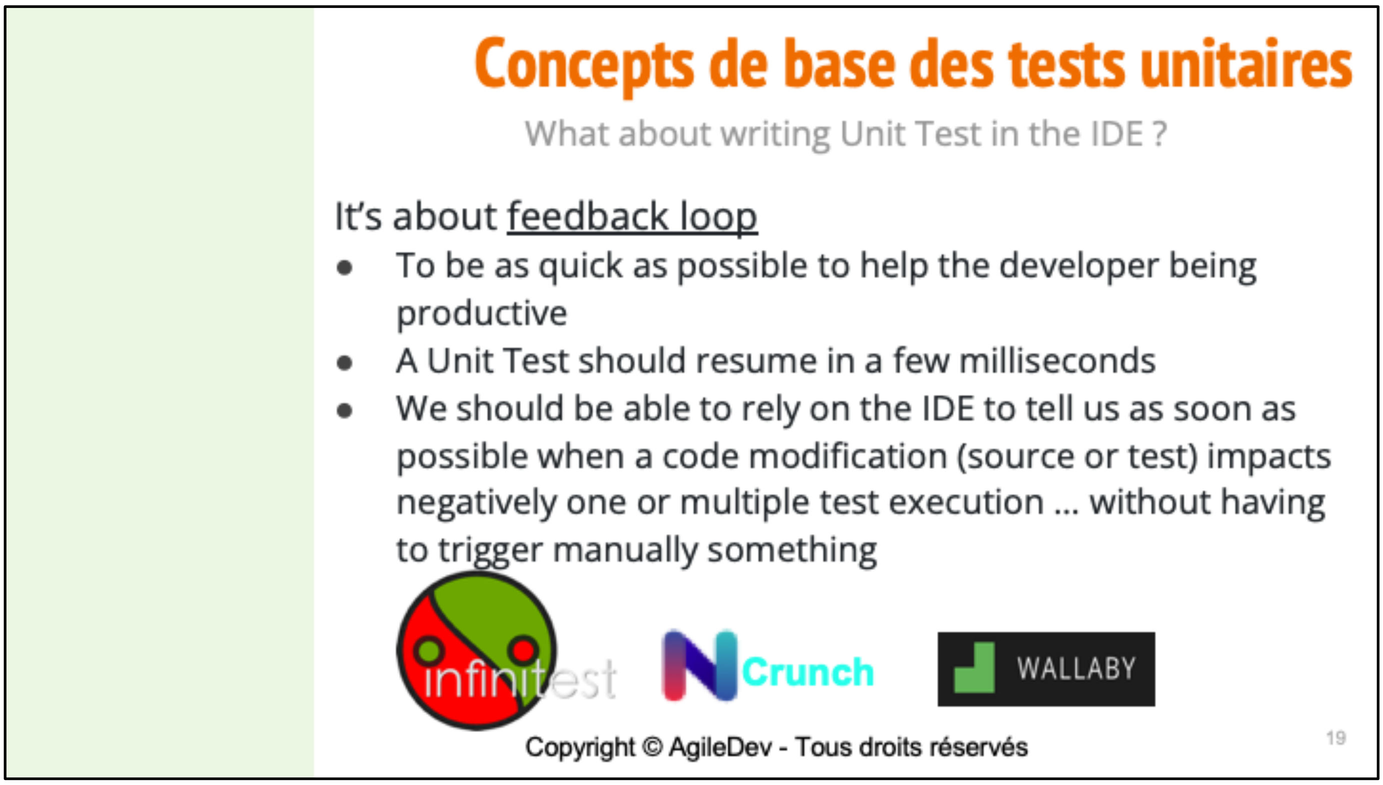
Concepts de base des tests unitaires

What is a(n automated) Unit Test ?

« Unit tests were never about a single class. It's all a big misunderstanding.»

- Stelios Frantzeskakis

We could use the term Component testing rather than Unit testing or just keep in mind that Unit does not mean method or class



Concepts de base des tests unitaires

What about writing Unit Test in the IDE ?

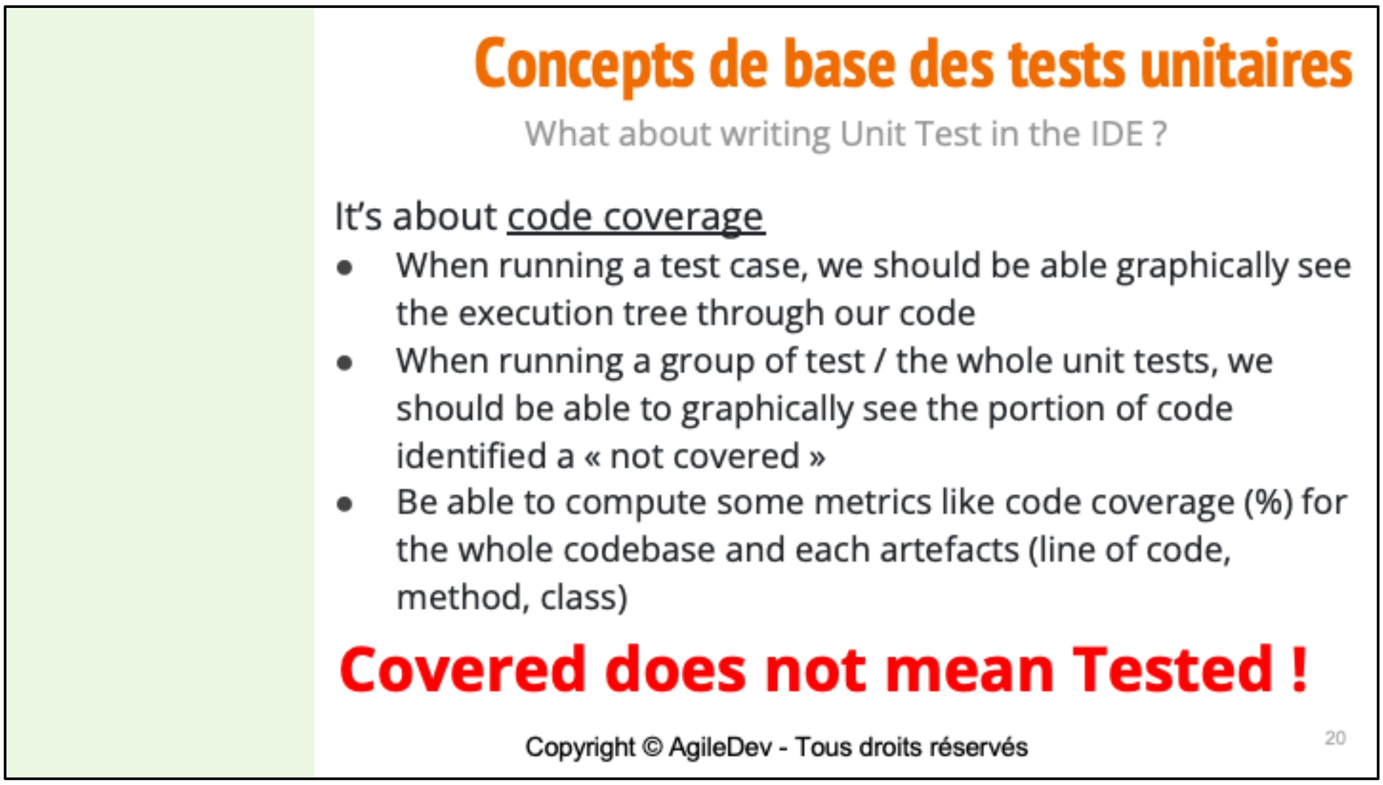
It's about feedback loop

- To be as quick as possible to help the developer being productive
- A Unit Test should resume in a few milliseconds
- We should be able to rely on the IDE to tell us as soon as possible when a code modification (source or test) impacts negatively one or multiple test execution ... without having to trigger manually something

Could be native like the --watch flag with Jest

Infinitest is a continuous test runner for the JVM with a plugin for IntelliJ : https://plugins.jetbrains.com/plugin/3146-infinitest
Ncrunch is a continuous test runner for C# with
- a plugin for JetBrains Rider : https://plugins.jetbrains.com/plugin/24054-ncrunch
- a plugin for Visual Studio : https://marketplace.visualstudio.com/items?itemName=NCrunch.NCrunchforVisualStudio

Wallaby.js is a continuous test runner for TypeScript / Javascript based code, with a plugin for IntelliJ / WebStorm : https://plugins.jetbrains.com/plugin/15742-wallaby



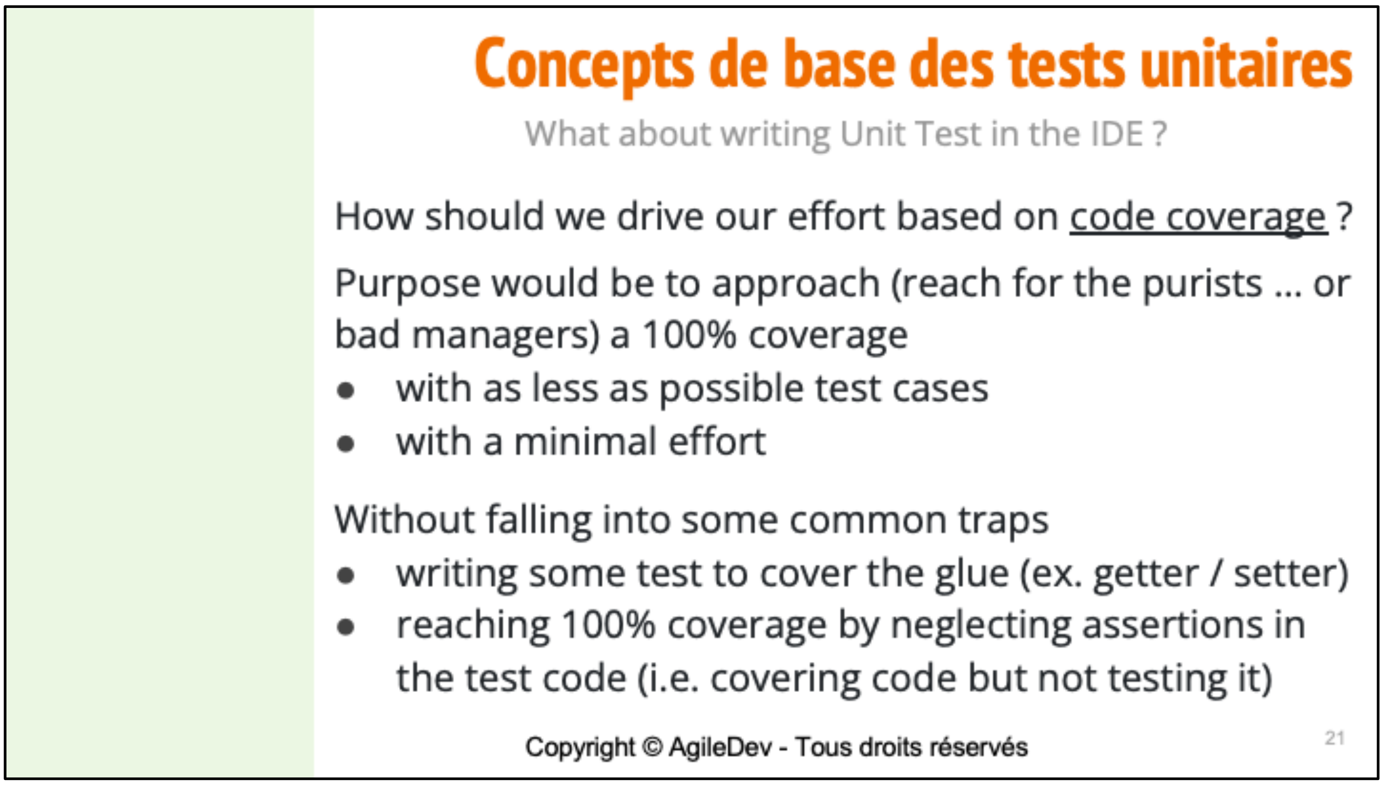
Concepts de base des tests unitaires

What about writing Unit Test in the IDE ?

It's about code coverage

- When running a test case, we should be able graphically see the execution tree through our code
- When running a group of test / the whole unit tests, we should be able to graphically see the portion of code identified a « not covered »
- Be able to compute some metrics like code coverage (%) for the whole codebase and each artefacts (line of code, method, class)

Covered does not mean Tested !



Concepts de base des tests unitaires

What about writing Unit Test in the IDE ?

How should we drive our effort based on <u>code coverage</u> ?

Purpose would be to approach (reach for the purists … or bad managers) a 100% coverage

- with as less as possible test cases
- with a minimal effort

Without falling into some common traps

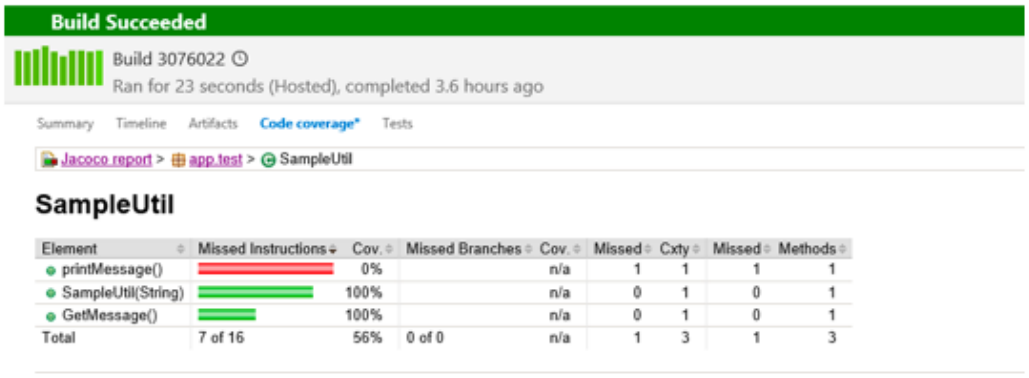
- writing some test to cover the glue (ex. getter / setter)
- reaching 100% coverage by neglecting assertions in the test code (i.e. covering code but not testing it)

We will see later how we can handle this coverage trap

Concepts de base des tests unitaires

What about writing Unit Test in the IDE ?

How should we drive our effort based on code coverage ?



Build Succeeded

Build 3076022

Ran for 23 seconds (Hosted), completed 3.6 hours ago

Summary Timeline Artifacts Code coverage* Tests

Jacoco report > app.test > SampleUtil

SampleUtil

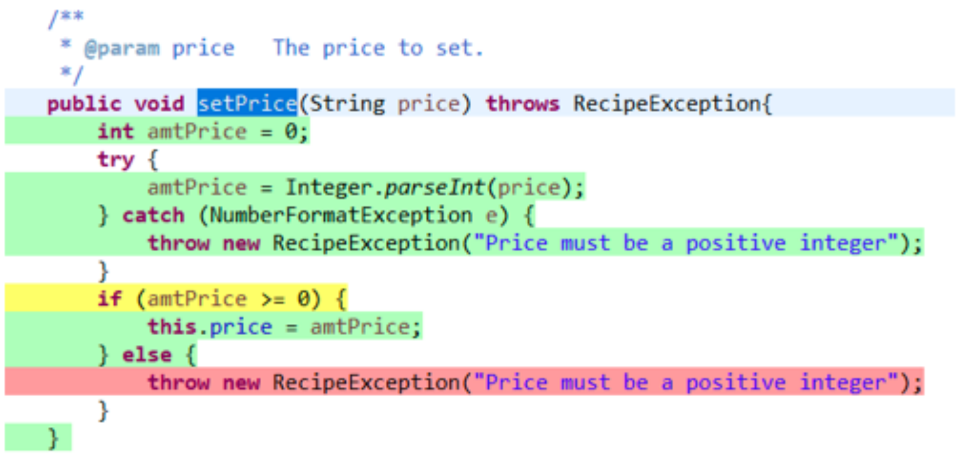
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Methods
printMessage()	🟥	0%		n/a	1	1	1	1
SampleUtil(String)	🟩	100%		n/a	0	1	0	1
GetMessage()	🟩	100%		n/a	0	1	0	1
Total	7 of 16	56%	0 of 0	n/a	1	3	1	3

Here it’s not IDE but CI but we would find same output (HTML file) on a local build

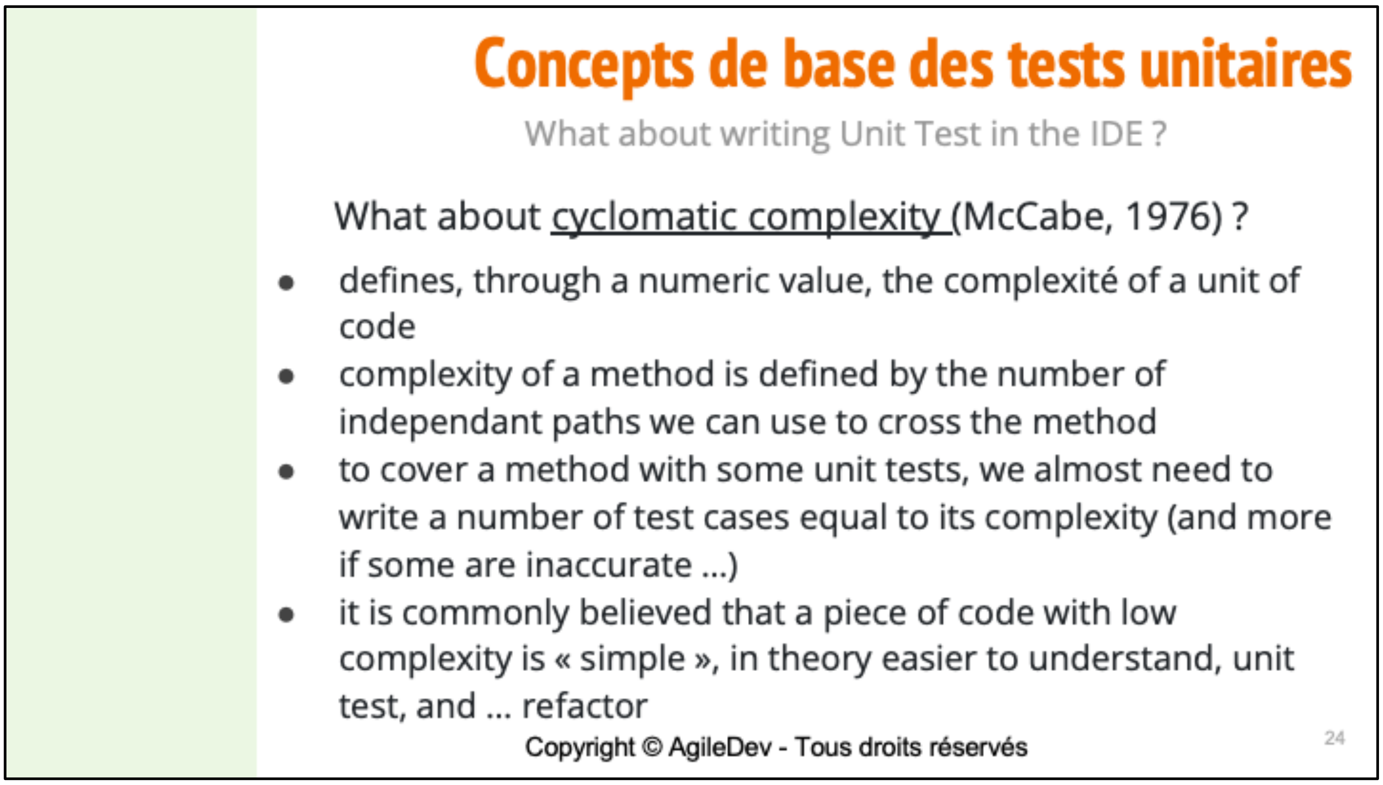
Concepts de base des tests unitaires

What about writing Unit Test in the IDE ?

How should we drive our effort based on code coverage ?



```
    /**
     * @param price   The price to set.
     */
    public void setPrice(String price) throws RecipeException{
        int amtPrice = 0;
        try {
            amtPrice = Integer.parseInt(price);
        } catch (NumberFormatException e) {
            throw new RecipeException("Price must be a positive integer");
        }
        if (amtPrice >= 0) {
            this.price = amtPrice;
        } else {
            throw new RecipeException("Price must be a positive integer");
        }
    }
```

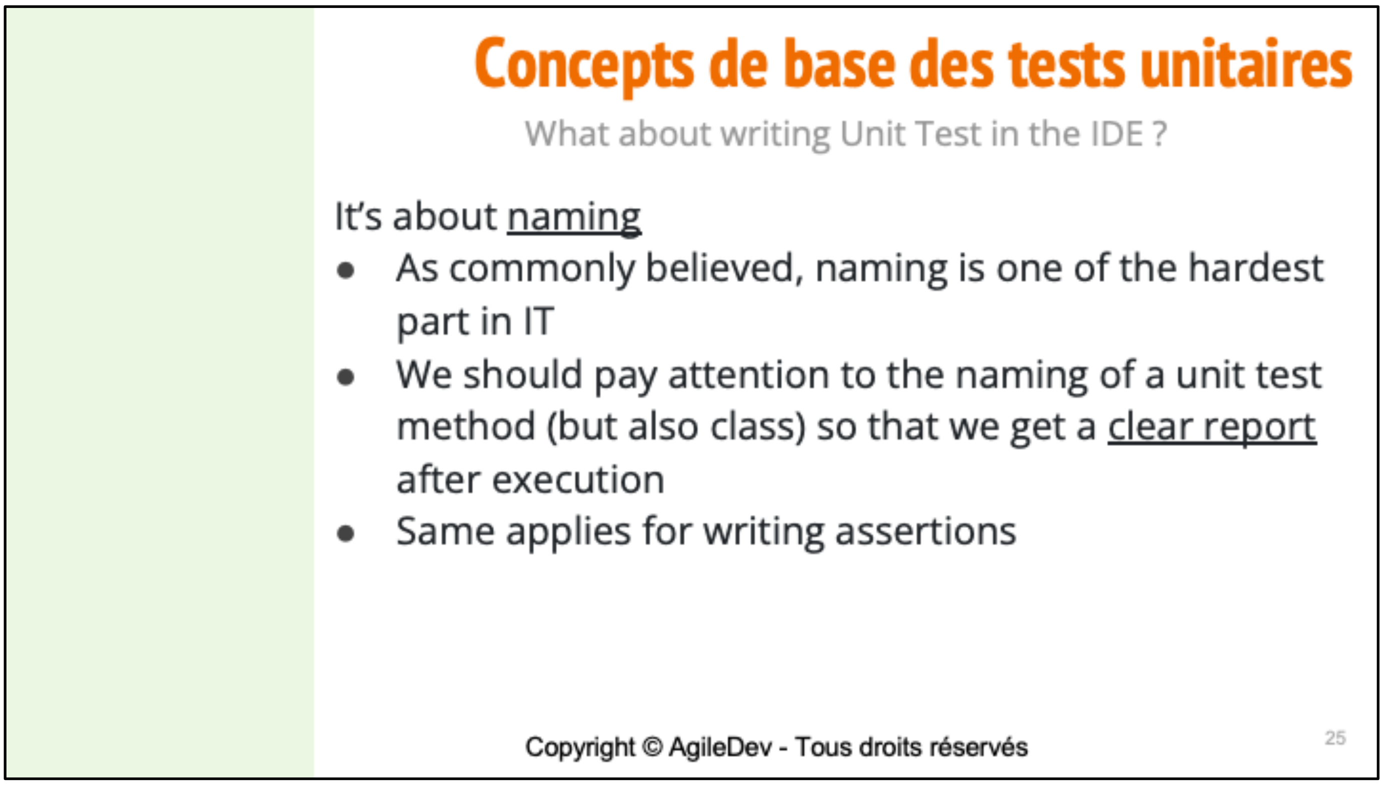


Concepts de base des tests unitaires

What about writing Unit Test in the IDE ?

What about cyclomatic complexity (McCabe, 1976) ?

- defines, through a numeric value, the complexité of a unit of code
- complexity of a method is defined by the number of independant paths we can use to cross the method
- to cover a method with some unit tests, we almost need to write a number of test cases equal to its complexity (and more if some are inaccurate ...)
- it is commonly believed that a piece of code with low complexity is « simple », in theory easier to understand, unit test, and ... refactor



Concepts de base des tests unitaires

What about writing Unit Test in the IDE ?

It’s about naming

- As commonly believed, naming is one of the hardest part in IT
- We should pay attention to the naming of a unit test method (but also class) so that we get a clear report after execution
- Same applies for writing assertions

I kept the hard part for the end …

Concepts de base des tests unitaires

What about writing Unit Test in the IDE ?

It's about naming : naming conventions (camelCase, snake_case) can be differend across langages / teams

```
@Test
public void invalidCouponShouldNotBeValidated() {



}
```

Concepts de base des tests unitaires

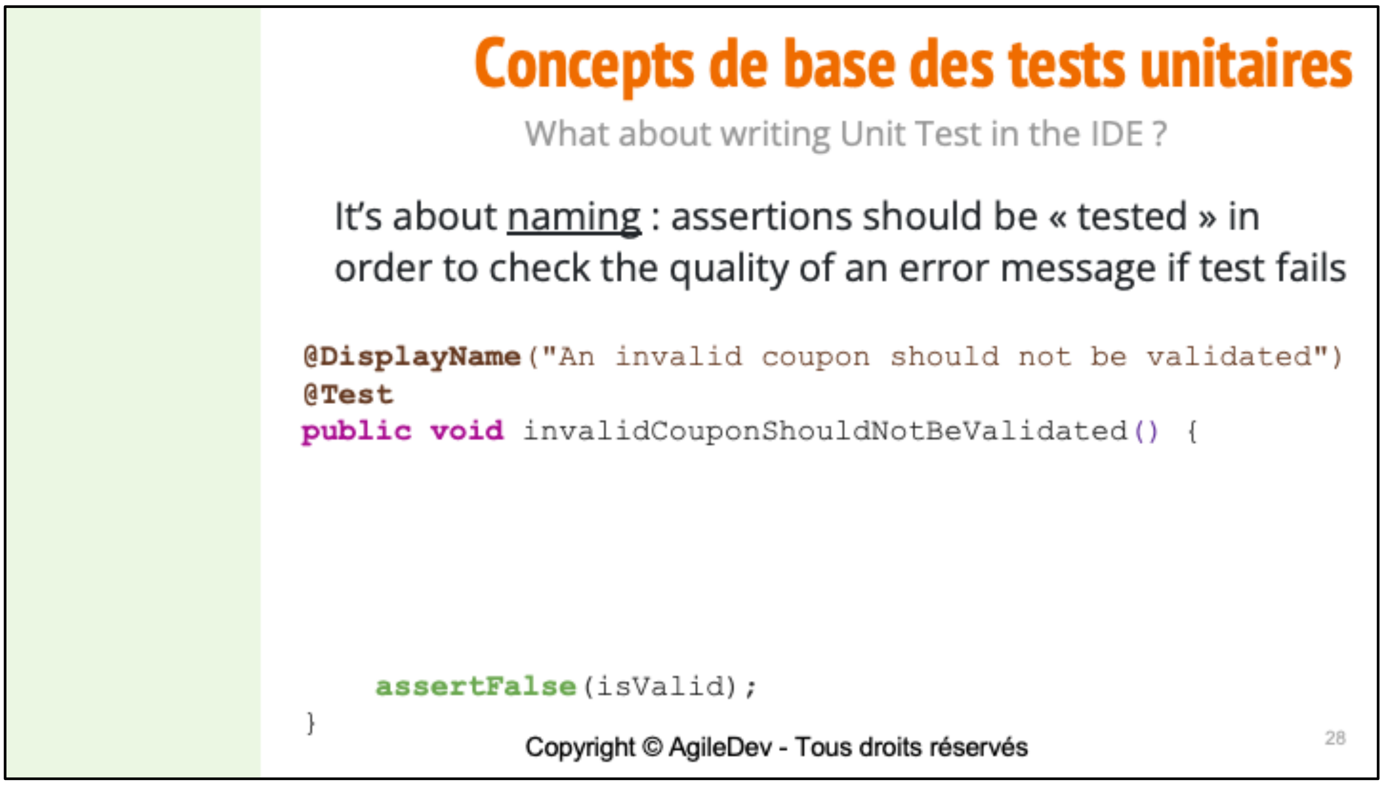
What about writing Unit Test in the IDE ?

Taking opportunities provided by our testing framework (available with both JUnit4 & JUnit5)

```
@DisplayName("An invalid coupon should not be validated")
@Test
public void methodNameIsNoMoreUsedAsTestCaseName() {




}
```



Concepts de base des tests unitaires

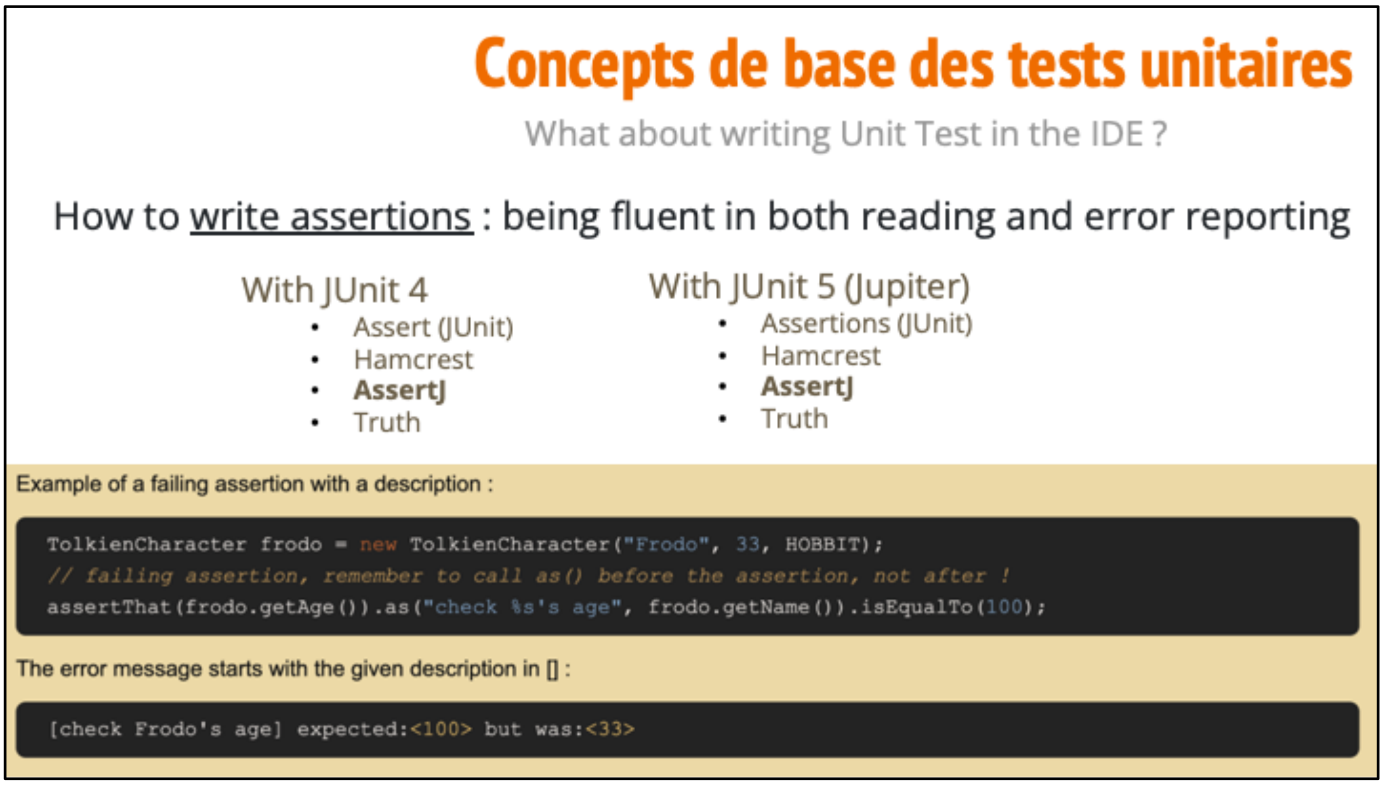
What about writing Unit Test in the IDE ?

It's about naming : assertions should be « tested » in order to check the quality of an error message if test fails

```
@DisplayName("An invalid coupon should not be validated")
@Test
public void invalidCouponShouldNotBeValidated() {




    assertFalse(isValid);
}
```



Concepts de base des tests unitaires

What about writing Unit Test in the IDE ?

How to write assertions : being fluent in both reading and error reporting

With JUnit 4
- Assert (JUnit)
- Hamcrest
- **AssertJ**
- Truth

With JUnit 5 (Jupiter)
- Assertions (JUnit)
- Hamcrest
- **AssertJ**
- Truth

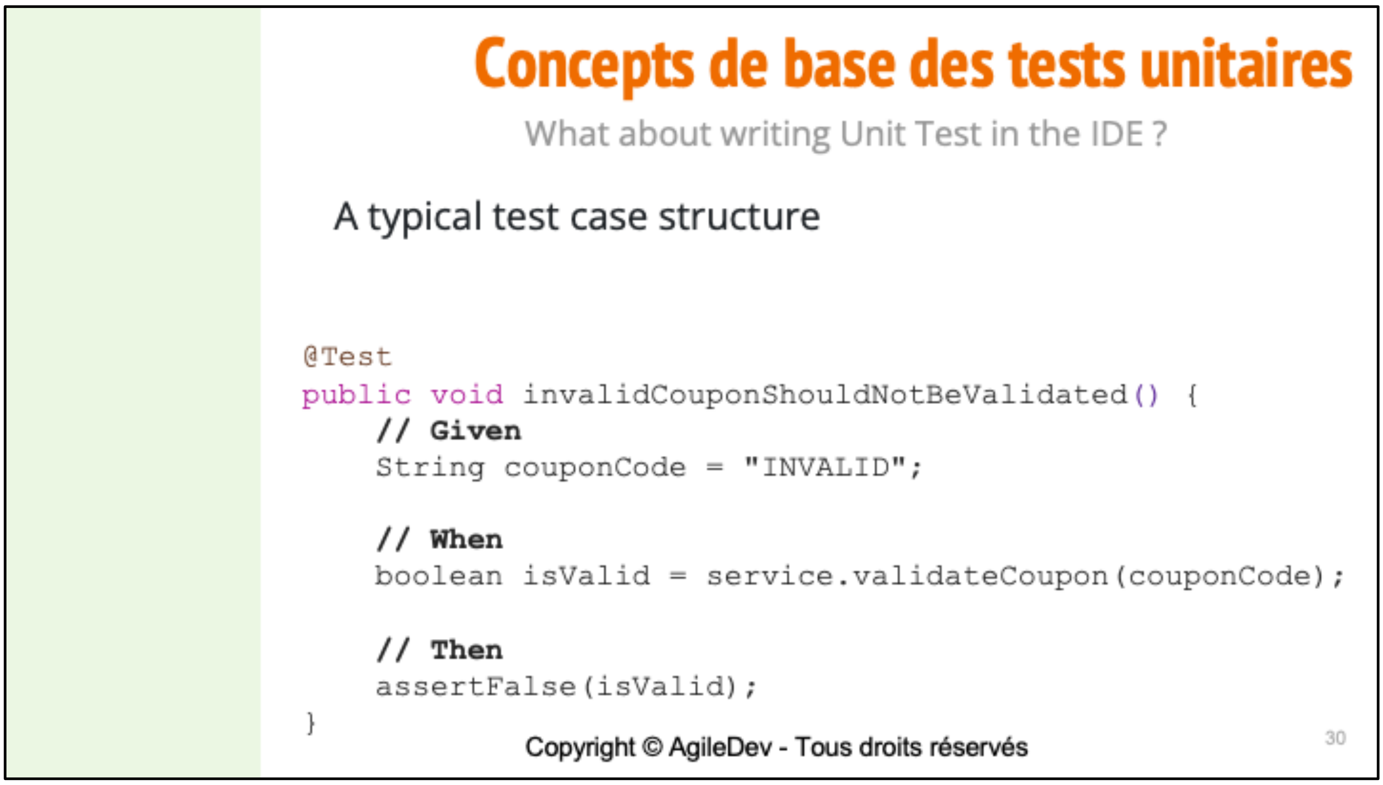
Example of a failing assertion with a description :

```
TolkienCharacter frodo = new TolkienCharacter("Frodo", 33, HOBBIT);
// failing assertion, remember to call as() before the assertion, not after !
assertThat(frodo.getAge()).as("check %s's age", frodo.getName()).isEqualTo(100);
```

The error message starts with the given description in [] :

```
[check Frodo's age] expected:<100> but was:<33>
```

Report built as a website is critical so that non developer people can have an idea of automated testing assets



Concepts de base des tests unitaires

What about writing Unit Test in the IDE ?

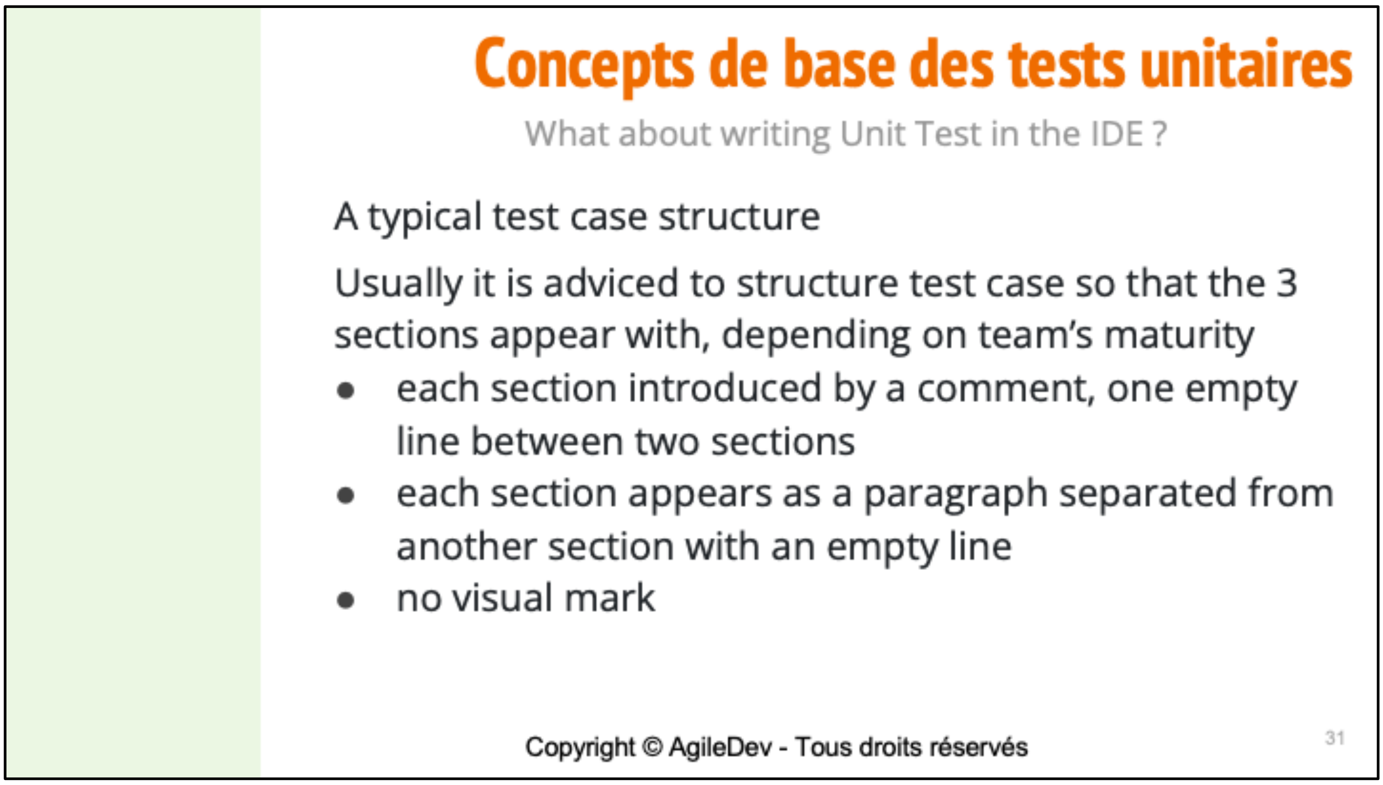
A typical test case structure

```
@Test
public void invalidCouponShouldNotBeValidated() {
    // Given
    String couponCode = "INVALID";

    // When
    boolean isValid = service.validateCoupon(couponCode);

    // Then
    assertFalse(isValid);
}
```

- Given When Then (Black Box)
- Arrange Act Assert (White Box)



Concepts de base des tests unitaires

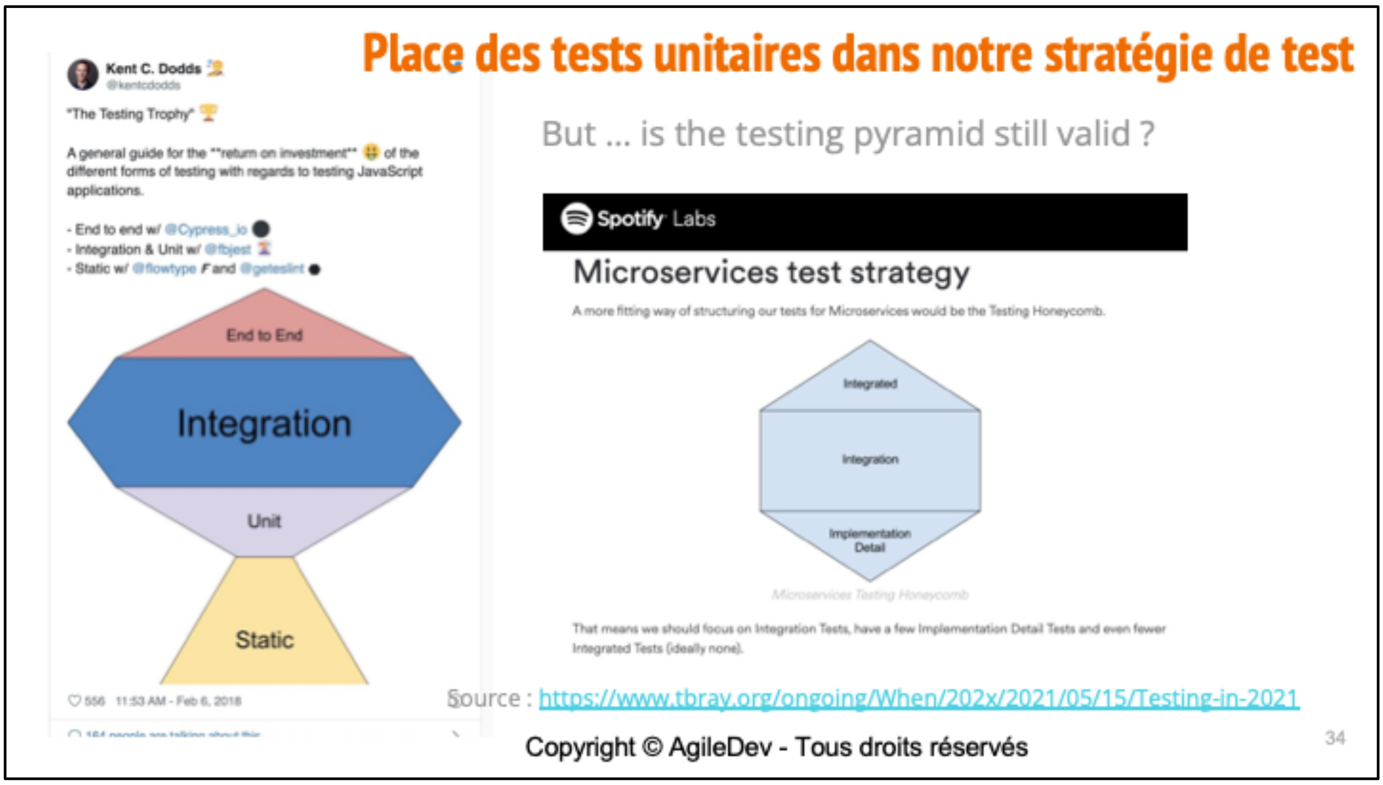
What about writing Unit Test in the IDE ?

A typical test case structure

Usually it is adviced to structure test case so that the 3 sections appear with, depending on team's maturity

- each section introduced by a comment, one empty line between two sections
- each section appears as a paragraph separated from another section with an empty line
- no visual mark

Chapitre 3 – Place des tests unitaires dans notre stratégie de test



Place des tests unitaires dans notre stratégie de test

Kent C. Dodds
@kentcdodds

"The Testing Trophy" 🏆

A general guide for the **return on investment** 🤑 of the different forms of testing with regards to testing JavaScript applications.

- End to end w/ @Cypress_io ⚫
- Integration & Unit w/ @fbjest 🃏
- Static w/ @flowtype 𝙁 and @geteslint ⚫

♡ 556 11:53 AM - Feb 6, 2018

But … is the testing pyramid still valid ?

Spotify Labs

Microservices test strategy

A more fitting way of structuring our tests for Microservices would be the Testing Honeycomb.

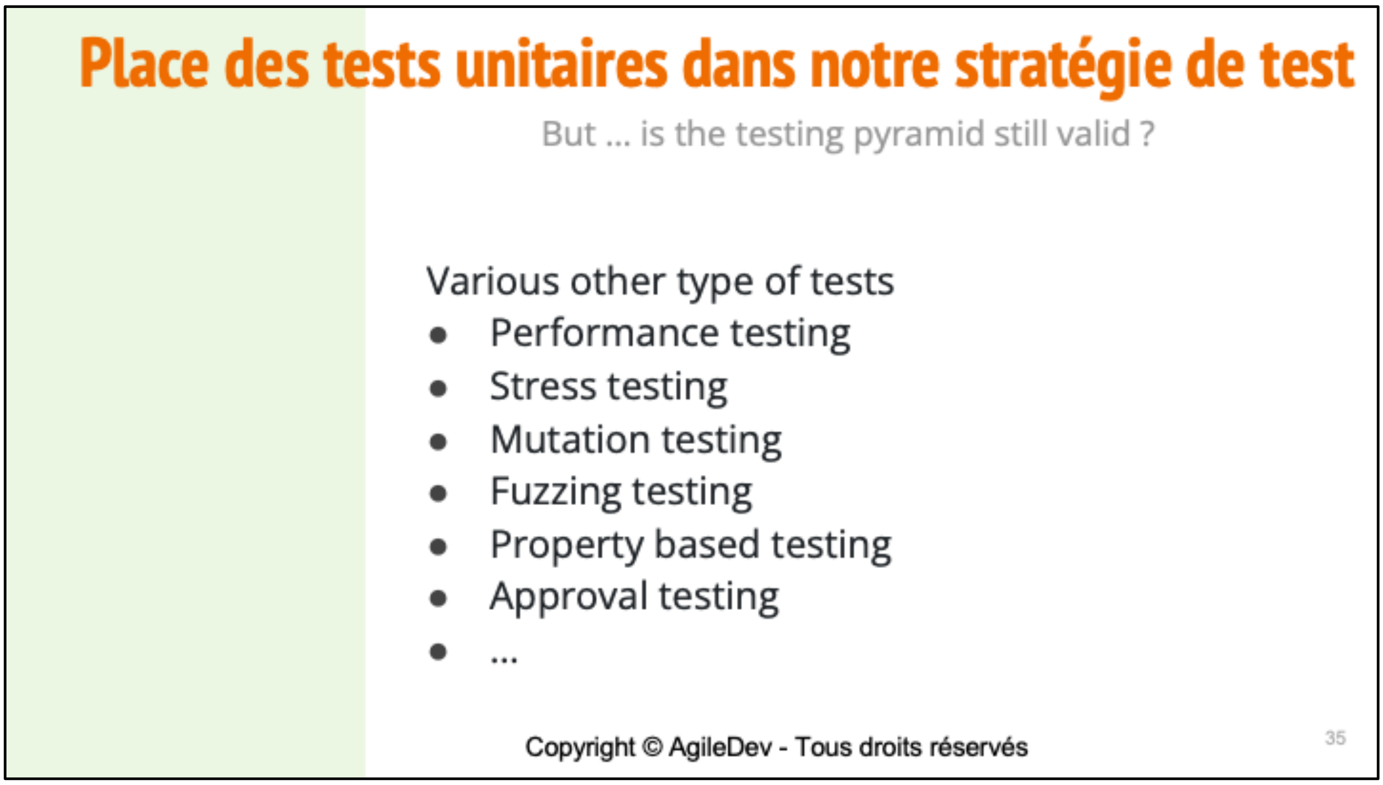
That means we should focus on Integration Tests, have a few Implementation Detail Tests and even fewer Integrated Tests (ideally none).

Source : https://www.tbray.org/ongoing/When/202x/2021/05/15/Testing-in-2021

Source : https://www.tbray.org/ongoing/When/202x/2021/05/15/Testing-in-2021

With emergence of microservices / distributed architectures, some new type of Integration testing appears with a lot more criticity => Contract Testing

Another example : Integration tests have evolved with containerization technologies => TestContainers let us be more efficient writing and running IT and targeting a production equivalent DB (and no more an inmemory one)

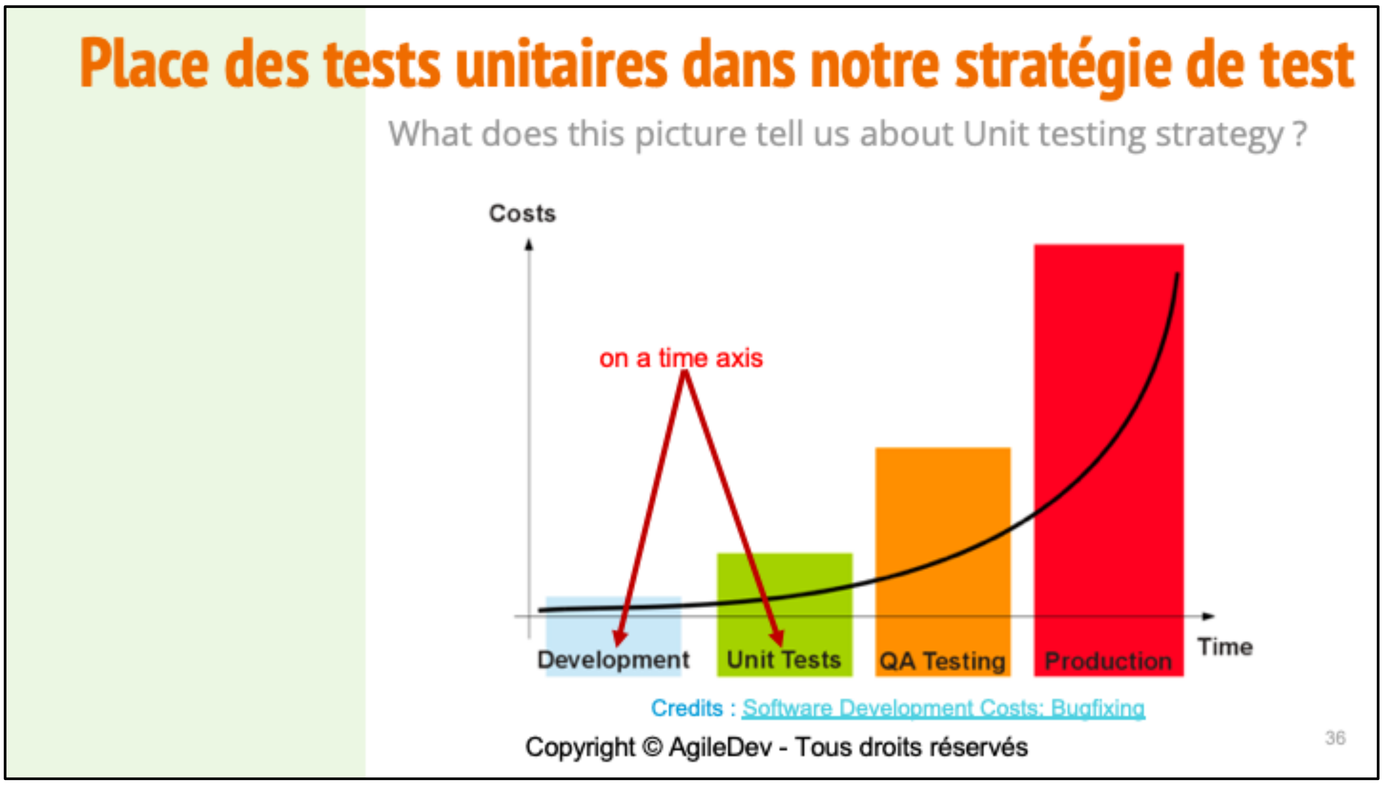


Place des tests unitaires dans notre stratégie de test

But ... is the testing pyramid still valid ?

Various other type of tests
- Performance testing
- Stress testing
- Mutation testing
- Fuzzing testing
- Property based testing
- Approval testing
- ...

Property based testing and Approval testing will help get coverage (though temporary generated tests based on some dataset to provide) on a untested legacy in order to refactor and setup some more accurate unit tests.



Place des tests unitaires dans notre stratégie de test

What does this picture tell us about Unit testing strategy ?

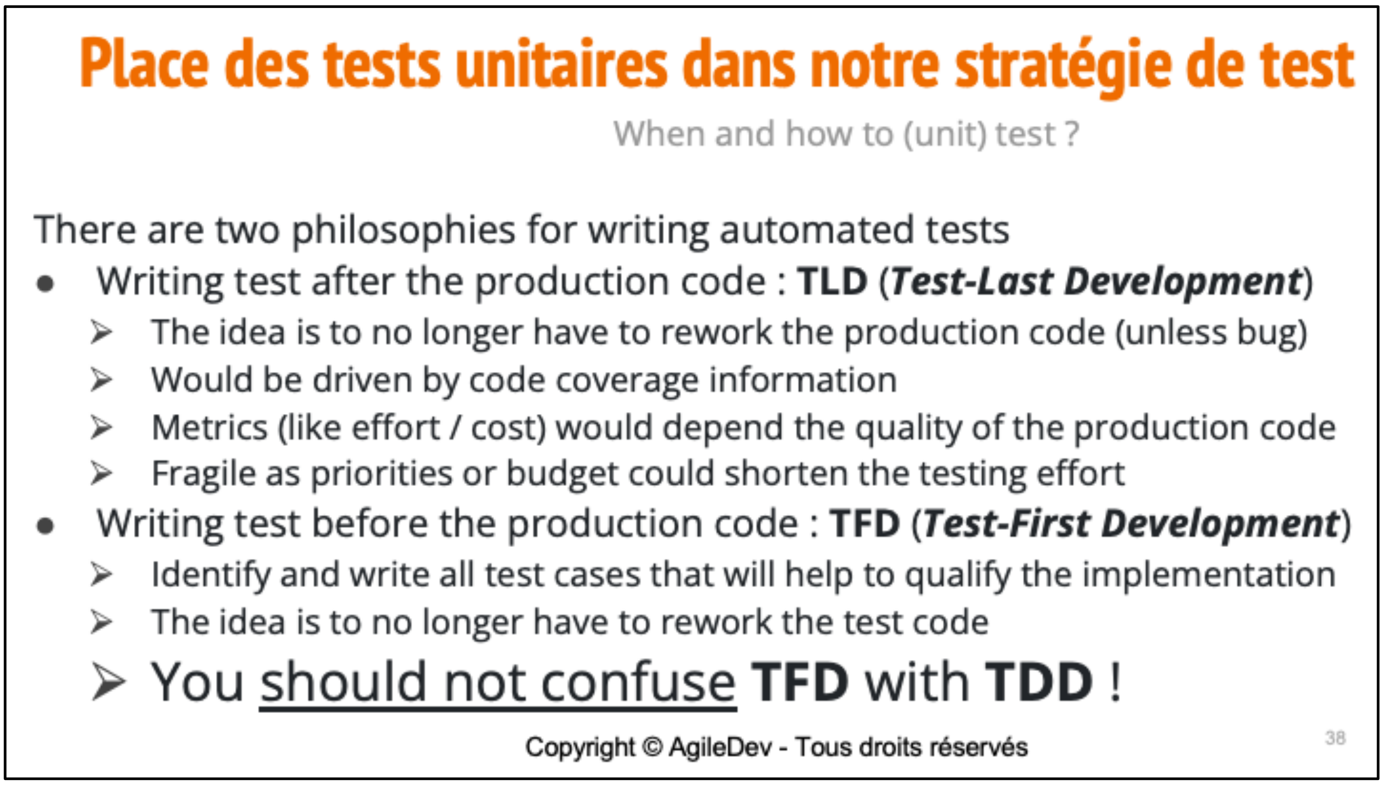
Credits : Software Development Costs: Bugfixing

Credits : https://blog.pdark.de/2012/07/21/software-development-costs-bugfixing/

This would assert that Unit Testing is done « after development », which is probably not what we want, but the curve would by the way become invalid if we mix « Development » and « Unit Tests » in a common rectangle named « Development »

Helps to introduct the Test First vs Test Last / After topic.

Chapitre 4 – Place des tests unitaires dans notre stratégie de développement



Place des tests unitaires dans notre stratégie de test

When and how to (unit) test ?

There are two philosophies for writing automated tests

- Writing test after the production code : **TLD** (***Test-Last Development***)
  - The idea is to no longer have to rework the production code (unless bug)
  - Would be driven by code coverage information
  - Metrics (like effort / cost) would depend the quality of the production code
  - Fragile as priorities or budget could shorten the testing effort
- Writing test before the production code : **TFD** (***Test-First Development***)
  - Identify and write all test cases that will help to qualify the implementation
  - The idea is to no longer have to rework the test code
  - You should not confuse **TFD** with **TDD** !

TLD or test later (maybe … if we have time / budget)

A propos des stratégies de test

What about refactoring ?

« Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.»

- Martin Fowler

A propos des stratégies de test

What about refactoring ?

There are various techniques, purpose being to reach as achievement the emergence of some Design Patterns.

We could summarize refactoring situations in 2 families :

- Using IDE based refactoring tooling
- Relying on some code coverage benefit of some automated test cases