Unit testing to improve the life of software developers and scientists

SÉBASTIEN VALAT – CPPM - MARSEILLES – 24/01/2022

Disclamer

- I see unit testing as a **progress path**
- We **cannot apply** all **in on week**…

Plan

1. A little bit of philosophy & motivation
2. Thinking about testing methods
3. A word on my own experience, feelings
4. Benefits
5. Some words about practice
6. Conclusion

A little bit of philosophy & motivation

How much mistakes costs later .. ?

- **Manhattan** project, **1945**, **Hanford**

- There was a **nuclear reactor**
- For **plutonium** production

- **Takes** water in
- **Cooled** the reactor
- ….and **dump** the water **out**…

https://commons.wikimedia.org/wiki/File:Hanford_N_Reactor_adjusted.jpg

Then there was wastes to handle…

- **Easy** and **quick** and **cheap** solution

- Make a **hole**,
- **Dump** everything in
- **Cover** with sand.

- **Costs** estimation…. ~12 mens,
- An excavator
- A truck

http://www.planetexperts.com/hard-lessons-in-the-us-environmental-protection-agency/

Then there was wastes to handle…

- For **liquids / muds**….
- Solution was to build 177 **tanks**
- **Store** 710,000 m^3

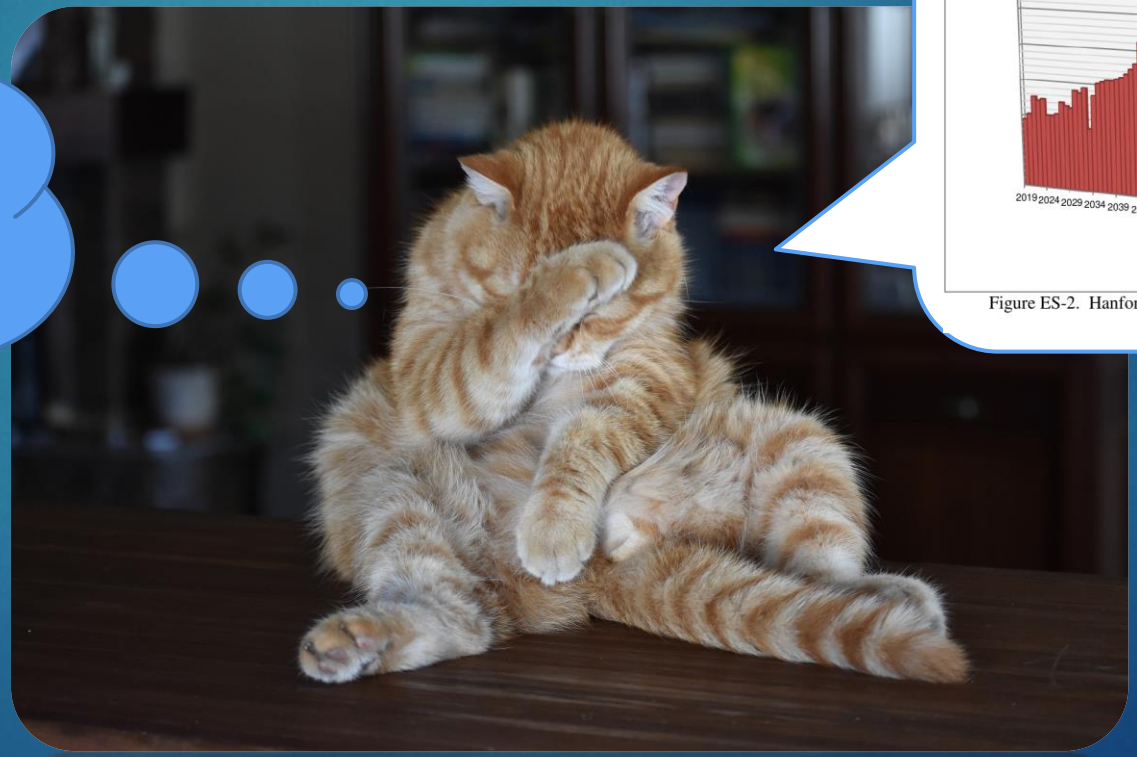
- In the **desert**,
- Dump wastes in
- And **cover with sand**….

- Now, **65 years** later….
- They now (2010) **start to leak**…

https://tlarremore.wordpress.com/2016/02/28/uncontrolled-spread-of-contamination-nuclear-waste-material-hanford-nuclear-reservation-usa/

Toda: that's **technical debt**

Cleanup until 2090
And estimated
~300-600 billions $.



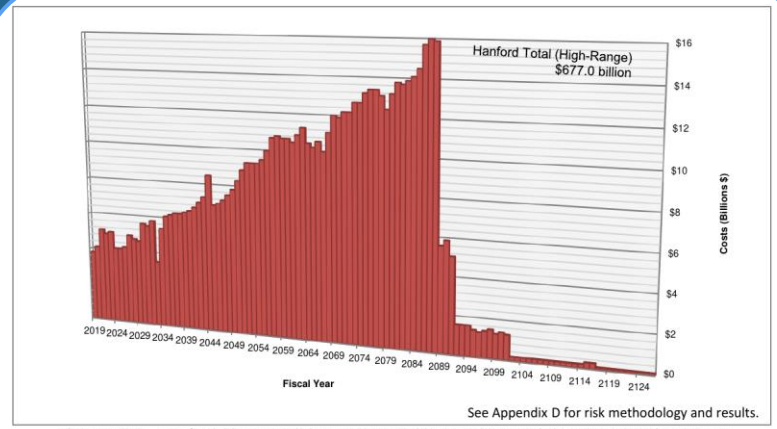


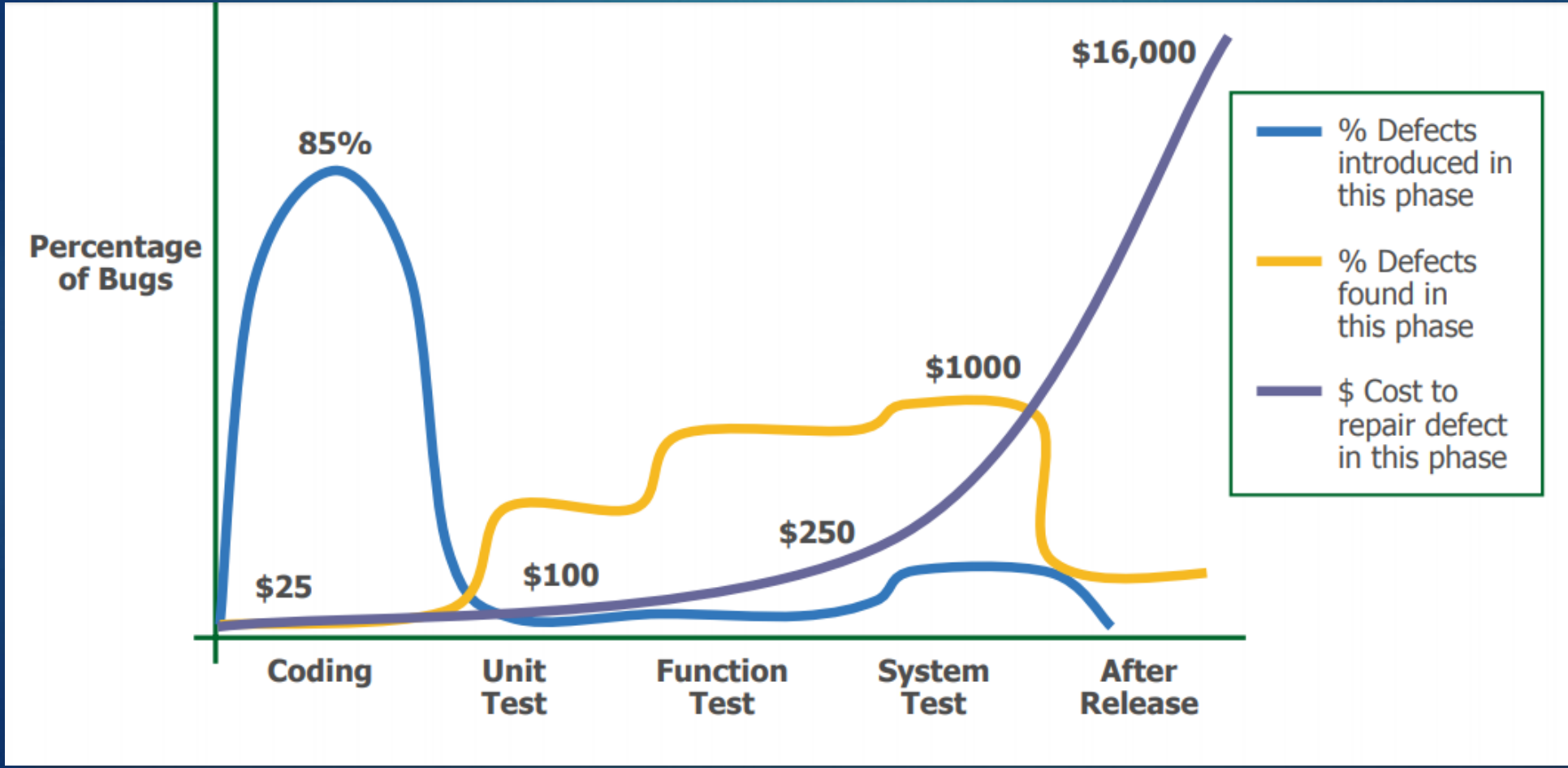
Figure ES-2. Hanford Site Remaining Estimated Cleanup Costs (High-Range) by Fiscal Year (includes both RL and ORP).

https://pixabay.com/photos/cat-redhead-striped-funny-posture-3602557/

https://www.hanford.gov/files.cfm/2019_Hanford_Lifecycle_Report_w-Transmittal_Letter.pdf

Came back to software….

Capers Jones, 1996



Source: Applied Software Measurement, Capers Jones, 1996

2011 - ref

675 compa.
35 gov/mili.
13500 proj.
24 countr.

Thinking about testing

Lets think you are a car engineer

https://commons.wikimedia.org/wiki/File:Manual_synchronized_gearbox.jpg

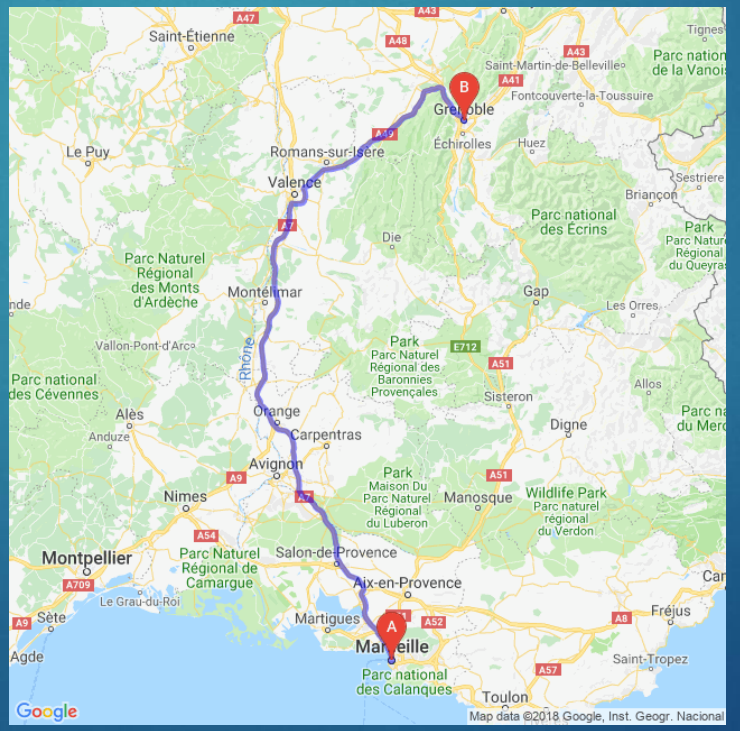
- You work for Renault
- You want to **build a car**
- You are assigned to the **gear box**

You make no test…

- You **designed**, **build**.
- **Sell** the car **directly to customer** and **see**
- **Would you by ?**

Method 1 : manual test

- Way to test a **new gear** we added
- Make a Grenoble – Marseille



Method 2 : **integration** tests

- You **build** a **prototype car** and make **a crash tests**
- **Every time** you **change a gear shape** in the gear box

http://www.thedetroitbureau.com/wp-content/uploads/2016/05/IIHS-Camaro-Crash-Test.jpg

https://www.automobile-propre.com/crash-test-renault-zoe-securite/
http://pngimg.com/download/10020

Method 3 : **unit** test

- You use **a test bed**
- Test **only** the **gear box**
- In **controlled situation**
- Can:
  - put **infrared camera**
  - **Probes** to see temperature.
  - **Vibration measurement**

https://www.techbriefs.com/component/content/article/tb/features/application-briefs/13978

Notice the continuous transition….

- There is **unit test**
  - Test **one gear**
- A little bit more, still unit test
  - Test **two gears**
- …
- A little bit more, **integration** test
  - Test the **gear box**
  - In a **car** on a **test bed.**
- **End to end**, now **test in the car**.

https://www.indiamart.com/proddetail/automotive-spur-gear-19598784273.html
https://en.wikipedia.org/wiki/Spiral_bevel_gear#/media/File:Gear-kegelzahnrad.svg

What is a unit test in python ?


```
from unittest import TestCase

class TestParticle(TestCase):
    def test_move(self):
        # build a particle
        particle = Particle(0)

        # test the initial position
        self.assertEqual(particle.get_x(), 0)

        # move
        particle.move(10)

        # test the final position
        self.assertEqual(particle.get_x(), 10)
```

A bit more advanced one


```
from unittest import TestCase

class TestParticle(TestCase):
    def test_collide(self):
        # build two particles
        particle1 = Particle( 0,5, -1.5)
        particle2 = Particle(-0,5,  1.5)

        # collide particules
        dt = 1.0
        collide = Physics.elastic_collide(particle1, particle2, dt)
        self.assertTrue(collide)

        # checks
        self.assertEqual(particle1.get_vx(), 1.5)
        self.assertEqual(particle2.get_vx(), -1.5)
```

Most unit test frameworks relies on:
assert keywords,
sometimes also **expect**

Run example - OK


```
sebv@sebv6:~/2022-01-unit-test$ pytest Particle.py
========================================= test session starts ==========================================
platform linux -- Python 3.9.7, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/sebv/2022-01-unit-test
collected 2 items

Particle.py ..                                                                                  [100%]

========================================== 2 passed in 0.02s ===========================================
```

Run example - failure


```
sebv@sebv6:~/2022-01-unit-test$ pytest Particle.py
========================================= test session starts ==========================================
platform linux -- Python 3.9.7, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: /home/sebv/2022-01-unit-test
collected 2 items

Particle.py F.                                                                              [100%]

=============================================== FAILURES ===============================================
_____________________________________ TestParticle.test_collide ______________________________________

self = <Particle.TestParticle testMethod=test_collide>

    def test_collide(self):
        particle1 = Particle(-0.5, 1.5)
        particle2 = Particle( 0.5, -1.5)
        collide = Physics.elastic_collide(particle1, particle2, 1)
        self.assertTrue(collide)
>       self.assertEqual(-1.5, particle1.get_vx())
E    AssertionError: -1.5 != -3.0

Particle.py:31: AssertionError
======================================= short test summary info ========================================
FAILED Particle.py::TestParticle::test_collide - AssertionError: -1.5 != -3.0
===================================== 1 failed, 1 passed in 0.08s ======================================
```

A realistic case


```
      Start 28: TestWorker
28/34 Test #28: TestWorker ......................   Passed    0.11 sec
      Start 29: TestWorkerManager
29/34 Test #29: TestWorkerManager ...............   Passed    0.16 sec
      Start 30: TestTaskIO
30/34 Test #30: TestTaskIO ......................   Passed    0.11 sec
      Start 31: TestTaskScheduler
31/34 Test #31: TestTaskScheduler ...............   Passed    0.11 sec
      Start 32: TestIORanges
32/34 Test #32: TestIORanges ....................   Passed    0.11 sec
      Start 33: TestTaskRunner
33/34 Test #33: TestTaskRunner ..................   Passed    0.18 sec
      Start 34: TestClientServer
34/34 Test #34: TestClientServer ................   Passed    1.85 sec

100% tests passed, 0 tests failed out of 34

Total Test time (real) =   9.42 sec
sebv@sebv6:~/Projects/iocatcher/build$
exit
```

TDD: Test Driven Development

- The **strict** approach:
  - You **write the tests**
  - **Implement** until it valides the tests
- More **realistic** approach:
  - Implement a **function / class**
  - Implement the **tests**
  - **Iterate** to **improve** the **implementation** and **API**

Some word on my own experience, feelings

When trying to push in teams…. [integration]

- **Integration test**
  - Mostly **everybody agree**
  - Not exactly on the way to do it….
  - Seems easier at first look
  - Most of the time it **starts** with **dirty bash scripts**.

- **Quickly cost a lot**
  - Eg. CEA MPC project, **10 000** MPI tests, …. **5000 fails**…
  - **One week** to run everything
  - **Depressing**
  - Harder to debug
  - **Nobody looked** on results except me and another one

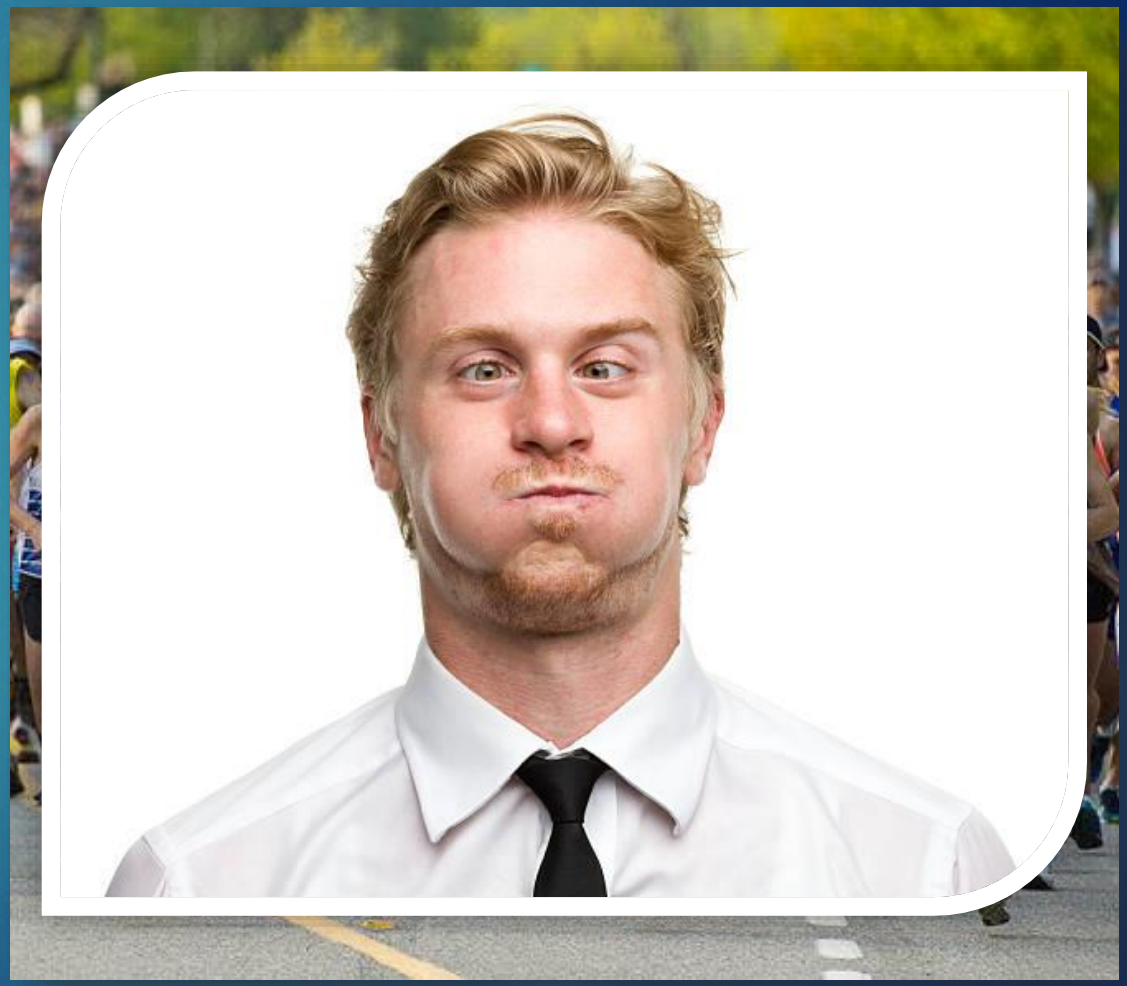
When trying to push in teams…. [unit tests]

- **Unit tests**
  - **Required an investment**
  - We are slower to start
  - **Hard to introduce in pre-existing software**
  - **Lots of gain on long term**

- **Common first kill** :
  - "This one is **too hard** to test" ✖
  - "This one **call many others**" ✖
  - "I'm sure of this function, it is **so simple**" ✖

First time I made unit tests

- I was **not convinced**
  - But **I tried**
- Had the impression to **loose my time**
- It **was hard**
- I **didn't see the benefits**
- **I already had most of my codes**
  - Painfull to unit test for weeks



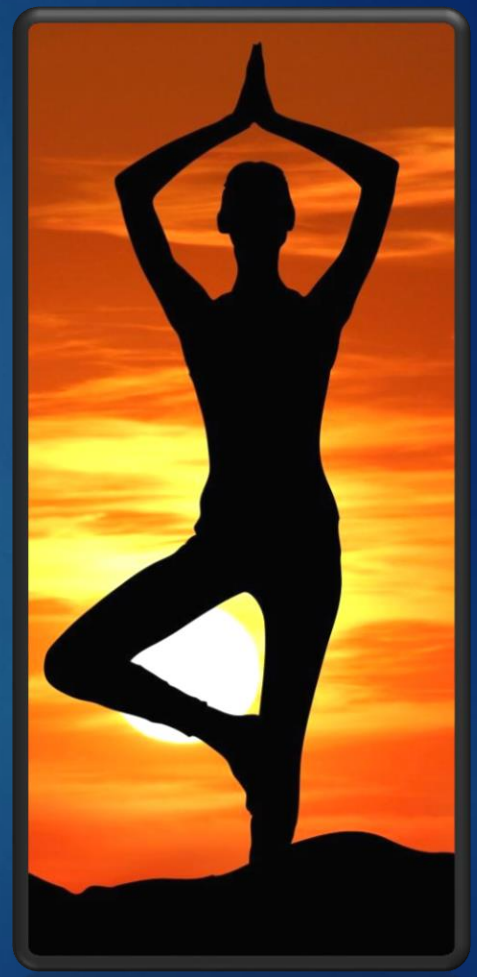
That's also adequate tools and ways to work

Day to day methodology : **discipline**

- **"This is a POC…. I will make my tests later"** ✖

- **You will never do them later**
  - Because your ***design*** *will* ***not permit***
  - Because you will **want to move** to **other stuff**
  - **Nobody** will be **happy** to write unit **tests for ~4-8 weeks**
  - **Your boss/commercial manager already sold it to clients**….

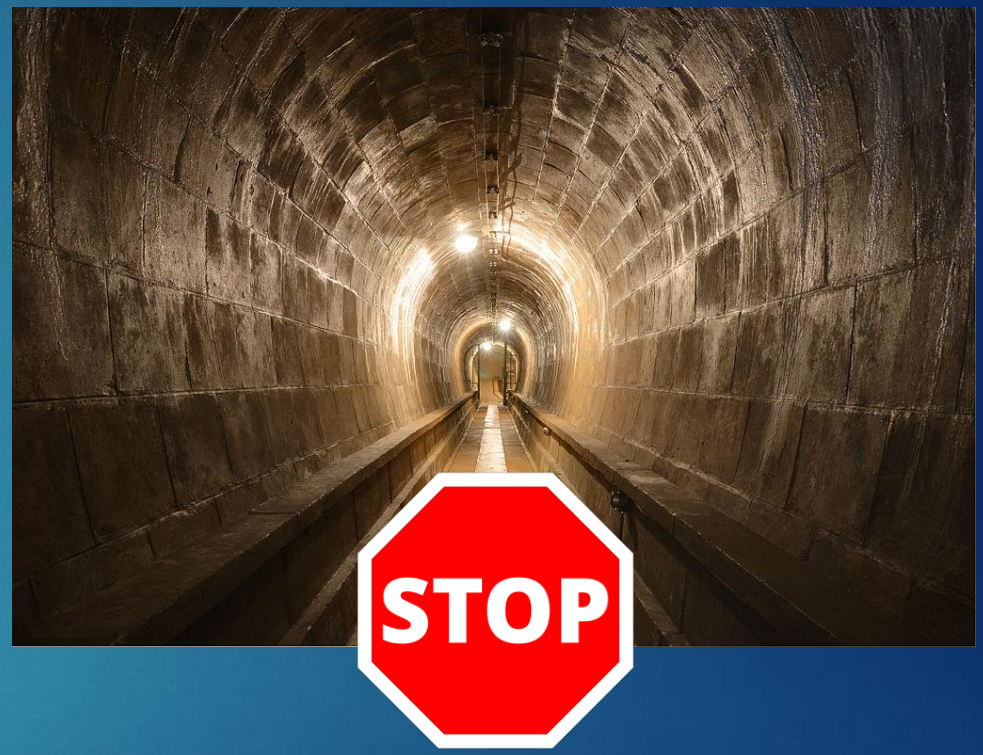
- You already **loosed half the benefits** of unit tests
  - **Become** a **more or less useless cost**

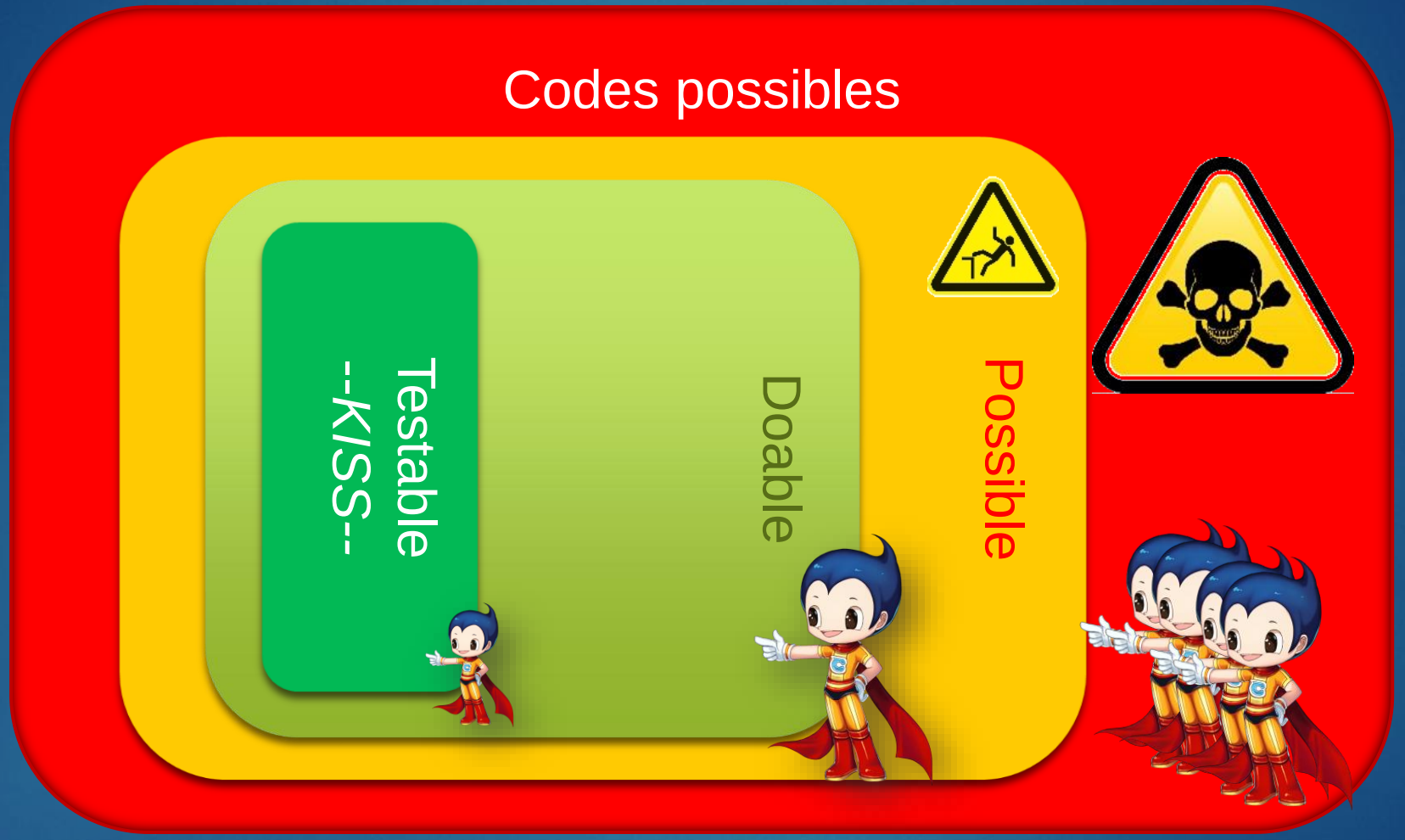


Benefits

Benefits of unit test

- **That's not only testing**
- Develop **outside the production** env.
- It forces you to **think your internal design**
- Is a **spec**, also for **internal APIs**
- Open easy door for **refactoring / rewriting**
- **New developers** are more confident (**you in 1 year** or **your interns**…)

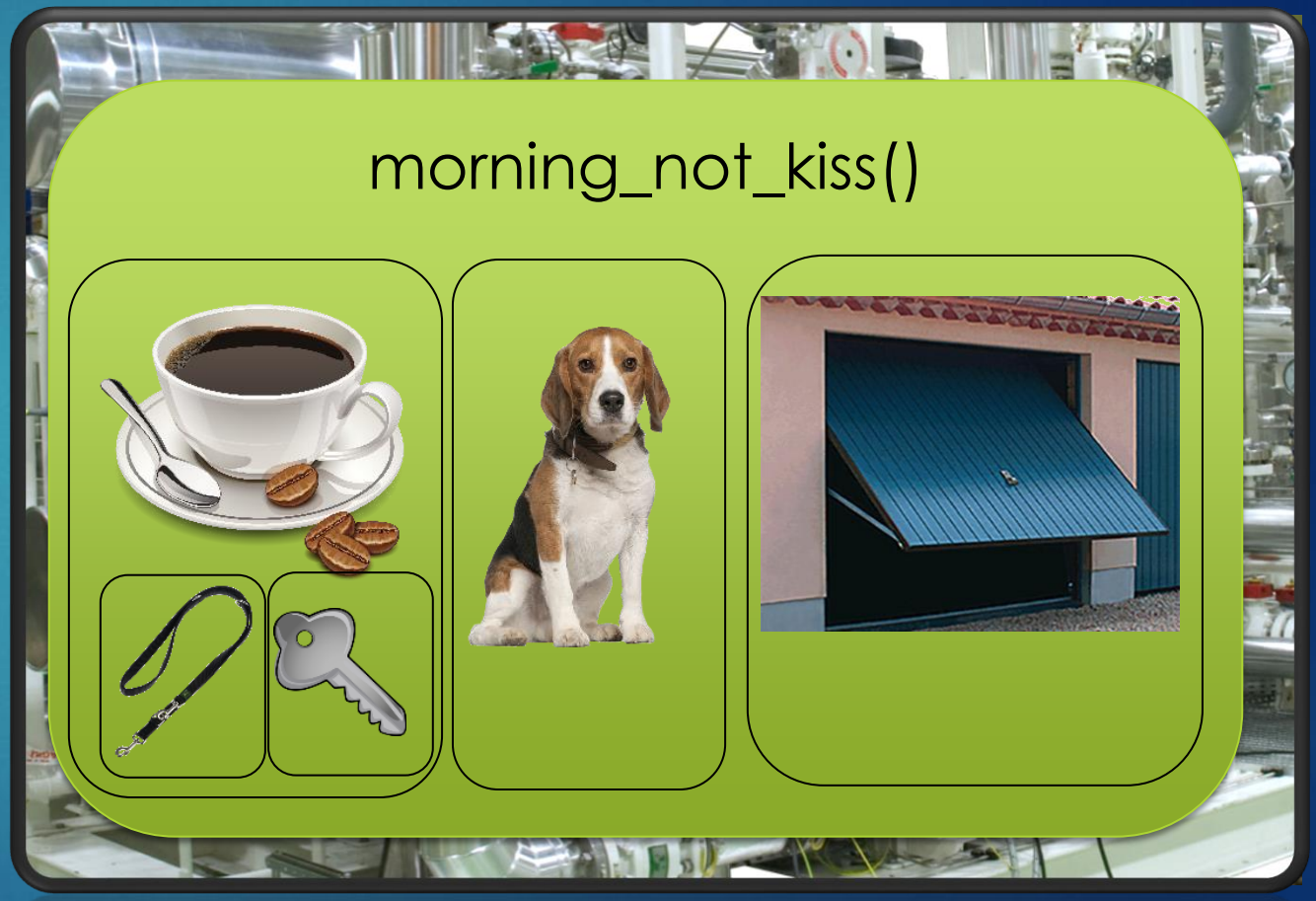




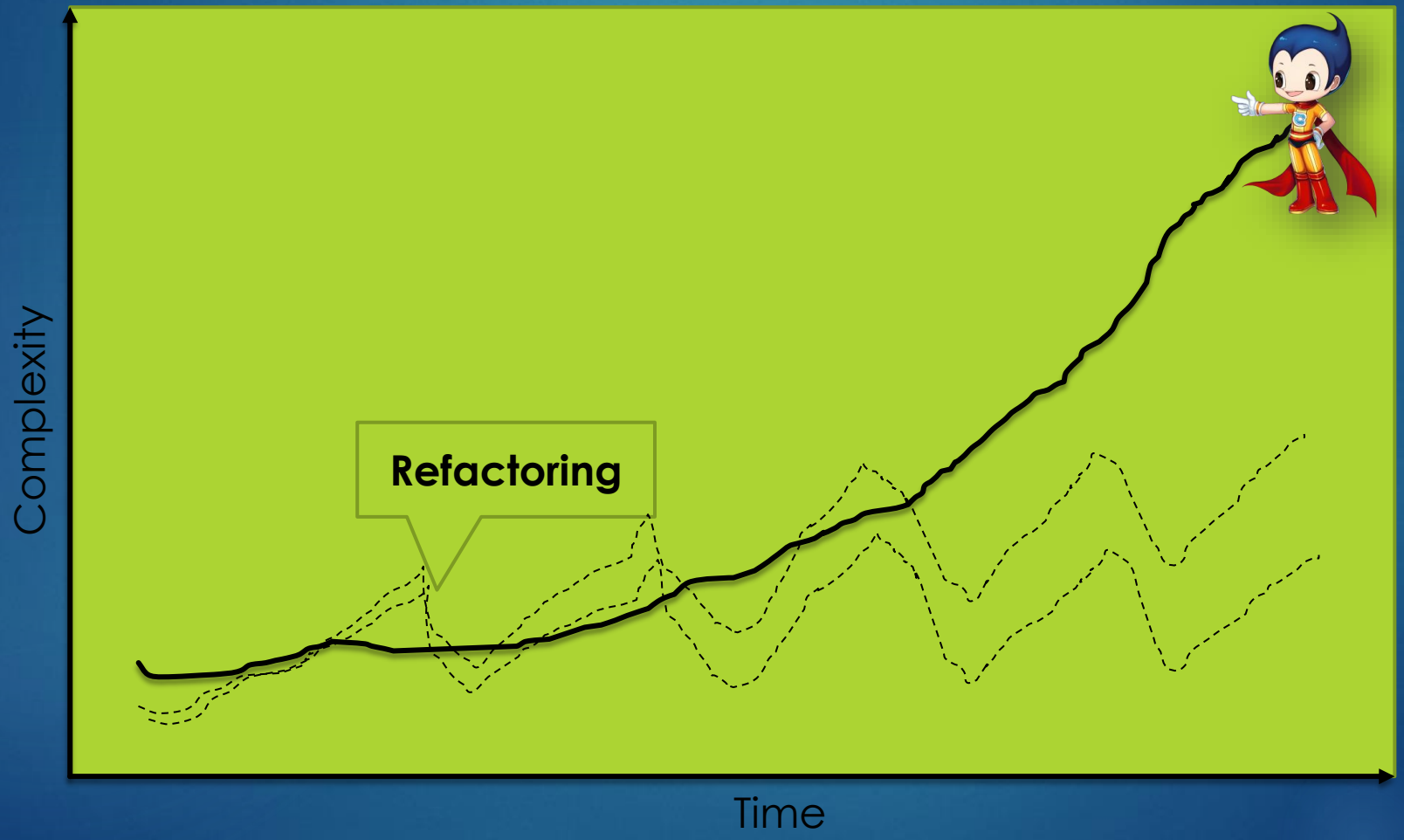
https://pixabay.com/fr/illustrations/codage-heroes-personnages-2771135/

Test a gas machine

- If your **test** become **too complex**
- You are **certainly** on the **wrong way**
- **Stop**, **think** and **KISS**



Keeping control on complexity



It is a valuable knowledge

- The **knowledge** of the **internal concepts** and **API**
  - Unit tests **document it**
  - Unit tests validate your patches **do not break it**
- Knowledge of **corner cases encountered**
  - Tests **keep track of them**
- Can be very usefull to **save your algorithms in case**:
  - **Rewriting** a **new version** from scratch
  - **Translating** to **another language**

About performance

- Having a unit test means:
- You **have a short example** to use **each part** of your code
- You can **extract it** in a simple file
- Make advanced **performance measurement**
- On **each distinct part** of the code

My feeling now

- **Applied** during **all my PhD. + Post-docs**
  - For the last **12 years** on **200 000 lines** of codes (C++/Python/Go/Rust/JS)
  - **Very usefull** and **pleasant**
  - Permited to **quicly reatched** **very interseting results**

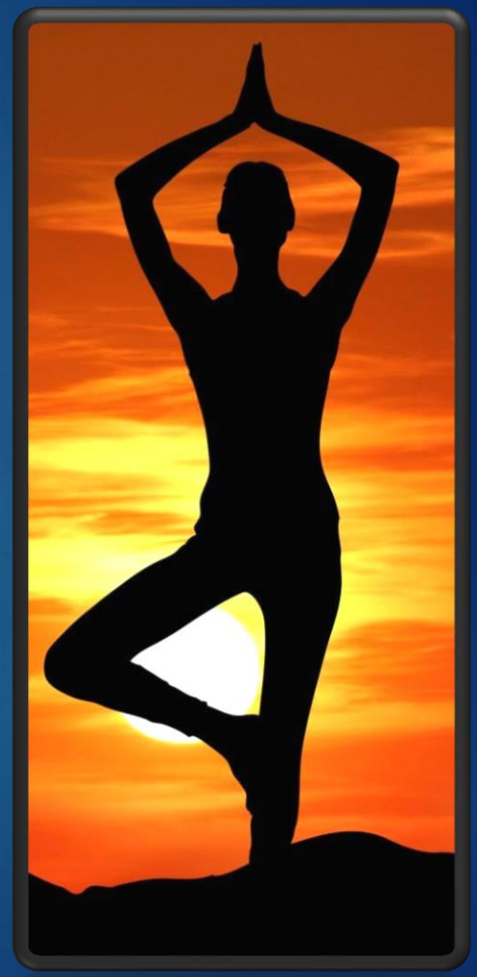
Some words about practice

My time rules

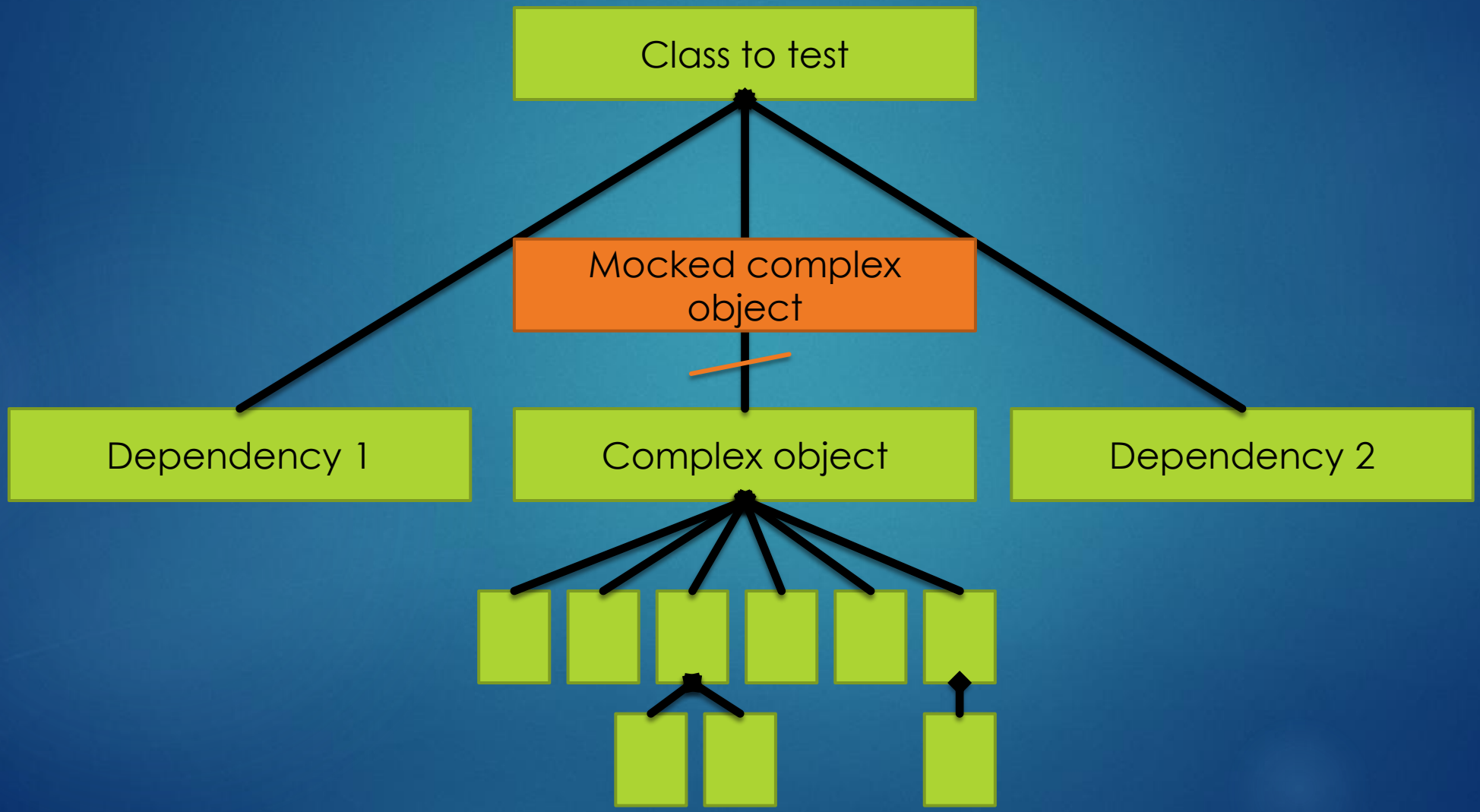
- Of course **depend on** language / objectives / complexity
- **2 weeks** of coding
  - Ok without unit test
- Start **for longer**
  - Immediate unit test
- Extend **after 2 weeks**
  - Take 1-2 weeks to **refactor + unit test**
  - **Before continuing** progressing
- For **~1 year project**
  - **Up to 1st month** loosed with “slower” progress
  - Largely compensated afterward

Unit tests should stay simple

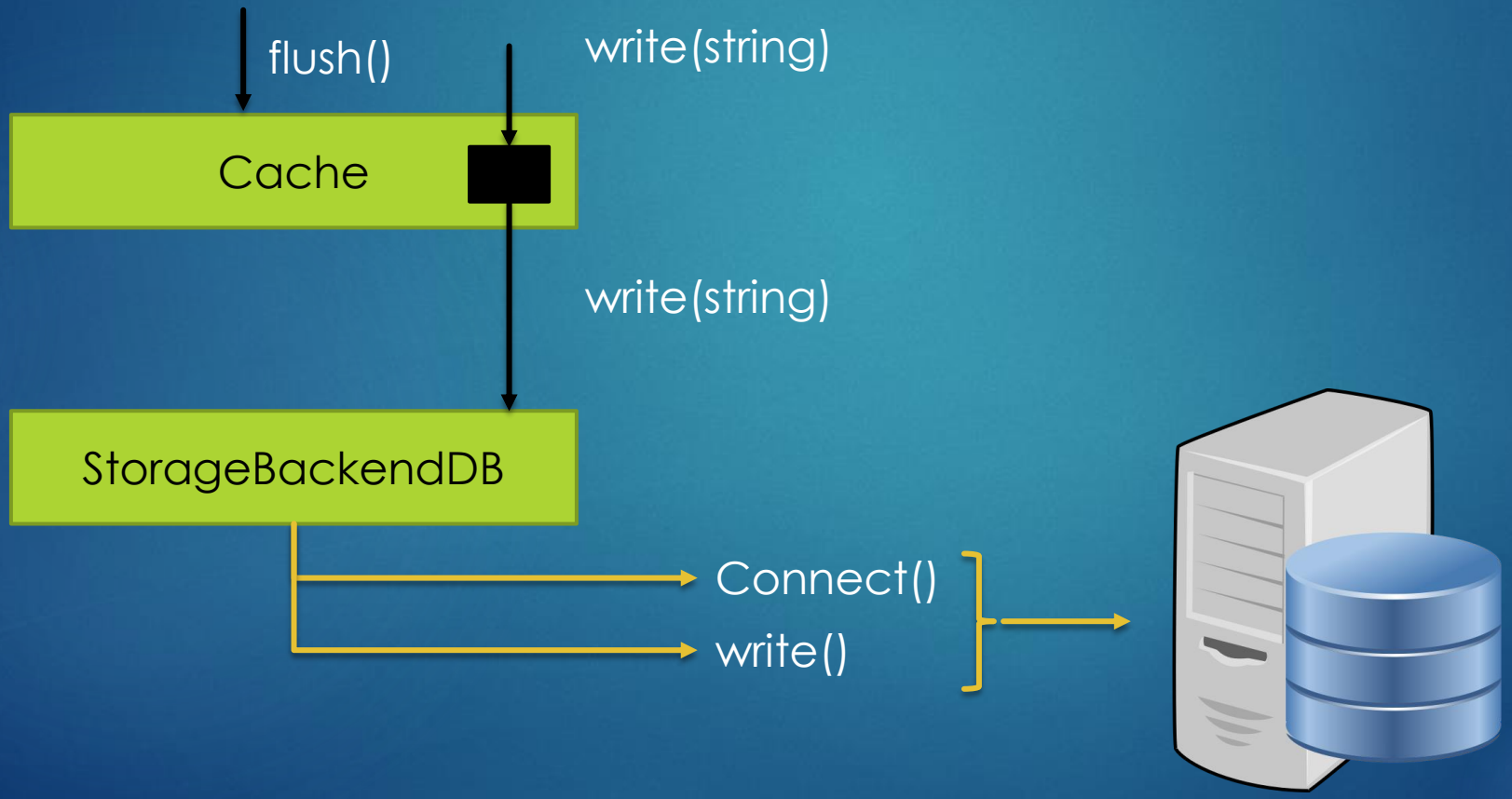

```
TEST(TestProject, loadContent_fail_minimum_required)
{
    FileLines content;
    content.push_back("[cdeps_minimum_required 2000.4.3]");

    SpecFile file;
    file.loadContent(content,"none.none");

    Options options;
    Project project(&options);
    EXPECT_EXIT(project.loadSpec(file),
       ::testing::ExitedWithCode(1, "version is too old");
}
```



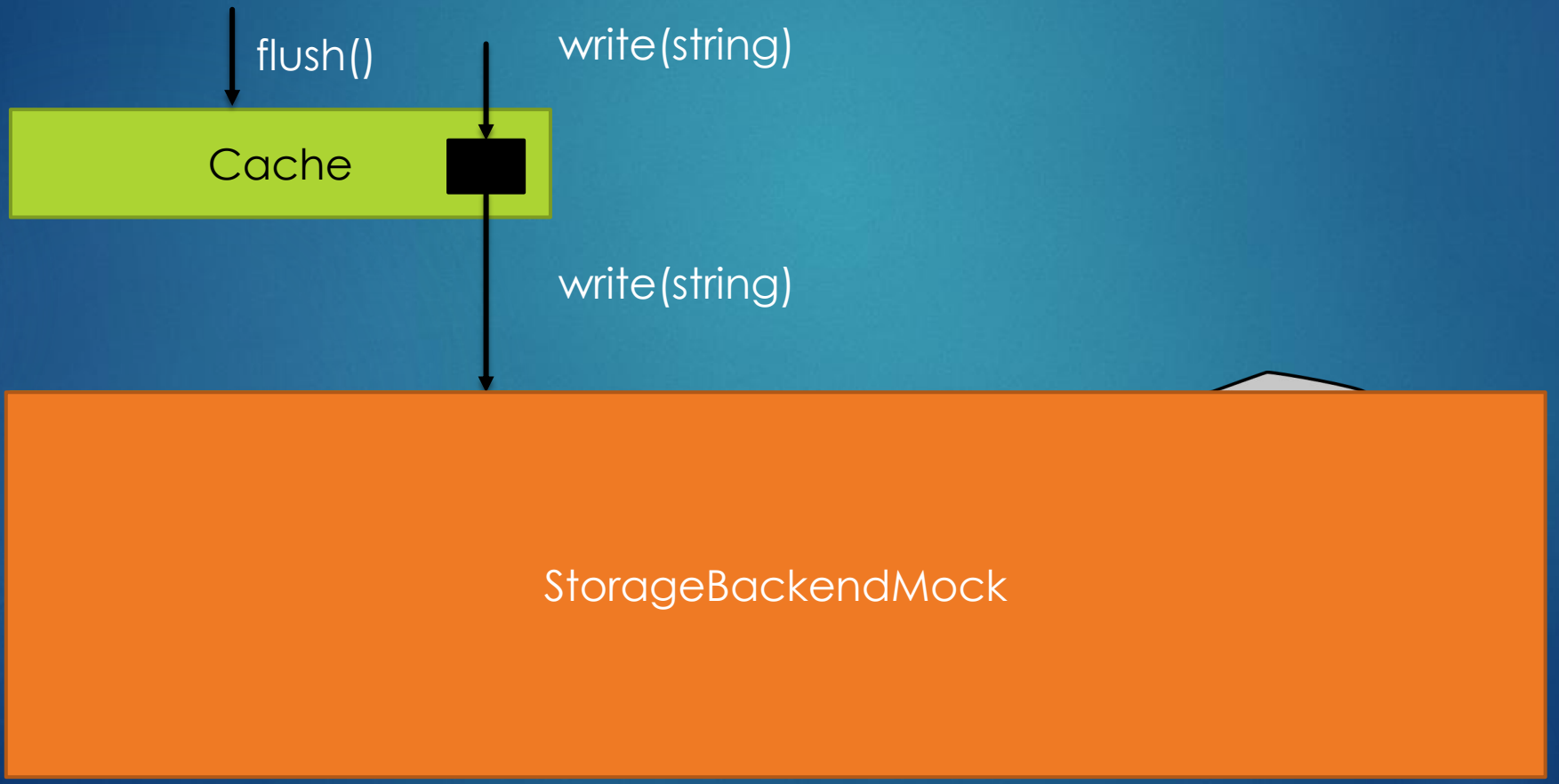
About mocking



A small example



A small example



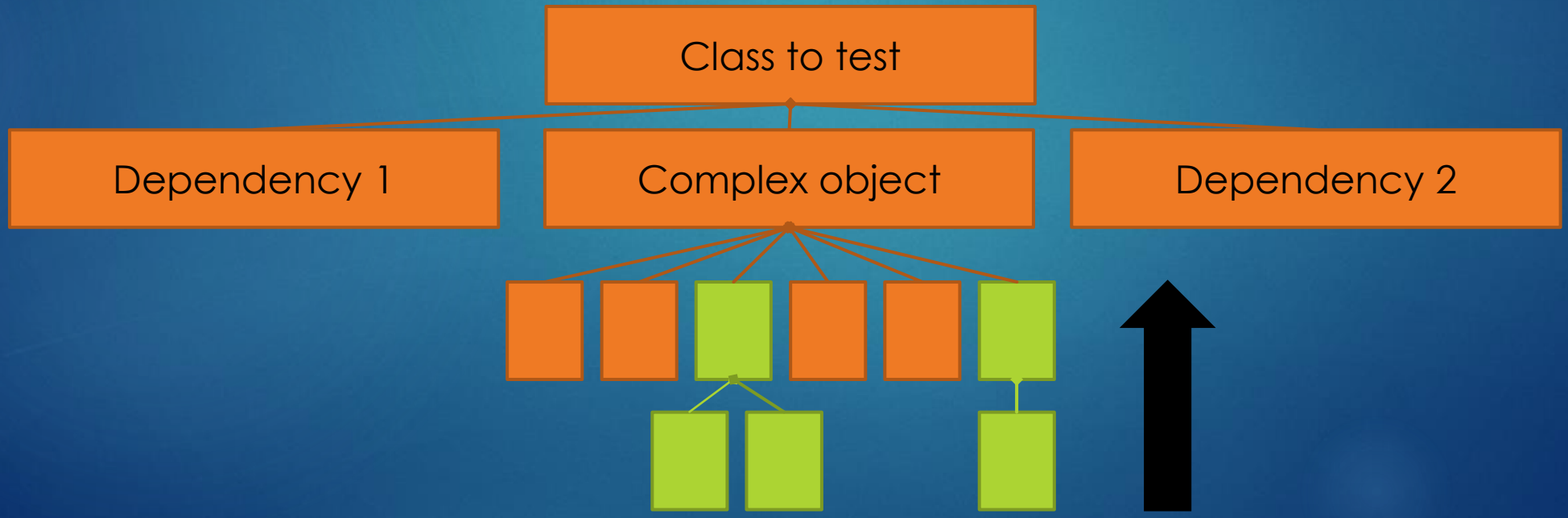
About mocking with a framework


```
import unittest
import mock

class TestObject:
    def test_flush(self):
        # build my hierarchy & mock the write function
        mocked_backend = DummyBackend()
        mocked_backend.write = mock.MagickMock(return_value=11)

        # create the cache
        cache = Cache(mocked_backend)

        # write & flush
        cache.write("string data")
        cache.flush()

        # check MockBackend.write was called
        mocked_backend.assert_called_with("string data")
```

Start in an existing software

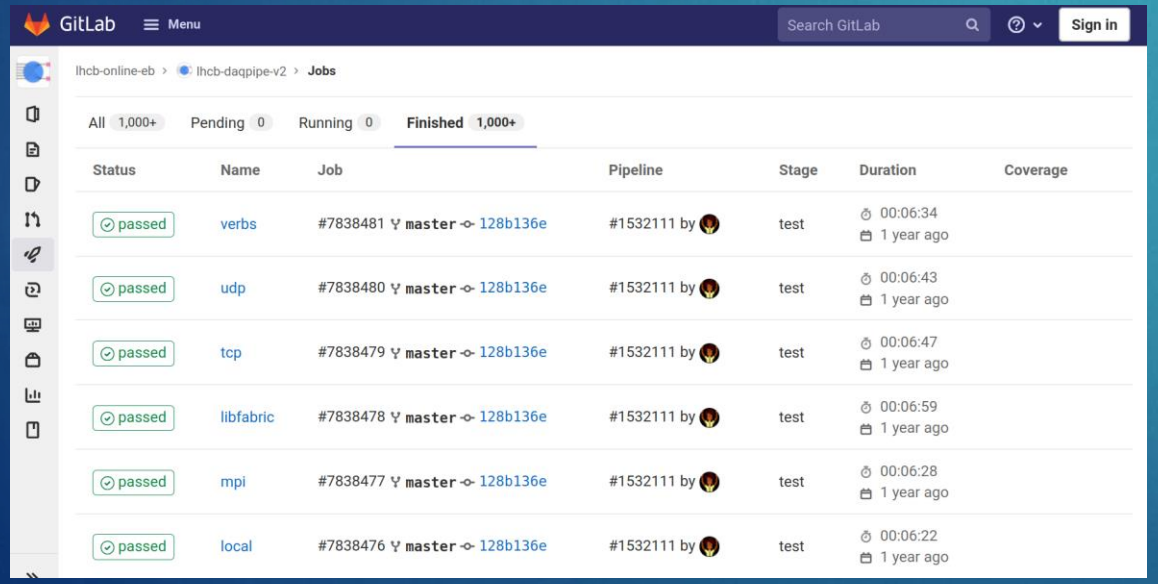
- This **will be hard** at **to cover all**

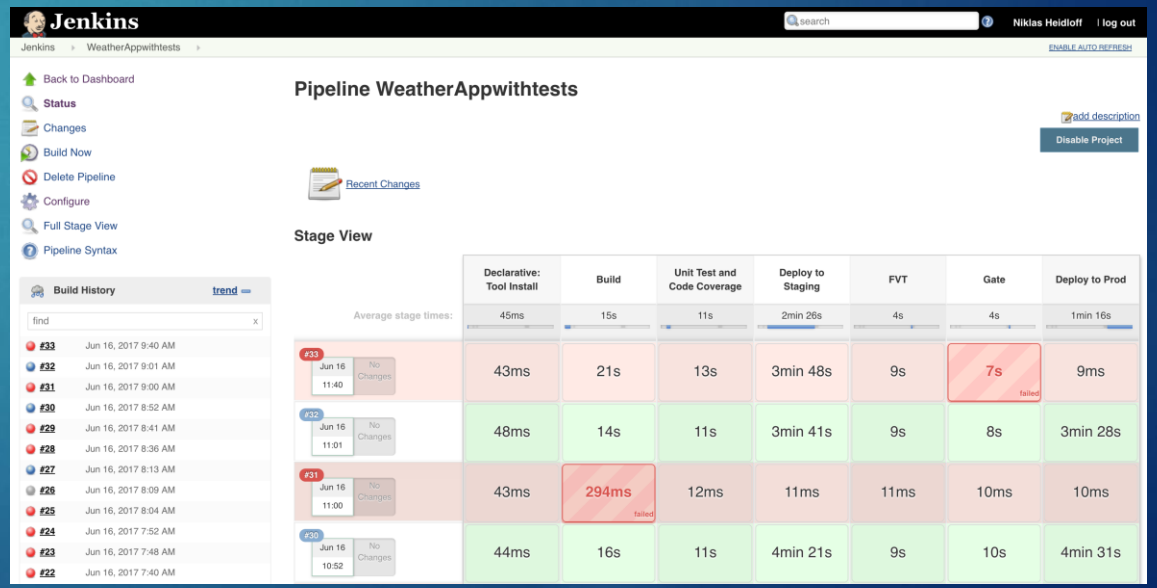
- Start **from the leaf** class / functions **up to the top**
  - It will require **refactoring** to make the middle elements testable



Test automation

- Automatically run the tests
  - when you push your code on your repo
- For example via **Gitlab-CI** or **Jenkins**





One remark about test automation

- This is **relatively easy** for **unit tests**
- But for **integration tests**: can be **time consuming**
  - If have a **large number of integration tests**
  - If the tests are **complex to run**
- Mostly an issue **when** the **test environment change**
  - Eg: in a team I was in, **2 PY** (Person Year) **consumed to move**

One word on coverage

- **Not required** to stricly **cover every cases** at start
- The point is: **at least one test** for every components
- To **ensure it is testable**

Some framework

Language	Test framework	Mocking
Python	unittest	unitest.mock
C++	Google test Catch2 Boost test library cppunit …	Google mock FakeIT
C	Google test Criterion	
Bash	bats	
Rust	[native]	*mockall*
Go	[native]	*gomock*

Conclusion

Conclusion

- **Integration tests** looks **easier at start**
  - That's **absolutely wrong** on **long term**
  - I'm **not saying** you should **not do any**
- Units testing **costs at the beginning** but **quickly win on long term**
- **Even more true** to target **performance**
  - Code splitted in **boxes**
  - Allow **refactoring**

Conclusion

- For **research** you always want to **test news algorithms**
- Your also do **not want to loose time in debugging**
- **Start KISS** to get a clean **working code, then complexify**
  - Agile methods
- **Be able** to **quickly refactor**
  - Agile methods

Conclusion

- The test is like a **teacher**
  - If it **fails** => **you need to fix**
  - If it is **too complex** => **wrong way**
- Look it as a **progress path**
- **With time** you will **learn** how to **split** your code to be **testable**
  - The good patterns are **domain specific**

THANKS

BACKUP

PhD. memory allocator

- Implement a **parallel allocator**
- To run on **Bull BCS** & **Tera100**
  - **16** processors (**NUMA**) => **128 cores**
  - **Target** application : Hera (**~1M line of C++**)
- **~1 year** of base **development** with unit test
- On my **workstation**

PhD. memory allocator

- Just got the malloc/free functions
- Pass all the tests
- Inject in all my KDE session
  - Just one bug
- First run in the application
  - OK
- Work on performance
  - After two weeks => 2X gain on the app.

A safety for QA guy

- Quality loss and **rush warnings**.
- Noticed **not via a human channel** through the **quality exigent guy** !

In agile methods

- This is a component for **agile method**
- You plane for **short term**
- Opposite to **long term planning of V model**
- Thanks to unit tests, **for a new feature**:
  - You **refactor** to prepare
  - You **implement** the new feature

Refactor with :
Only Integration tests ?
No tests ?