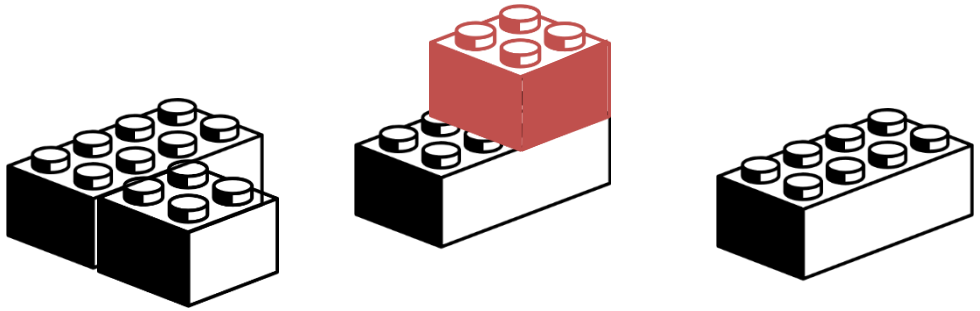
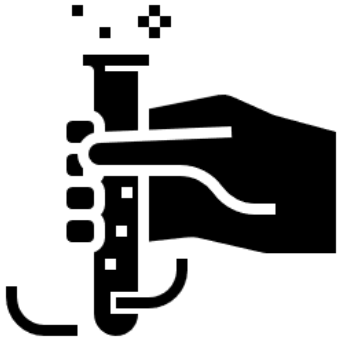
Collaboration et Robustesse

Mise en œuvre par le langage Python

J. Saraydaryan

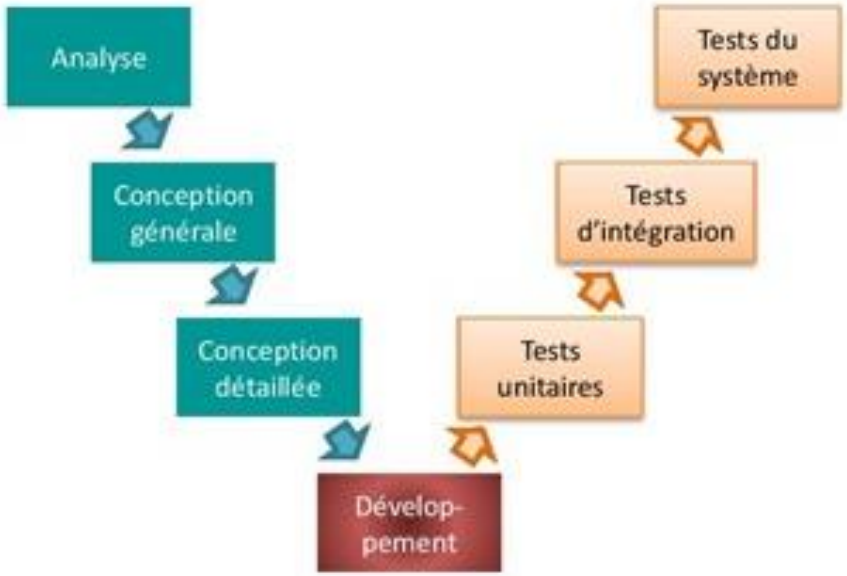




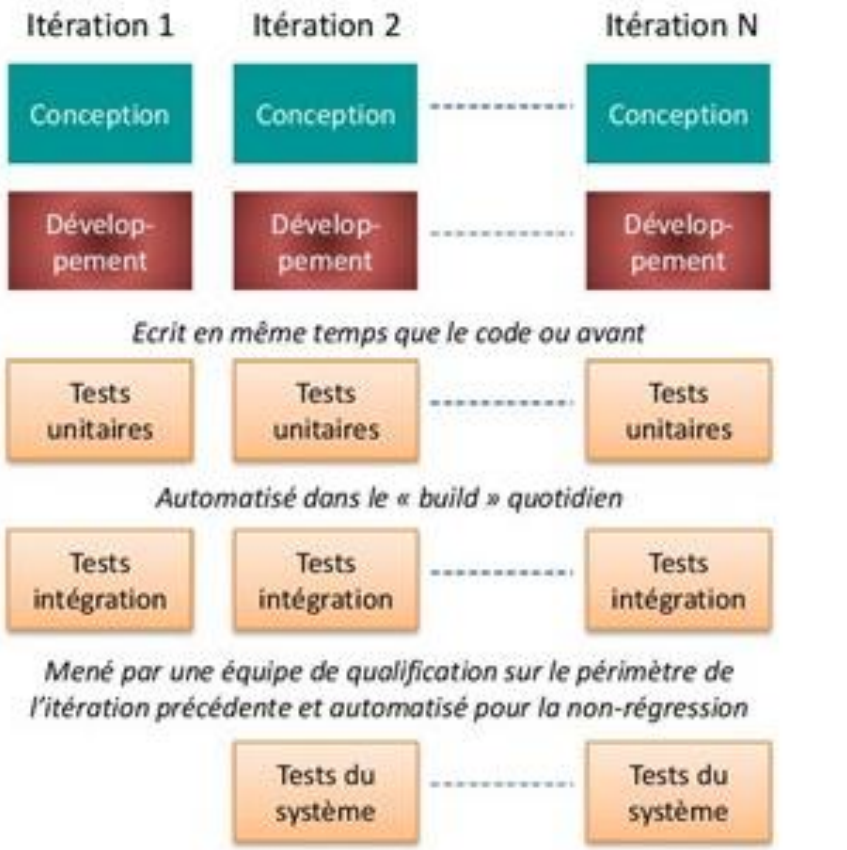
TESTs !

Cycle en V

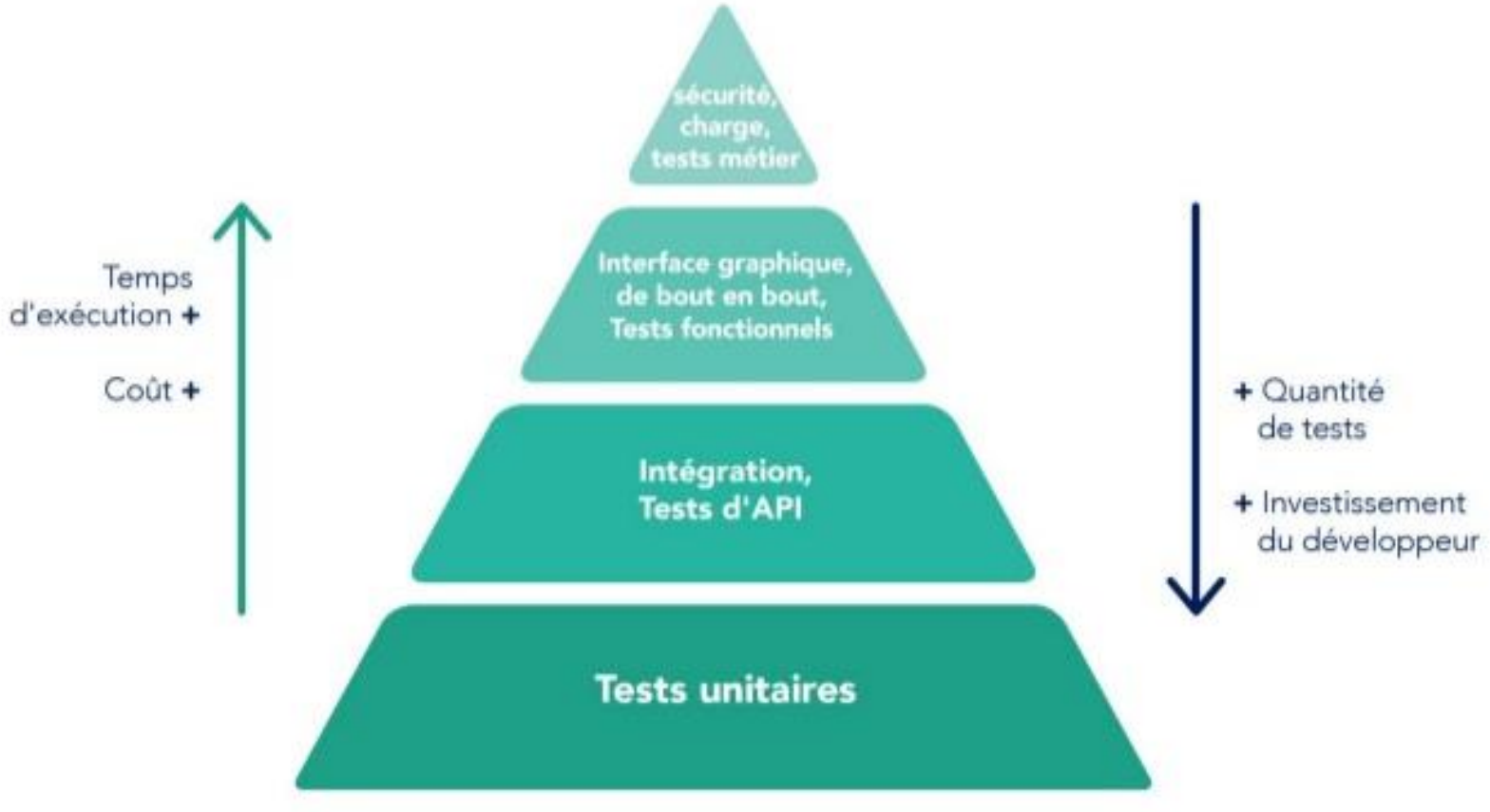
Calendrier global du projet sur une longue période



Méthode Agile



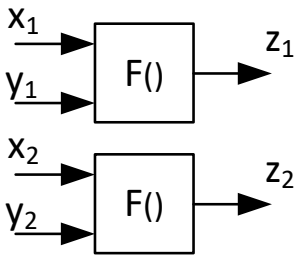
https://www.softfluent.fr/blog/societe/Les-meilleures-pratiques-du-test-logiciel



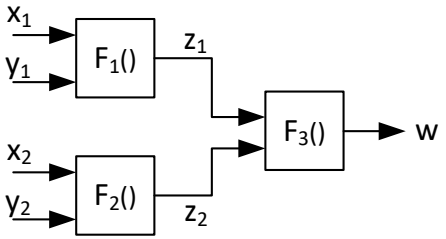
https://www.slideshare.net/DamienBeaufils1/lightning-talk-meetup-swc-pyramide-des-tests

Type de Tests

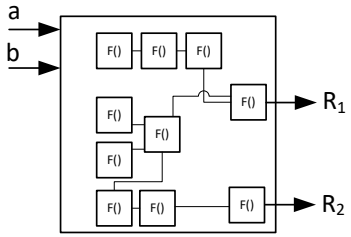
- **Tests Unitaires :**
  - **Définition:** Vérification du bon fonctionnement d'une unité de programme
  - **Objectif:** vérifier le contrat de service lié à l'unité de programme



- **Tests d'intégration :**
  - **Définition:** vérifier le bon fonctionnement d'un ensemble d'unité de programme
  - **Objectif:** s'assurer que le comportement d'un ensemble d'unité de programme respecte bien le comportement attendu.



- **Tests Systèmes :**
  - **Définition:** vérifier le comportement d'un système complet de bout en bout
  - **Objectif:** s'assurer que le comportement du système complet de bout en bout est conforme aux spécifications et aux exigences



Type de Tests: Exemple

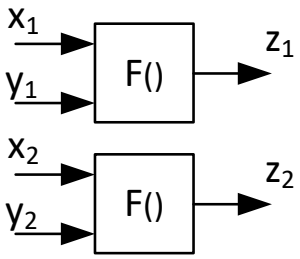
« Application de gestion de cartes: Objectif créer des cartes de jeux et évaluer automatiquement la valeur de chaque carte:
1 Carte est composée:

- Titre String sans caractère spéciaux
- id Integer
- hd compris entre 1-100
- attack compris entre 1-50
- defense compris entre 1-50
- energy compris entre 1-100
- price compris calculer en fonction des caractéristiques

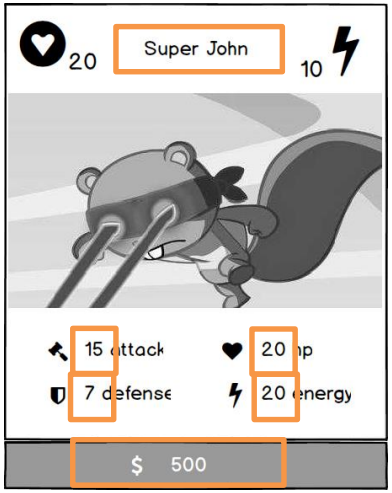
La fonction finale du système est de créer une carte et de l'afficher »

Type de Tests: Exemple

- **Tests Unitaires :**



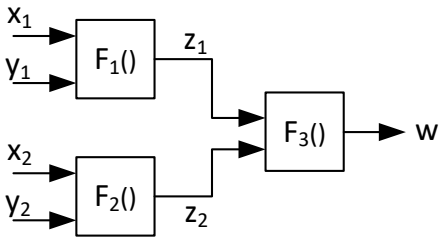
- **Création** d'une carte sans élément
- Ajout d'un **titre**
- Récupération d'un **titre**
- Ajout d'**hp**
- Récupération d'**hp**
- Ajout d'**attack**
- Récupération d'**attack**
- Ajout de **defense**
- Récupération de **defense**
- Ajout d'énergie récupération d'**energy**
- Récupération du **price** calculé
- Récupération de l'**affichage** de la carte

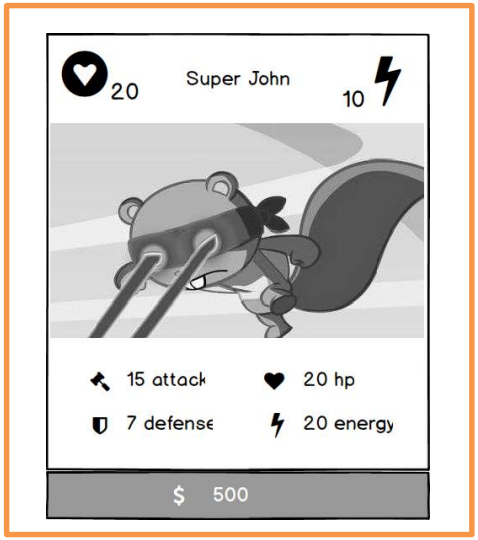


Type de Tests: Exemple

- **Tests d'intégration :**

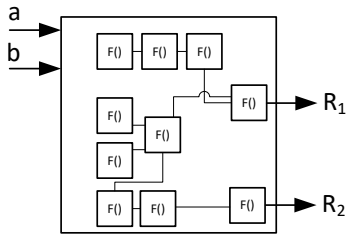
  - **Création** d'une avec tous les paramètres
  - **Affichage** de la carte



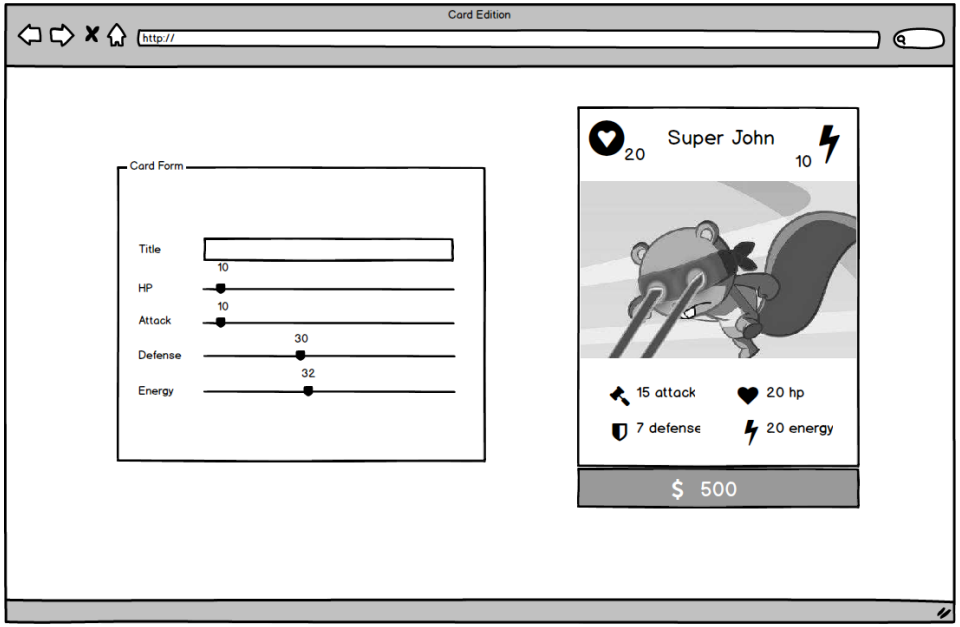


Type de Tests: Exemple

- **Tests Systèmes :**



  - Ajout d'une carte via l'UI
  - Visualisation de la carte via UI



Outils de tests unittest

unittest — Unit testing framework

Source code: Lib/unittest/__init__.py

(If you are already familiar with the basic concepts of testing, you might want to skip to the list of assert methods.)

The unittest unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

To achieve this, unittest supports some important concepts in an object-oriented way:

test fixture
: A *test fixture* represents the preparation needed to perform one or more tests, and any associated cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case
: A *test case* is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.

test suite
: A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner
: A *test runner* is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

https://docs.python.org/3/library/unittest.html

Outils de tests unittest

Classes and functions

This section describes in depth the API of `unittest`.

Test cases

class `unittest.`**`TestCase`**(*methodName='runTest'*)

Instances of the `TestCase` class represent the logical test units in the `unittest` universe. This class is intended to be used as a base class, with specific tests being implemented by concrete subclasses. This class implements the interface needed by the test runner to allow it to drive the tests, and methods that the test code can use to check for and report various kinds of failure.

Each instance of `TestCase` will run a single base method: the method named *methodName*. In most uses of `TestCase`, you will neither change the *methodName* nor reimplement the default `runTest()` method.

Changed in version 3.2: `TestCase` can be instantiated successfully without providing a *methodName*. This makes it easier to experiment with `TestCase` from the interactive interpreter.

`TestCase` instances provide three groups of methods: one group used to run the test, another used by the test implementation to check conditions and report failures, and some inquiry methods allowing information about the test itself to be gathered.

Methods in the first group (running the test) are:

`setUp`()

Method called to prepare the test fixture. This is called immediately before calling the test method; other than `AssertionError` or `SkipTest`, any exception raised by this method will be considered an error rather than a test failure. The default implementation does nothing.

`tearDown`()

Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception, so the implementation in subclasses may need to be particularly careful about checking internal state. Any exception, other than `AssertionError` or `SkipTest`, raised by this method will be considered an additional error rather than a test failure (thus increasing the total number of reported errors). This method will only be called if the `setUp()` succeeds, regardless of the outcome of the test method. The default implementation does nothing.

`run`(*result=None*)

Run the test, collecting the result into the `TestResult` object passed as *result*. If *result* is omitted or `None`, a temporary result object is created (by calling the `defaultTestResult()` method) and used. The result object is returned to `run()`'s caller.

The same effect may be had by simply calling the `TestCase` instance.

Changed in version 3.3: Previous versions of `run` did not return the result. Neither did calling an instance.

`skipTest`(*reason*)

Calling this during a test method or `setUp()` skips the current test. See Skipping tests and expected failures for more information.

New in version 3.1.

`subTest`(*msg=None*, ***params*)

Return a context manager which executes the enclosed code block as a subtest. *msg* and *params* are optional, arbitrary values which are displayed whenever a subtest fails, allowing you to identify them clearly.

A test case can contain any number of subtest declarations, and they can be arbitrarily nested.

See Distinguishing test iterations using subtests for more information.

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

https://docs.python.org/3/library/unittest.html

Outils de tests (exemple)

```
class Calc:
    def __init__(self, name) -> None:
        self._name = name
        self._history = []

    def add(self, a,b):
        result = a + b
        self._history.append(str(a)+'+'+str(b) +'='+str(result))
        return result

    def set_name(self, name):
        self._name = name

    def get_name(self):
        return self._name
```

```
import unittest
from Calc import Calc

class Calc_test(unittest.TestCase):
    def setUp(self) -> None:
        self._current_calc = Calc('test')
        self._current_calc._name = "fake_name"

    def tearDown(self) -> None:
        self._current_calc._history =[]

    def test_add(self):
        result = self._current_calc.add(10,5)
        self.assertEqual(result , 15 , "error during addition")

    def test_setName(self):
        self._current_calc.set_name("new_name")
        name = self._current_calc.get_name()
        self.assertIsNot(name , "fake_name" )
        self.assertEqual(name , "new_name" ,
                                "error when setting name")
if __name__ == '__main__':
    unittest.main()
```

Outils de tests (exemple)

Démo

Exercice

« Application de gestion de cartes: Objectif créer des cartes de jeux et évaluer automatiquement la valeur (price) de chaque carte:
1 Carte est composée:

- Titre String sans caractère spéciaux
- id Integer
- hd compris entre 1-100
- attack compris entre 1-50
- defense compris entre 1-50
- energy compris entre 1-100
- price compris calculer en fonction des caractéristiques

La fonction finale du système est de créer une carte et de l'afficher »

Créer les tests unitaires à l'aide de `unittest` du projet présenté ci-dessus

TDD Test Driven Development