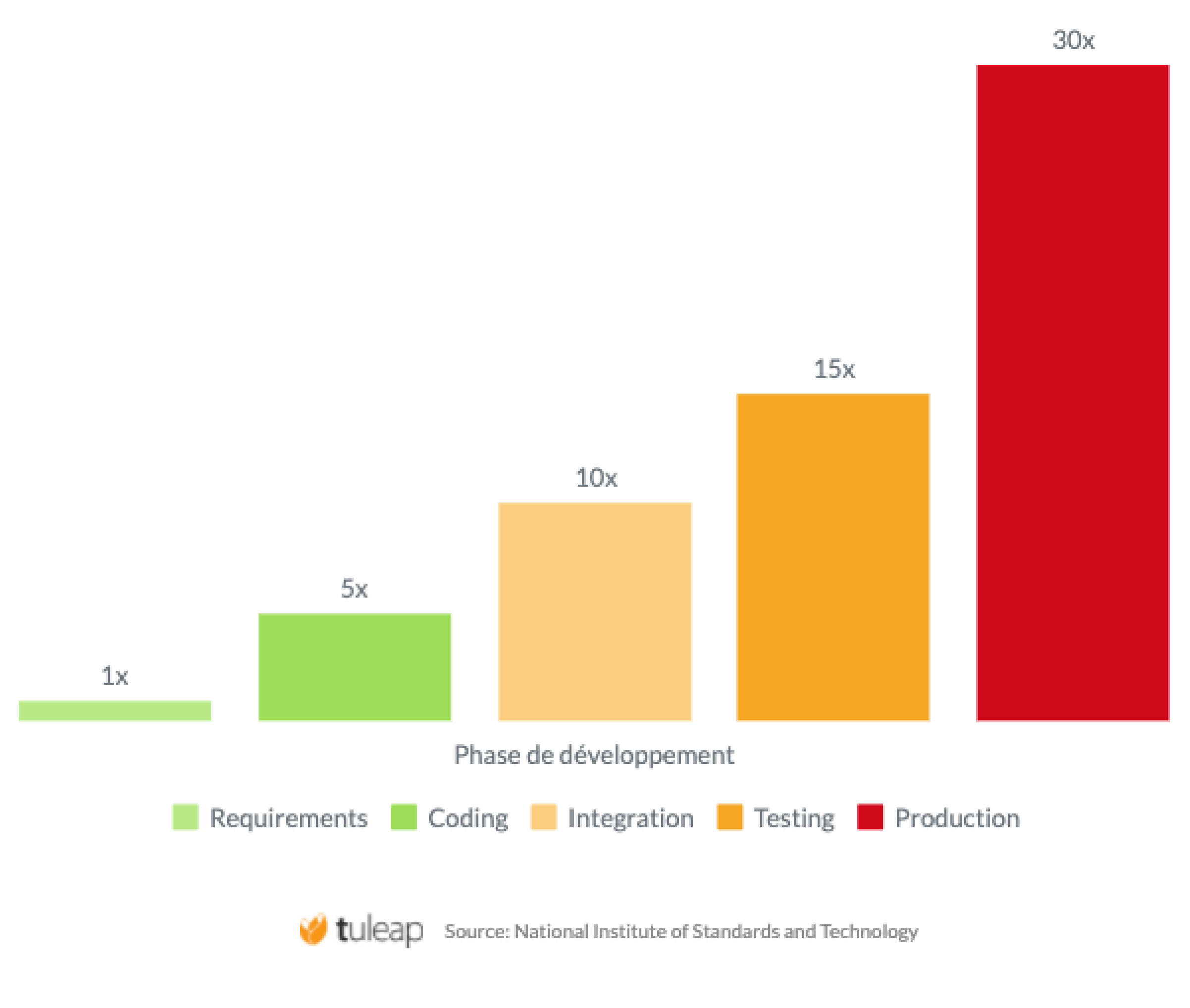
OUTILS DE TEST

POURQUOI TESTER ? [MODE DÉVELOPPEUR PRESSÉ]

- Ouais c'est long et rébarbatif ...
- C'est du temps de perdu
- Je le ferai quand j'aurais du temps , là il faut que je délivre le produit
- J'ai juste changé une ligne
- Ce bout de code est déjà en fonctionnement

POURQUOI TESTER ?

- Mieux gérer les risques : détecter les bugs avant la sortie en production, améliorer le confort des équipes, avoir un filet de sécurité
- Étendre la responsabilité individuelle : s'assurer du niveau de qualité d'un logiciel avant de le mettre à disposition
- Avoir un retour rapide sur votre développement
- Se prémunir des cas d’utilisation invalides
- Eviter d'accumuler de la dette technique avec une application maintenable et évolutive
- C'est un retour sur investissement !



Détectez vos bugs le plus tôt possible!
Coût relatif de détection d'un bug selon l'étape de développement
30x
15x
10x
5x
1x
Phase de développement
Requirements
Coding
Integration
Testing
Production
tuleap
Source: National Institute of Standards and Technology

STRATÉGIE DE TESTS

- L'écriture de tests peut prendre du temps donc cette étape doit être prévue dans le cycle de développement
- Plutôt que prévoir un temps dévolu seulement à l'écriture des tests, faite le au fur et à mesure
- Ou même avant de coder - cf TDD
- Adaptez votre code pour qu'il soit testable, n'adaptez pas vos tests ! Votre code va gagner en lisibilité et découplage

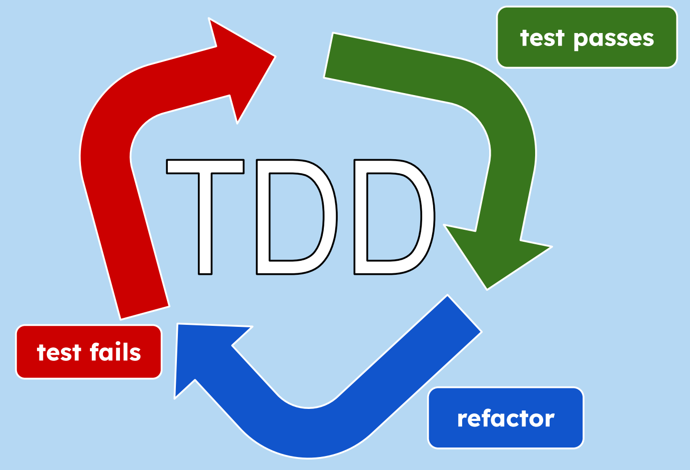
TEST DRIVEN DEVELOPMENT

Développement piloté par les tests (TDD): écrire le test unitaire avant d'écrire le code

Cycle préconisé

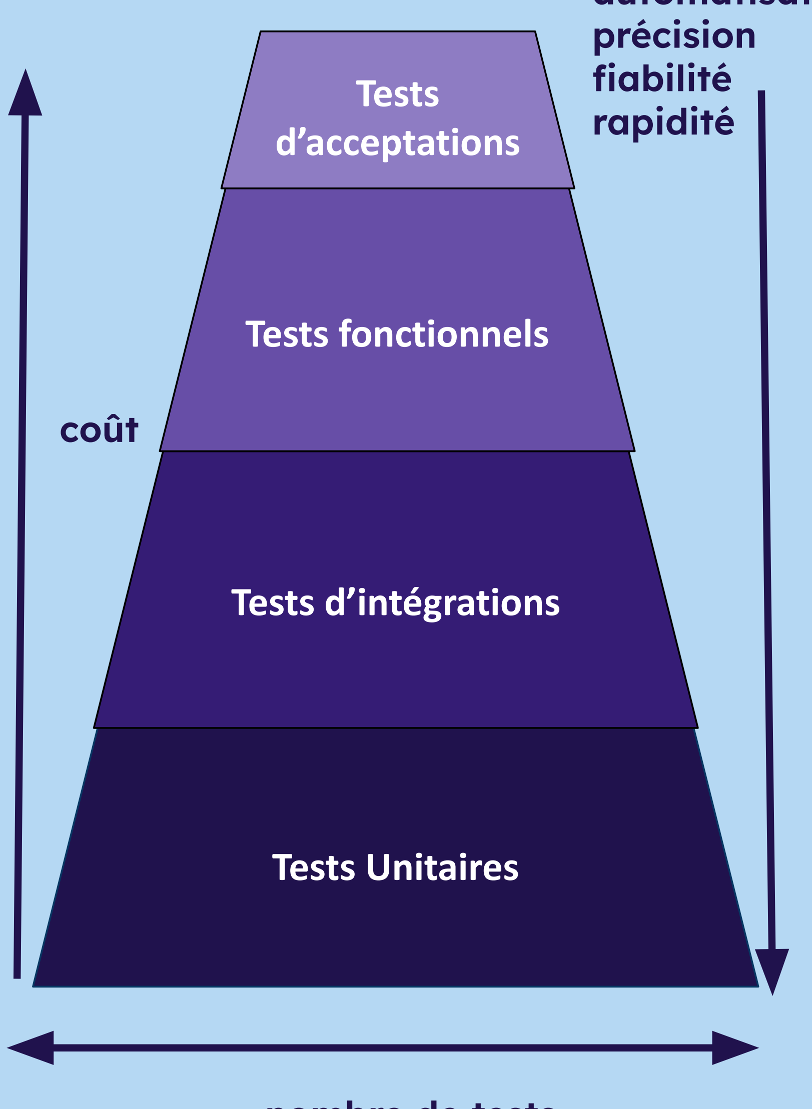
---

1. Écrire le test unitaire
2. Vérifier qu'il échoue
3. Ecrire le code suffisant pour qu'il passe le tests
4. Vérifier que le test passe
5. Nettoyez et refactoriser le code écrit



Les différentes catégories de tests

Vérifications			Validations
Tests Unitaires	Tests d'Intégrations	Tests Systèmes (fonctionnels)	Tests d'Acceptation
Procédure permettant de Vérifier le bon fonctionnement d'une partie précise d'un logiciel : « unité »	Tests effectués pour montrer des défauts dans les interfaces et interactions de composants	Processus de test d'un système intégré pour vérifier qu'il réponde à des exigences spécifiées	Test en rapport avec les besoins, exigences et processus métier, conduit pour déterminer si un système satisfait ou non aux critères d'acceptation
Le code fonctionne correctement seul ?	Le code fonctionne correctement avec le reste ?	Est ce que l'application logicielle "complète" fonctionne ?	Le système satisfait-il les utilisateurs ?



- Est ce que le client est satisfait par cette page de connection ?

- Le comportement attendu est vérifié, est ce que l'utilisateur peut se connecter en cliquant sur le bouton de connexion après avoir entré un nom d'utilisateur et un mot de passe valides ?
- Y a-t-il un message d'erreur qui devrait apparaître sur une connexion invalide?
- Un utilisateur inactivé peut-il se connecter ? [...]

- L'utilisateur voit le message de bienvenue après avoir saisi des valeurs valides et appuyé sur le bouton de connexion.
- L'utilisateur doit être dirigé vers la page d'accueil après une entrée valide et en cliquant sur le bouton Connexion.
- L’utilisateur est redirigé vers une page d’erreur en cas de problème

- Longueur des champ - champs du nom d'utilisateur et du mot de passe.
- Les valeurs des champs de saisie doivent être valides.
- Le bouton de connexion n'est activé qu'après la saisie de valeurs valides dans les deux champs.
- Vérification des expressions régulières (email par ex)

Exemple d’une page de connexion

LES TESTS UNITAIRES

LES TESTS UNITAIRES

Un test est unitaire lorsque :

- Il ne communique pas avec la base de données
- Il ne communique pas avec d’autres ressources sur le réseau
- Il ne manipule pas un ou plusieurs fichiers
- Il peut s’exécuter en même temps que les autres tests unitaires
- Il est automatique
- Il est répétable

LES TESTS UNITAIRES - BONNES PRATIQUES

- 100% des appels externes du composant sont simulés par des bouchons.
- Un test doit être simple, normé, commenté, facile à lire et à comprendre.
- Un test doit être rapide à exécuter (la suite de tests doit pouvoir être exécutée souvent).
- Les tests doivent être écrits le plus tôt possible
- Un test = une assertion.
- Les parties critiques de l'application sont testées en priorité.
- Le code de l'application doit être écrit et refactorisé de façon à être testable.
- Le nom d'un test doit être normalisé (la lecture du nom doit permettre de savoir exactement ce que va faire le test).

LES TESTS UNITAIRES - STRUCTURE "AAA"

- **« Arrange » :** initialisation du contexte d’exécution du test unitaire (variables, données, dépendances, bouchons…)
- **« Act » :** exécution du traitement nécessaire pour la vérification du comportement du code testé
- **« Assert » :** vérification du critère de réussite du test au travers d’une assertion.

LES TYPES DE TESTS

Lors de la rédaction de tests pour une unité, il y a trois types principaux de tests auxquels il convient de penser

- **les tests de cas normaux**, qui vérifient que l'unité se comporte bien dans les situations « normales »
- **les tests de cas d'erreur**, qui vérifient que les erreurs qui doivent être signalées le sont bien, p.ex. lorsqu'un argument invalide est fourni
- **les tests de cas aux limites**, qui vérifient que l'unité se comporte bien dans les situations délicates, p.ex. qu'une méthode qui accepte un tableau de taille quelconque fonctionne correctement s'il est vide

BONNES PRATIQUES : CAS AUX LIMITES

- **pour des arguments entiers** : tester avec 0 ainsi que des entiers positifs et négatifs
- **pour les chaînes de caractères** : tester avec une chaîne vide, une chaîne contenant des caractères quelconques, une chaîne contenant des mots séparés par des espaces, virgules ou des retours à la ligne, voir si les chiffres ou caractères spéciaux sont bien supportés, voir s'il est possible de traiter une très longue chaîne de caractères, ...
- **pour une liste** : tester avec une liste vide, une liste avec un seul élément, une liste contenant des éléments tous différents, une liste contenant plusieurs fois le même élément, …
- **pour des fichiers** : tester avec des fichiers vides, valides et invalides

LES TEST EN PYTHON

FRAMEWORKS POUR PYTHON

Il existe différents frameworks de tests, les plus connus sont **Unittest** et **Pytest**

Alternatives possibles : Doctest , Hypothesis, tox, mock

Pytest est à la fois un framework de test et un exécuteur de test. L'exécuteur de test est un exécutable dans la ligne de commande qui, à un niveau général, peut :

- Effectuer une collecte des tests en recherchant tous les fichiers de test, classes de test et fonctions de test pour une série de tests
- Lancer une série de tests en exécutant tous les tests
- Effectuer le suivi des échecs, des erreurs et de la réussite des tests
- Fournir des rapports complets à la fin d'une série de tests

PYTEST

- Site de référence : https://docs.pytest.org/en/latest/
- Installation avec pip:

```
pip install -U pytest
```

- Execution en mode discovery:

```
pytest
pytest test_mod.py
pytest testing
pytest -k "My Class"
```

  - à partir de testpaths si configuré
  - à partir du répertoire courant
  - avec éventuellement des arguments : répertoire, noms de fichiers, module, classe
  - récursivement sauf indication contraire (norecursedirs)
  - sur les fichiers test_*.py ou *_test.py, ou importés par leur nom de package de test.

ORGANISATION DES TESTS

- Un scénario de test est une classe avec une unité logique groupant un certain nombre de tests
- Un test est une méthode dont le nom est préfixé par test_
- Le cœur de chaque test est un appel à une instruction assert
- Si le test échoue, une exception est levée avec un message explicatif, et pytest identifie le scénario de test comme un échec. Toute autre exception est traitée comme une erreur

</> CODE TEST SIMPLE

```python
# content of test_sample.py
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

EXECUTION

```
$ pytest
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-7.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 1 item

test_sample.py F                                                     [100%]

================================= FAILURES =================================
_______________________________ test_answer ________________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:6: AssertionError
========================= short test summary info ==========================
FAILED test_sample.py::test_answer - assert 4 == 5
============================ 1 failed in 0.12s =============================
```

PRINCIPALES ASSERTIONS

EQUAL TO OR NOT EQUAL TO

```
assert 5 == 5 # Success Example
assert 5 == 3 # Fail Example

---------------------------------------------------------------------------
AssertionError

----> 1 assert 5 == 3 # Fail Example

AssertionError:
assert 5 != 3 # Success Example
assert 5 != 5 # Fail Example

---------------------------------------------------------------------------
AssertionError

----> 1 assert 5 != 5 # Fail Example

AssertionError:
```

ISINSTANCE

```
assert isinstance('5', str) # Success Example

assert isinstance('5', int) # Fail Example

---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)
----> 1 assert isinstance('5', int) # Fail Example

AssertionError:
assert not isinstance('5', int) # Success Example
assert not isinstance('5', str) # Fail Example

---------------------------------------------------------------------------

AssertionError                            Traceback (most recent call last)
----> 1 assert not isinstance('5', str) # Fail Example

AssertionError:
```

TYPE()

```
assert type(5) is int # Success Example
assert type(5) is not int # Fail Example

---------------------------------------------------------------------------
AssertionError
----> 1 assert type(5) is not int # Fail Example
AssertionError:
```

BOOLEAN

```
true = 5==5
assert true is True # Success Example
assert true is False # Fail Example

---------------------------------------------------------------------------
AssertionError
----> 1 assert true is False # Fail Example
AssertionError:
```

GÉRER LES EXCEPTIONS

LA MÉTHODE raises()

SYNTAXE

```
with pytest.raises(exception, match)
```

PARAMETERS

- exception: This is the exception to be thrown.
- match: This parameter can be a literal string or a regular expression that matches the string representation of the exception.

```
import pytest

def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        1 / 0

test_zero_division()
```

ORGANISER LES TESTS AVEC UNE CLASSE

- Les classes augmentent la flexibilité et la réutilisabilité

```
import os

def is_done(path)
    """
    Fonction qui teste le contenu d'un fichier
    Retourne true si le fichier contient yes et false si le fichier contient no ou n'existe pas
    """
    if not os.path.exists(path):
        return False
    with open(path) as _f:
        contents = _f.read()
    if "yes" in contents.lower():
        return True
    elif "no" in contents.lower():
        return False

class TestIsDone:

    def test_yes(self):
        with open("/tmp/test_file", "w") as _f:
            _f.write("yes")
        assert is_done("/tmp/test_file") is True

    def test_no(self):
        with open("/tmp/test_file", "w") as _f:
            _f.write("no")
        assert is_done("/tmp/test_file") is False
```

MÉTHODES D'ASSISTANCE

- **setup :** s’exécute une fois avant chaque test d’une classe
- **teardown :** s’exécute une fois après chaque test d’une classe
- **setup_class :** s’exécute une fois avant tous les tests d’une classe
- **teardown_class :** s’exécute une fois après tous les tests d’une classe

```python
class TestIsDone:

    def teardown(self):
        if os.path.exists("/tmp/test_file"):
            os.remove("/tmp/test_file")

    def test_yes(self):
        with open("/tmp/test_file", "w") as _f:
            _f.write("yes")
        assert is_done("/tmp/test_file") is True

    def test_no(self):
        with open("/tmp/test_file", "w") as _f:
            _f.write("no")
        assert is_done("/tmp/test_file") is False
```

QUAND UTILISER UNE CLASSE AU LIEU D’UNE FONCTION ?

- Lorsque les tests ont besoin d’une configuration ou d’un nettoyage similaires
- Lorsqu’il est logique de les regrouper
- S’il y a au moins quelques tests
- Lorsque les tests peuvent bénéficier d’un ensemble commun de méthodes d'assistances

USAGES AVANCÉS

LES MOCKS

DÉFINITION

Les **mocks** permettent

- de simuler le comportement des classes, méthodes, constantes ...
- de créer tous les attributs et méthodes nécessaires à une simulation
- de garder vos tests "unitaires" en maîtrisant le comportement des dépendances

---

INSTALLATION

```
pip install pytest-mock
```

FONCTIONNEMENT

- La méthode `mocker.patch.object()` permet de mocker une constante.
- La méthode `mocker.patch` permet de mocker une fonction ou un objet.

LES MOCKS : EXEMPLE

```python
# src/example.py
from datetime import datetime

def get_time_of_day():
    """return string Night/Morning/Afternoon/Evening depending on the hours range"""
    time = datetime.now()
    if 0 <= time.hour <6:
        return "Night"
    if 6 <= time.hour < 12:
        return "Morning"
    if 12 <= time.hour <18:
        return "Afternoon"
    return "Evening"
```

```python
import pytest
from datetime import datetime
from src.example import get_time_of_day

def test_get_time_of_day(mocker):
mock_now = mocker.patch("src.example.datetime")
mock_now.now.return_value = datetime(2023, 5, 20, 14, 10, 0)

assert get_time_of_day() == "Afternoon"
```

DOCTEST : D'UNE PIERRE 2 COUPS

Placez vos test dans les docstrings des fonctions !

- La ligne qui commence par trois chevrons (>>>) contient l'instruction à exécuter
- la ligne suivante contient la représentation textuelle du résultat attendu

```python
def add_numbers(a, b):
    """Sums the given numbers.

    :param int a: The first number.
    :param int b: The second number.

    :return: The sum of the given numbers.

    >>> add_numbers(1, 2)
    3
    >>> add_numbers(50, -8)
    42
    """
    return a + b
```

```
$ python -m doctest operations.py
```

LES TESTS DES APPLICATIONS WEB

L'APPROCHE FONCTIONNELLE

- En pratique même si on peut tester le fonctionnement précis du controller avec des tests unitaires
- On va plutôt tester les différentes couches de l'application avec des tests fonctionnels
  - Effectuer une requête auprès de l’application, simuler un accès de page
  - Parcourir le DOM (le code HTML de la page)
  - Contrôler la réponse du serveur

EXEMPLE : FORMULAIRE DE CRÉATION D'UN ARTICLE

Create new Article

Title

Symfony: les tests de bout-en-bout

Description

Cet article décrit les tests de bout-en-bout avec Symfony

Content

We love Panther !

Save

back to list

© 2022 - Built with ♥ by Sooyoos.

Articles

Id	Title	Description	CreatedAt	actions
1	Symfony: les tests de bout-en-bout	Cet article décrit les tests de bout-en-bout avec Symfony	2022-01-21 10:41:39	show edit

Create new

© 2022 - Built with ♥ by Sooyoos.

EXEMPLE : PHPUNIT

```
class ArticleTest extends WebTestCase
{
    public function testCreateArticle(): void
    {
          $client = static::createClient(); // Création du client
          $client->request('GET', '/article/'); // Requete de type GET sur l'url de la page
          $client->followRedirects(); // le client doit suivre les redirections (formulaire)

          $this->assertPageTitleSame('Articles'); // vérification du titre
          $client->clickLink('Create new'); // simulation d'un nouvel article

          $this->assertPageTitleSame('New Article'); // vérification du titre du formulaire
          $this->assertSelectorTextSame('h1', 'Create new Article');

          $client->submitForm('Save', [
              'article[title]' => 'Symfony: les tests de bout en bout',
              'article[description]' => 'Cet article décrit les tests',
              'article[content]' => 'We love Panther !'
          ]); // Soumission du formulaire
          $this->assertSelectorTextContains('table', 'Symfony: les tests de bout en bout'); // vérification de la table
    }
}
```

L'APPROCHE FONCTIONNELLE : EXEMPLES DE TESTS À METTRE EN OEUVRE

- Tests de statut de page : autorisation, 404 ...

```
// Test d'accès à une page restreinte
$this->crawler = $this->client->request('GET', '/access_restricted');
$this->assertEquals(Response::HTTP_FORBIDDEN, $this->client->getResponse()->getStatusCode());
$this->authentication->logIn();
$this->assertEquals(Response::HTTP_OK, $this->client->getResponse()->getStatusCode());

// Test d'accès à une page avec des paramètres
$this->crawler = $this->client->request('POST', '/ma_page/parameters/');
$this->assertEquals(404,$this->client->getResponse()->getStatusCode(), "Missing form paramete
$this->crawler = $this->client->request('POST', '/ma_page/parameters/param1');
$this->assertEquals(Response::HTTP_OK, $this->client->getResponse()->getStatusCode());
```

LIMITATION DE CETTE MÉTHODE

- Approche systématique des tests unitaires manquante
- Assertions sur la réponse limitée aux retour de votre serveur
- Css et javascripts non testés

LA SOLUTION COMPLÉMENTAIRE : SELENIUM

- Selenium est une suite d’outils permettant d’automatiser des interactions avec les navigateurs Web.

Selenium IDE

- L’IDE de Selenium se présente sous la forme d’une addon pour navigateur Web. Il permet d’enregistrer des scénarios de test à partir des actions de l’utilisateur. Cette solution s’adresse avant tout à des personnes ne maîtrisant par la programmation.

Selenium WebDriver

- Selenium WebDriver fournit une API permettant d’interagir avec un navigateur web de manière programmatique dans différents langages (Java, C#, Python, JavaScript, Ruby)

Selenium Grid

- Grid permet d’exécuter les tests développés à partir de l’API du WebDriver sur plusieurs types de navigateurs Web et même sur plusieurs systèmes d’exploitation.