# Automatisation des tests

Sandy Ingram

# Plan du cours



- Définition du tests agile
- Approches de tests (TDD, BDD)
- Type de tests (tests unitaires, tests d'intégration, tests systèmes, tests non fonctionnels)
- Pyramide de tests
- Quadrants de test agiles
- Doublons de tests

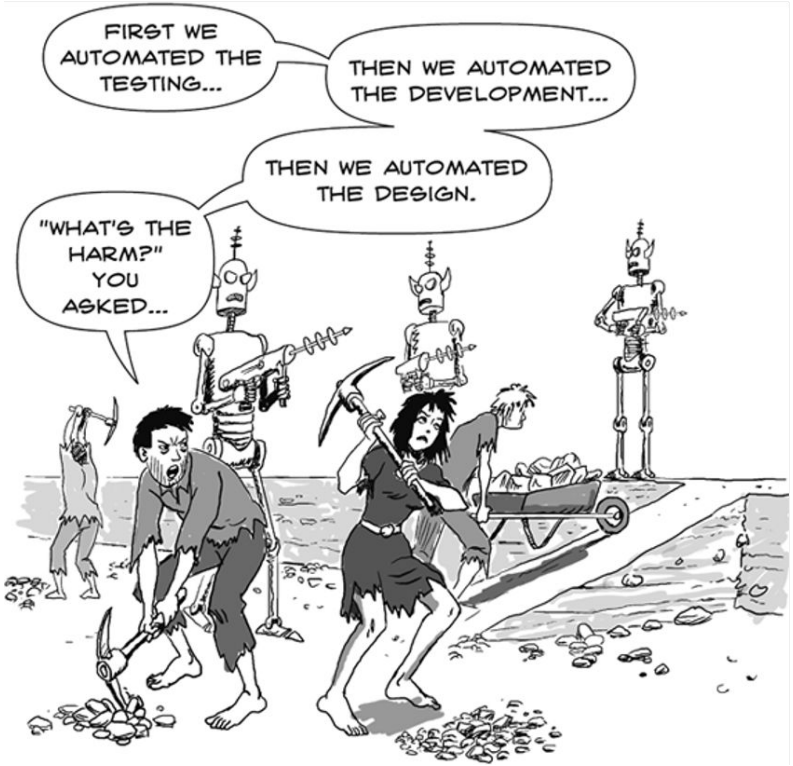
# Définition de “tests agiles”

# Test Agile

- Automatiser les tests pour faciliter les tests “continus” dans le cadre du développement agile (CI/CD).
  - les tests manuels doivent être réduits au minimum.
- Résoudre les “bugs” le plus vite possible, sans **délai**.
- Lier le développement au test et à l’assurance qualité (les rôles peuvent aussi se chevaucher).
- Tester **tôt**.

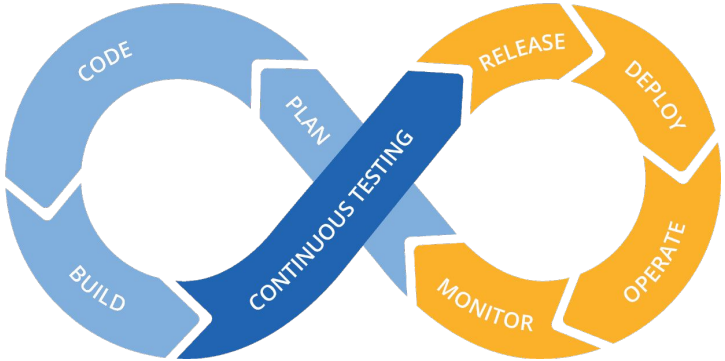
# A propos de l'automatisation des tests





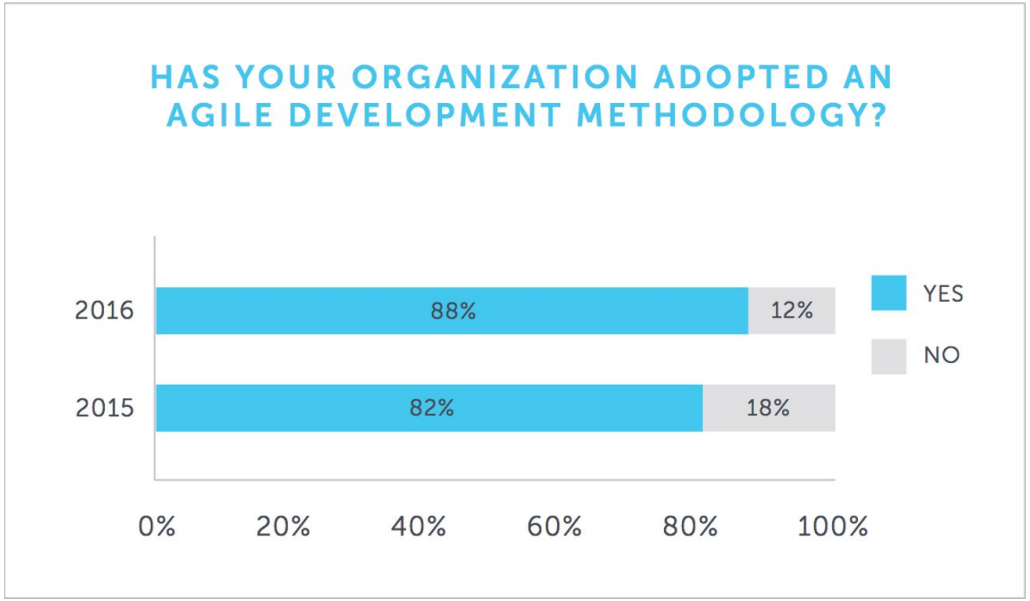
# L’automatisation des tests permet les tests “continus”





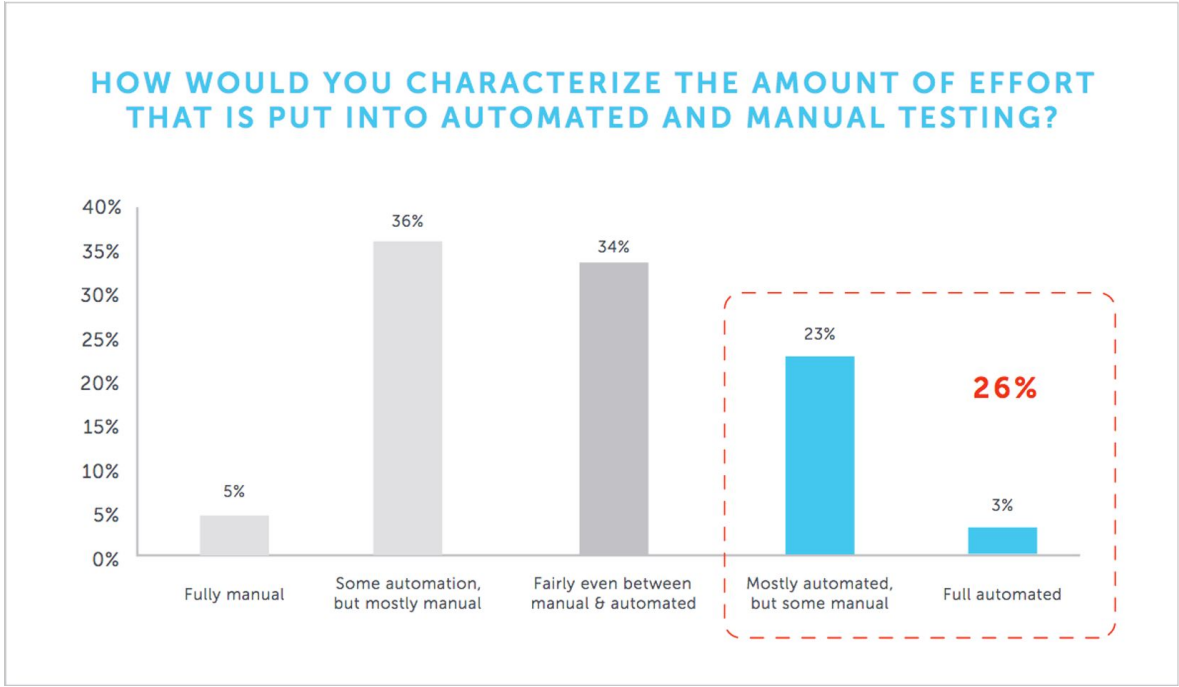

- L’automatisation des tests facilite la maintenance du code.
- Les tests manuels doivent être réduits au minimum (~<15%).
- Les tests doivent être **automatiquement** lancés à chaque mise à jour du code.
- Les tests automatisés sont intégrés dans le pipeline CI/CD.

# Alors même que le développement agile est relativement bien répandu …



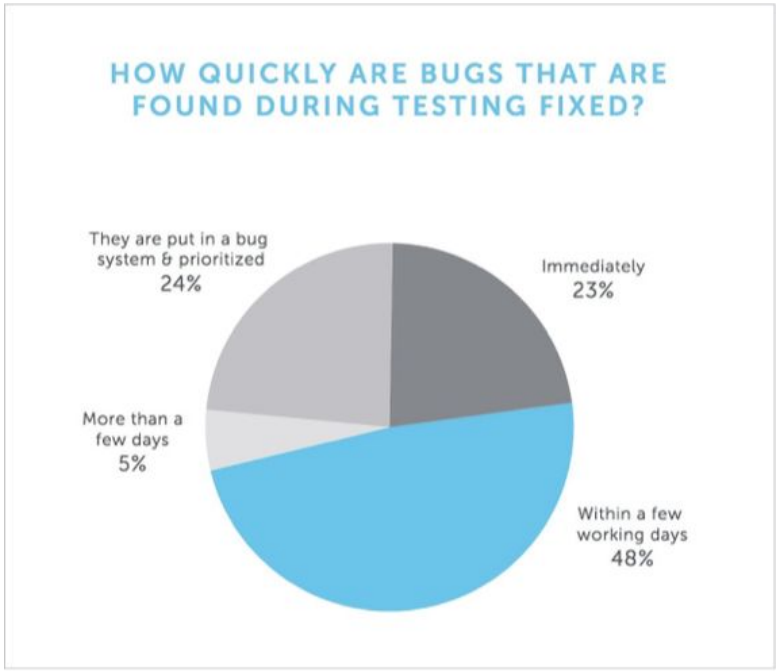

*2016 Report by Sauce Labs involving 520 software professionals*

# Les tests automatiques n'ont pas évolué à la même vitesse que d'autres aspects du développement agile



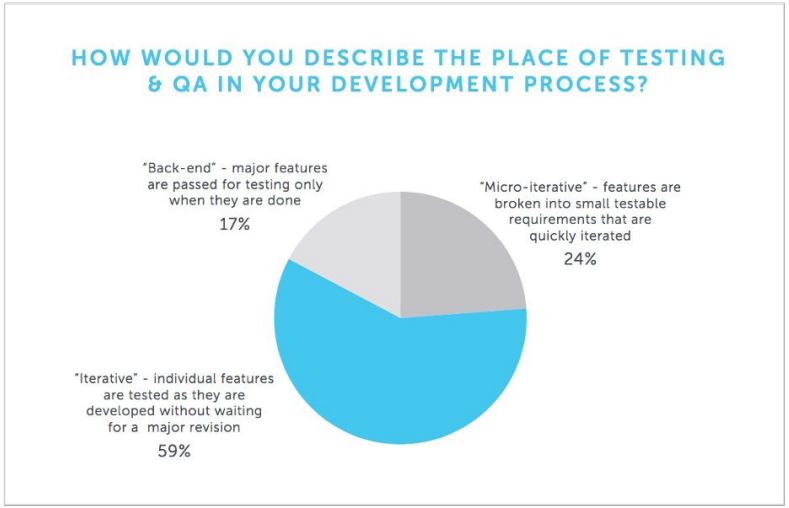
# Résolution des “bugs” au plus vite






*2016 Report by Sauce Labs involving 520 software professionals*

# Test tôt



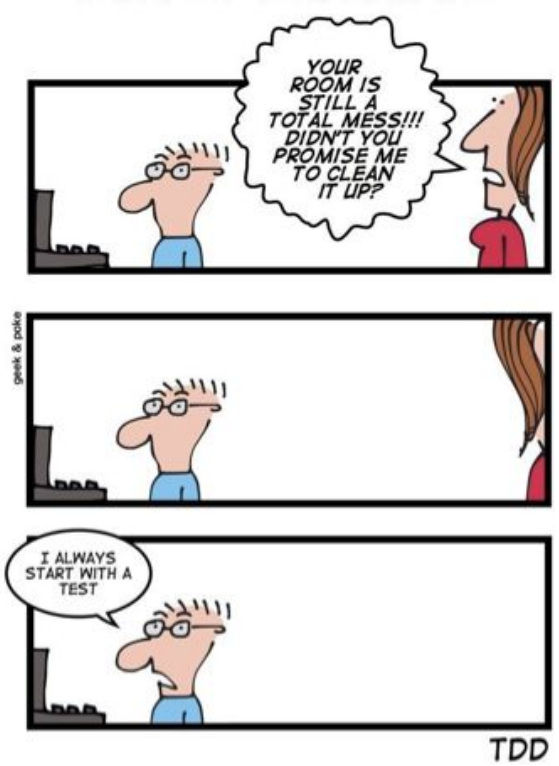

*2016 Report by Sauce Labs involving 520 software professionals*

- Diviser chaque fonctionnalité en petites unités testables.
- Soumettre le code **avec** ses tests
- Soumettre les tests pour l'assurance qualité, avant même la réalisation complète de la fonctionnalité à développer

# Approches de test
# TDD, BDD

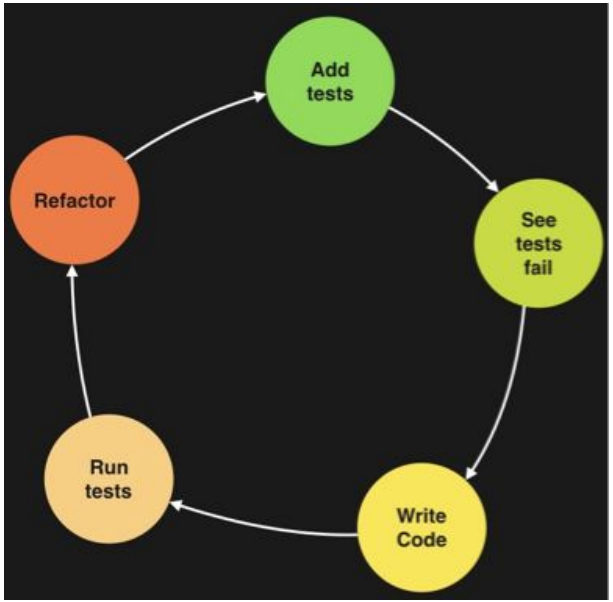
# TDD pour “Test Driven Development”

- TDD consiste en un processus itératif de test consistent à réaliser le test **avant** le code de la fonction à développer.
- TDD est indépendant du langage de programmation et applicables à différents types ou niveau de tests.



# TDD en bref

1. Définir les inputs et outputs.
2. Choisir une signature pour la fonction à réaliser.
3. Se concentrer sur un seul aspect de cette fonction.
4. Implémenter d'abord le test pour cet aspect. Le test doit **échouer** à ce stade, autrement il ne teste pas cet aspect en particulier!
5. Implémente la fonction correspondante.
6. Refactoriser le code si applicable.
7. Reprendre depuis l'étape 3.



# Avantages du TDD

- Garantit une couverture de tests élevée (proportion de code actuellement testée).
- Réduit les “bugs”.

# BDD pour “Behavior Driven Development”

- Se focalise sur le “comportement” (est-ce bien le comportement attendu qui est réalisé ou pas?) et non pas sur comment ce comportement a été codé.
- Le comportement est exprimé avec une syntaxe compréhensible

# BDD pour “Behavior Driven Development”

- BDD augmente la compréhension mutuelle entre développeurs “business” et IT.
- Les “stakeholders” expriment leur besoins en rédigeant des scénarios de test “haut niveau”
- Les développeurs traduisent ces besoins en tests automatiques.
- BDD permet de formaliser et **lier explicitement** les scénarios de test haut-niveau avec le test correspondant.

# Exemple de BDD avec la librairie Cucumber

Les besoins (features) sont exprimés avec *Gherkin* en se concentrant sur le “quoi” et pas le “comment”

*Feature*: *Users must be able to search for content using “the Search” button.*

*Scenario*: *Search for a term.*

*Given* *Given I have entered “watir” into the query.*

*When* *I click “Search"*

*Then* *I should see some result.*

https://www.tutorialspoint.com/cucumber/cucumber_ruby_testing.htm

# Voici un exemple complet réalisé avec Cucumber

Slide optionnel uniquement pour information

## Feature Description File (Gherkin):

```
Feature: Addition
        Test if calculator adds two positive numbers correctly

        Scenario: Addition of two positive numbers
                Given I have number 2 in calculator
                When I entered number 3
                Then I should see result 5
```

## Tested module

```
const { setWorldConstructor } = require('cucumber')

class CustomWorld {
  constructor() {
  this.variable = 0
  }

  setTo(number) {
    this.variable = number
  }

  incrementBy(number) {
    this.variable += number
  }
}

setWorldConstructor(CustomWorld)
```

## The corresponding test

```
const { Given, When, Then } =
require('cucumber');
const assert = require('assert')
                        Regular expressions
//// Your step definitions /////
Given(/^I have number (\d+) in
calculator$/, function (num) {
    this.setTo(num);
});

When(/^I entered number (\d+)$/, function
(num) {
    this.incrementBy(num);
});

Then(/^I should see result (\d+)$/,
function (result) {
    assert.equal(this.variable,
parseInt(result));
});
```

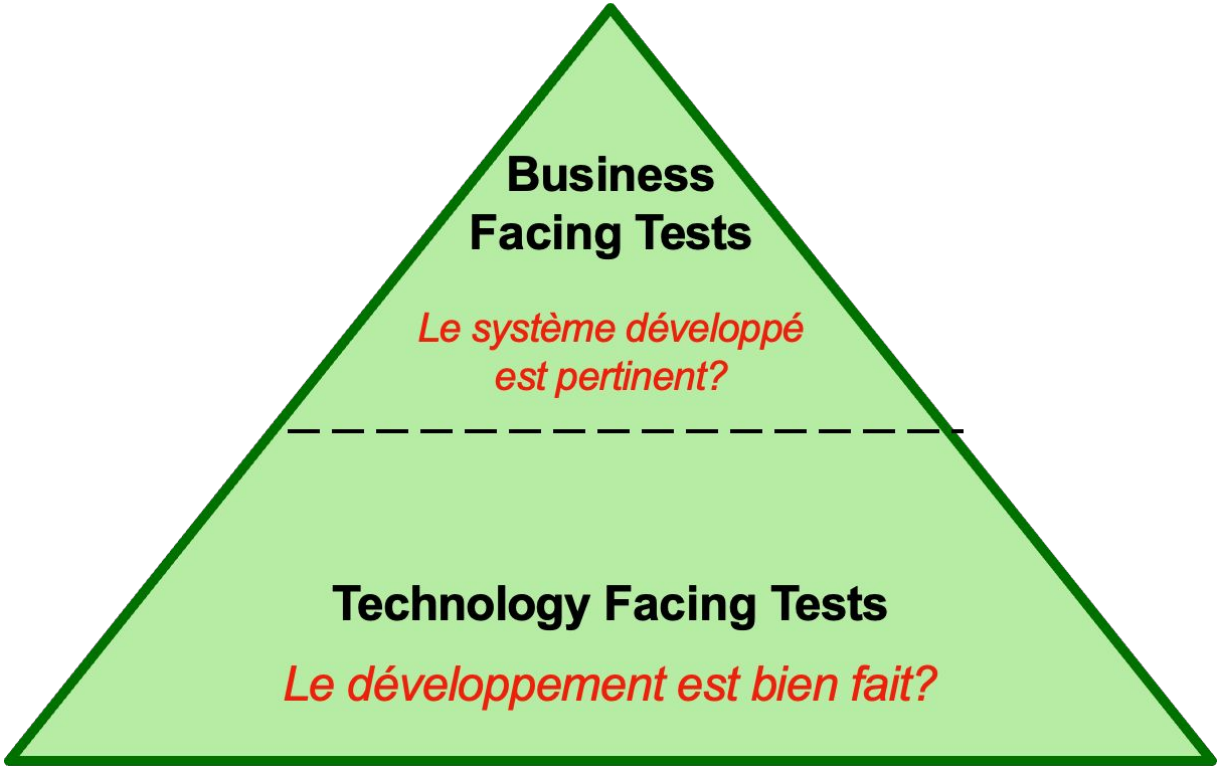
# Synthèse des approches de tests

- TDD se concentre sur rédiger le test du code **avant** le code à tester.
- BDD se concentre sur tester le comportement d'une fonction par rapport à la spec ou aux besoins business.
- Les deux approches sont tout à fait complémentaires.

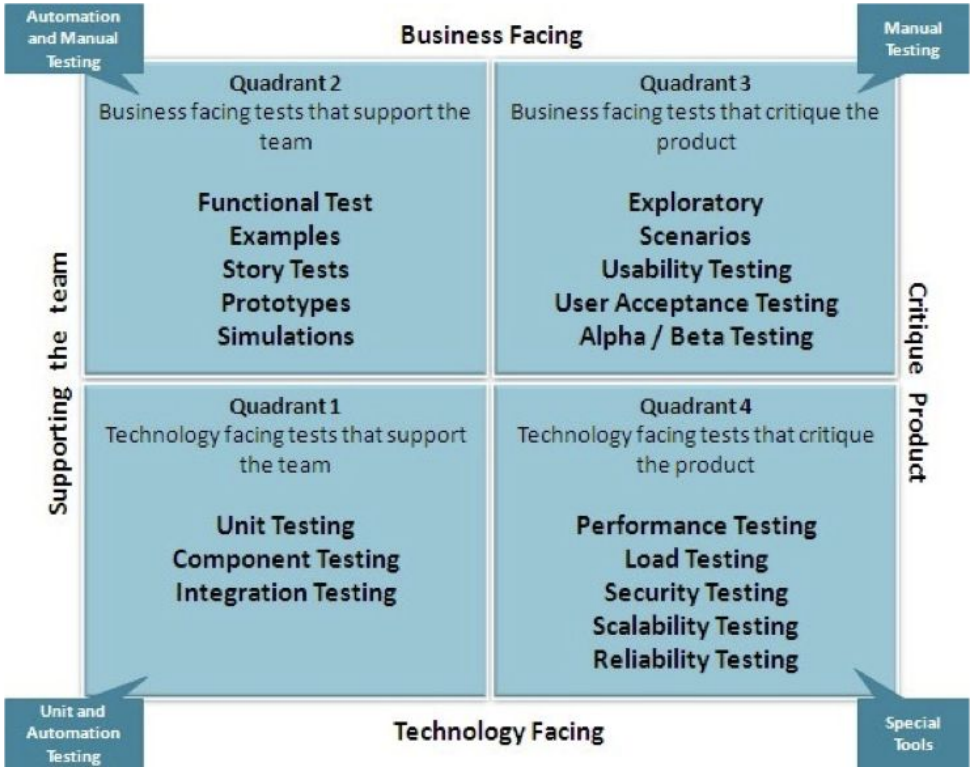
# Quadrants de tests

# Tests orientés business vs tests orientés technologies



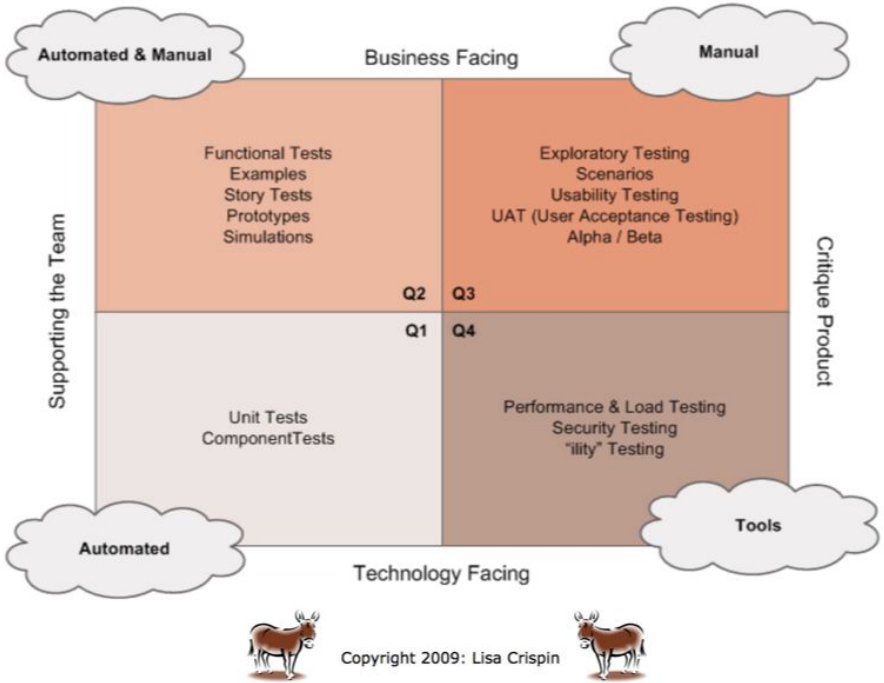


# Quadrants de tests Agile




http://tryqa.com/what-are-test-pyramid-and-testing-quadrants-in-agile-testing-methodology/

# Une autre représentation des quadrants Agiles de Tests



# Types (et étapes) de tests

# Q1 tests unitaires

- Est écrit par les développeurs de la fonction/unité testée.
- Teste si une unité fonctionne comme attendu, de manière **isolée**, répétée, et automatisée.
- Requiert habituellement des doublons de tests afin de pouvoir tester une unité de code séparément de ces dépendances et pouvoir reproduire le test; Si le test efface ou modifie les doublons de tests c'est OK, car ces derniers sont recréés à chaque test.

# Q1 Tests de composants

- Est écrit par les développeurs.
- **White Box** testing: se concentre sur le code du composant, sans se soucier des services externes et en dépendre.
- Les **objets réels** sont utilisés pour toutes les fonctions ou classes propres au composant (contrairement au test unitaire).
  un doublon = testing double
- Les **doublons de tests** ne concernent que les classes ou fonctions d'autres composants que celui testé.

# Q1 (du quadrant agile) Tests d'intégration

- Est aussi écrit par les développeurs.
- **Black Box** testing: teste l'intégration entre des composants, les (micro-)services externes.
- Plus lent que les tests unitaires, mais plus rapides que les tests d'acceptance (ou test systèmes) qui implique ***tous*** les composants d'un système.

# Types de tests

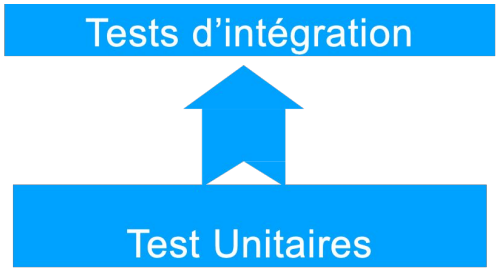


Test Unitaires

*Tester si chaque fonction, ou unité de code se comporte comme attendu*

# Types de tests



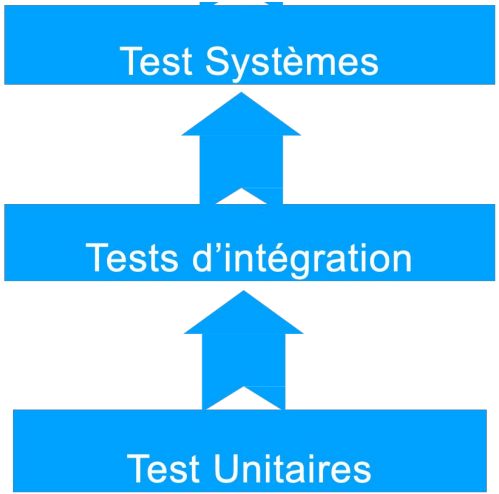



*Tester si une combinaison ou un ensemble d'unités pour voir s'il y a des erreurs liées à **l'interaction** entre ces unités.*

*Tester si chaque fonction, ou unité de code se comporte comme attendu*

# Types de tests

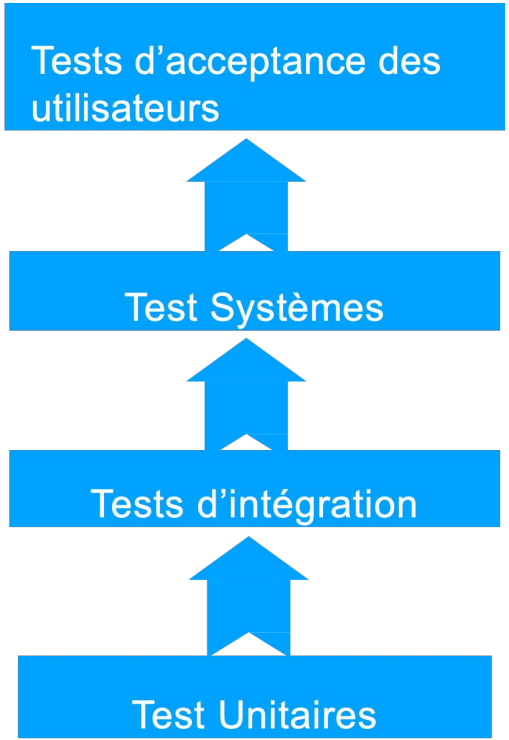





*Tester si une combinaison ou un ensemble d'unités pour voir s'il y a des erreurs liées à **l'interaction** entre ces unités*

*Tester si chaque fonction, ou unité de code se comporte comme attendu*

# Types de tests




*"**User** Acceptance Tests": tester l'acceptabilité du système, son adéquation avec les besoins business*

*Aussi dits "acceptance tests" ou tests E2E (end to end): tester un système dans son entité (à ne pas confondre avec les User Acceptance Tests".*

*Tester si une combinaison ou un ensemble d'unités pour voir s'il y a des erreurs liées à **l'interaction** entre ces unités*

*Tester si chaque fonction, ou unité de code se comporte comme attendu*

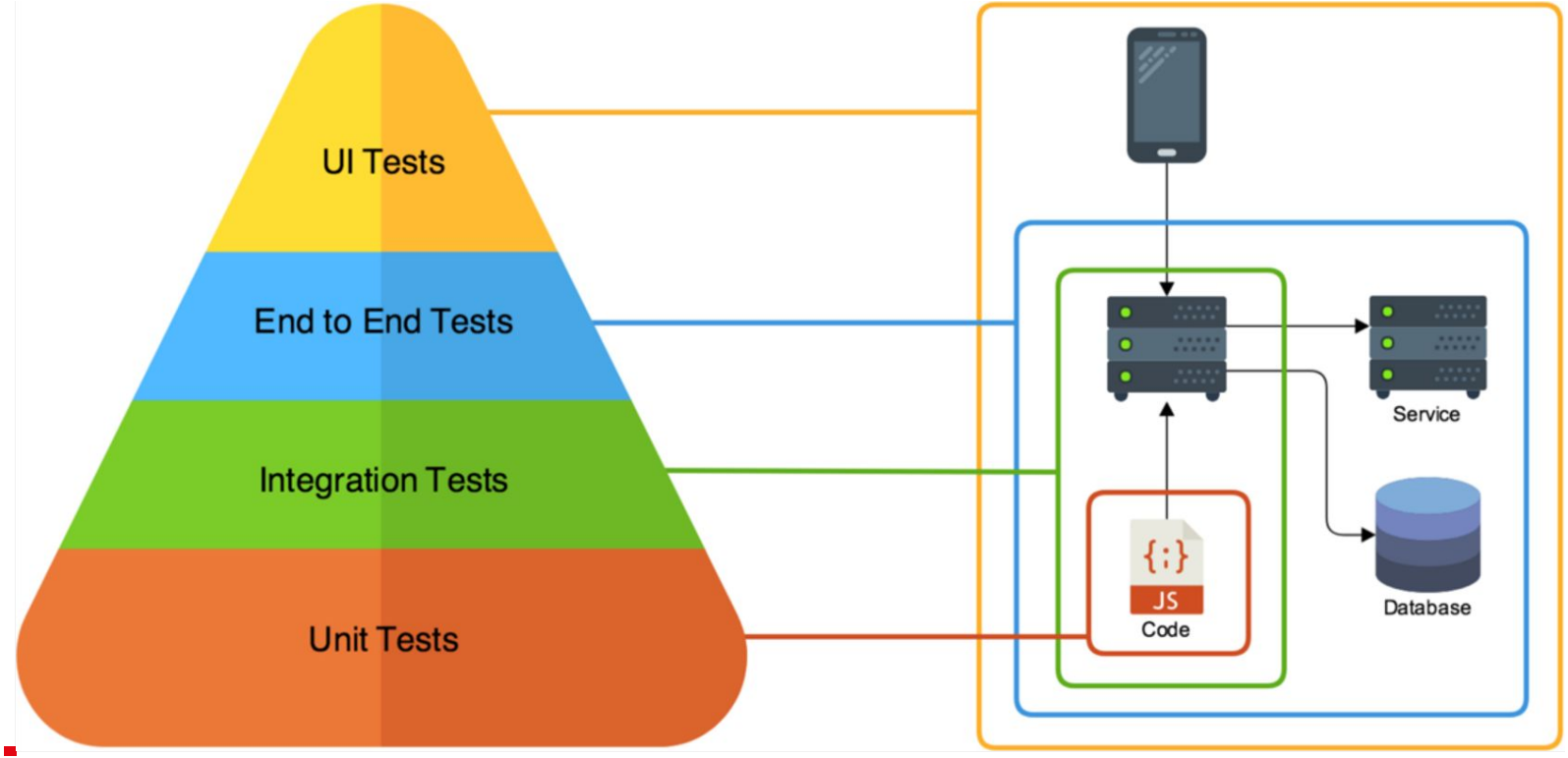
# Exemple concrets de tests pour mieux comprendre

- **Test unitaire:** tester si quand je clique sur un bouton de “sauvegarder” avec les champs correctement remplies(en supposant que toute la communication avec les restes des modules émulés fonctionnent correctement), je reçois un message “mesures sauvegardées”.

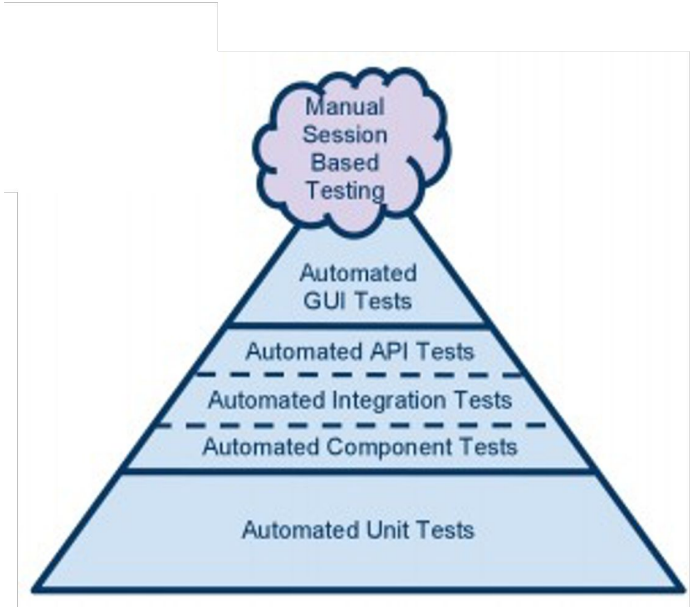
- **Test d’intégration:** *communication backend et base de données, ou communication capteurs et serveur communication interface utilisateurs et serveur backend. Par exemple, tester si le serveur backend envoie à la base de données des données valides à sauver, indépendamment de tout le reste, cette communication fonctionne bien et il reçoit en retour l’ID de l’entrée rajoutée dans la base de données.*

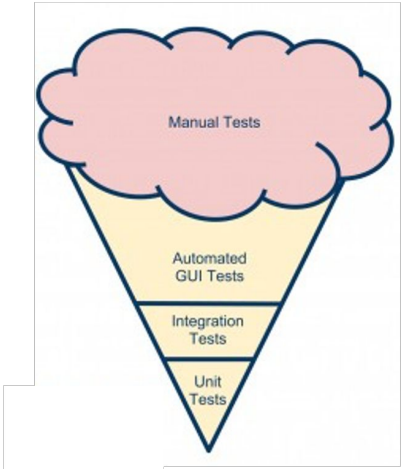
- ***Test E2E:*** tester si quand je clique sur un bouton de “sauvegarder” avec les champs correctement remplies, une requête est envoyée au serveur Web lequel communique avec la base de données, les données envoyées sont bel et biens sauvées et je reçois un message “mesures sauvegardées”.

# Quelles sont les deux conclusions principales de cette pyramide de tests?



# Glace ou Pyramide?



# Doublons de tests

# Doublons de tests

Comme les cascades au cinéma

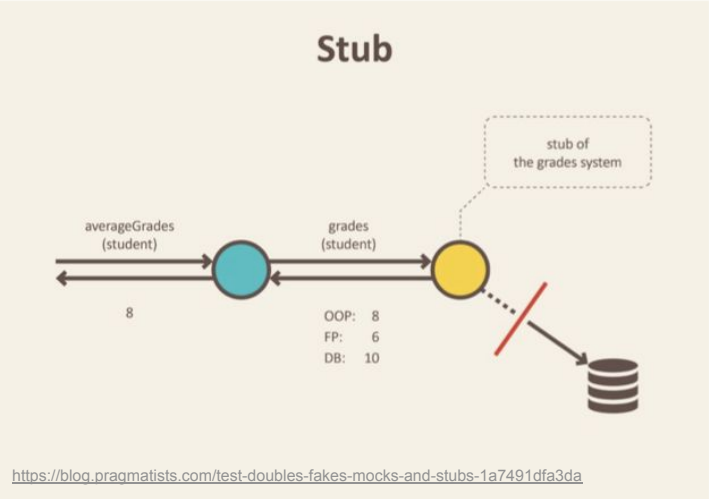
https://www.vulyplay.com/img/blog/trampoline-stunts-movies-pyrotechnics.jpg

# Types de doublons de tests

- Le terme de “Test Doubles” a été introduit par Meszaros dans son livre “xUnit Test Patterns”.
- On en distingue trois types: Les fakes, stubs, et les mocks.
- Dans certains cas, on parle aussi d’un 4ème type: les spys.

# Les “Stubs”

- Ne contiennent pas de logique.
- Quand ils sont appelés, ils retournent des valeurs prédéfinies pour permettre au test d’avancer à la prochaine étape.
- Ils remplacent donc un vrai appel retournant des vrais valeurs.

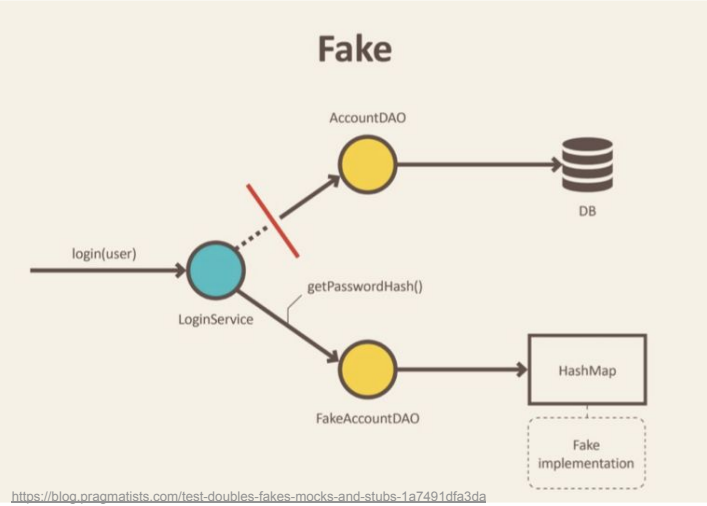



https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

Pour tester la fonction “averageGrades” en **isolation** de la DB, on remplace l’appel à la DB, pour un retour de valeurs prédéfinies **isolant** tout éventuel problème liée à la communication avec la DB et se concentrer sur le test de la fonction de calcul de la moyenne pour autant qu’elle reçoive les données attendues.

# Les “Fakes”

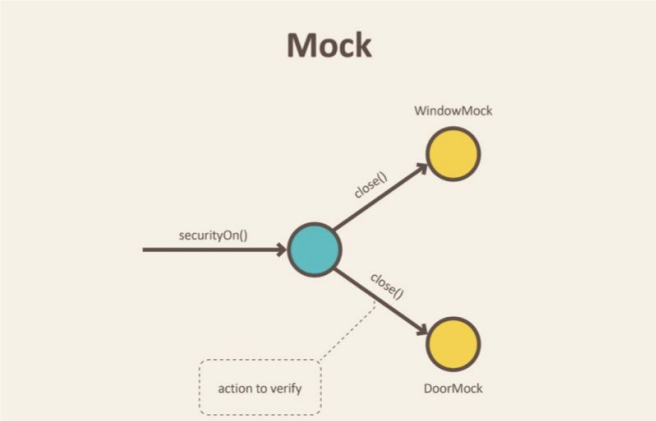
- En comparaison avec les “stubs”, les “fakes” sont un peu plus proches de l’objet réel.
- En général, celui qui gère l’objet réel, écrit aussi son “fake”.




https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

# Les “Mocks”

- Les “Mocks” permettent de tester l’interaction entre deux composants ou fonctions, services.
- **Ils sont typiquement utiles, lorsque les fonctions impliquées dans le test, ne retournent pas de valeurs.**
- Le test se concentre sur ***si*** la fonction a été appelée au bon moment (ou encore si elle a été appelée autant de fois qu’attendu).




https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da

Le test échoue si la fonction testé n’est pas appelée comme attendu.

# “Stub” ? “Mock” ?

```
public interface MailService {
  public void send (Message msg);
}
public class MailServiceStub implements
MailService {
  private List<Message> messages = new
ArrayList<Message>();
  public void send (Message msg) {
    messages.add(msg);
  }
  public int numberSent() {
    return messages.size();
  }
}
```

*// We can then use state verification on the stub like this.*

class OrderStateTester...

```
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new
MailServiceStub();
    order.setMailer(mailer);
    order.fill(warehouse);
    assertEquals(1, mailer.numberSent());
  }
```

•

Class OrderInteractionTester...

```
public void  testOrderSendsMailIfUnfilled()

{

        Order order = new Order(TALISKER, 51);

        Mock warehouse = mock(Warehouse.class);

        Mock mailer = mock(MailService.class);

        order.setMailer((MailService)

mailer.proxy());

mailer.expects(once()).method("send");
warehouse.expects(once()).method("hasInventory")

            .withAnyArguments()

            .will(returnValue(false));

order.fill((Warehouse)

warehouse.proxy());

    }

}
```

https://martinfowler.com/articles/mocksArentStubs.html

# Que fait ce test?

```
"??????????" : function(){
    var message = 'an example message';
    var stub = sinon.stub().throws();
    var spy1 = sinon.spy();
    var spy2 = sinon.spy();

    PubSub.subscribe(message, stub);
    PubSub.subscribe(message, spy1);
    PubSub.subscribe(message, spy2);

    PubSub.publishSync(message, undefined);

    assert(spy1.called);
    assert(spy2.called);
    assert(stub.calledBefore(spy1))
    ;
}
```

- Dans la librairie Sinon.js, de laquelle cet exemple a été extrait :
- Les Stubs permettent de modifier les comportements, les valeurs de retour du doublons de test (par exemple lever une exception).
- Les Spys permettent simplement d'enregistrer les appels de fonction, les valeurs de retour, et les éventuelles exceptions levées.
- Les Mocks API, implémente les Stubs et les Spy APIs.

# Que fait ce test?



```
"test should call all subscribers, even if there are exceptions" : function(){
    var message = 'an example message';
    var stub = sinon.stub().throws();
    var spy1 = sinon.spy();
    var spy2 = sinon.spy();

    PubSub.subscribe(message, stub);
    PubSub.subscribe(message, spy1);
    PubSub.subscribe(message, spy2);

    PubSub.publishSync(message, undefined);

    assert(spy1.called);
    assert(spy2.called);
    assert(stub.calledBefore(spy1));
}
```

**Why is a stub used here and not a spy?**

# Réalisation du test avec un mock

```
"test should call all subscribers even with exceptions": function ()
{
    var myAPI = { method: function () {} };
    var spy = sinon.spy();
    var mock = sinon.mock(myAPI);
    mock.expects("method").once().throws(); // throw an exception
    PubSub.subscribe("message", myAPI.method);
    PubSub.subscribe("message", spy);
    PubSub.publishSync("message", undefined);
    mock.verify();
    assert(spy.calledOnce);
}
```

https://github.com/cypress-io/sinon/blob/master/docs/_releases/v2.4.0/mocks.md

# Spécificités des tests systèmes embarqués

- Sont plus chronophages que les tests de logiciels
- Sont plus coûteux surtout s'il y a des "bugs" à résoudre (quand le système est déjà assemblé)
- Dépendant au prototype "hardware" donc moins flexible que les tests de de logiciels
- Nécessite une connaissance de l'architecture logicielle mais aussi des composants "hardware": capteurs, PCBs ou circuits imprimés, dispositifs IOT (Internet of things).

# Références



- Scenario Outlines with Cucumber (Scenario Templates)
- xUnit Test Patterns Book
- Test Doubles
- Testing trends in 2016
- Testing Pyramid (Unit, Service, UI) by Mike Kohn.
- Agile testing Automation
- Agile testing matrix by Brian Marick.
- Agile testing book with Agile test Quadrants
- Azure devOps: creating test plans and test suits.