



Code
Garage



Introduction aux tests logiciels

My-Serious-Game



Qui suis-je ?

Nicolas Brondin-Bernard

- Formateur depuis 2021
- Freelance depuis 2016
- Master en Architecture Logicielle

1 an chez My-Serious-Game en 2017-2018



La formation

Matinée

- Conception logicielle
- Gestion de projet
- Tests
 - Tests unitaires
 - Exercices
- Bonnes pratiques
- Autour des tests

Après-midi



Mise en pratique



Objectif



“Que vous testiez vos projets”



Objectif

“Que vous testiez vos projets”
(systématiquement)



Objectif

“Que vous testiez vos projets”

(systématiquement)
(ou presque)



Objectif

“Que vous testiez vos projets”

(systématiquement)

(ou presque)

(en vrai c'est fun)



Code
Garage

Introduction

Pré-requis





Pré-requis

- Aucun



Pré-requis

- Aucun
- Savoir coder (un peu)



Qu'est-ce qu'un logiciel ?

Définition : “Un logiciel, est un ensemble d’instructions logiques (code), qui résout une problématique selon des spécifications.”



Qu'est-ce qu'un logiciel ?

Définition : “Un logiciel, est un ensemble d’instructions logiques (code), qui résout une problématique selon des spécifications.”

Faire du code != faire un logiciel



Qu'est-ce qu'un test ?

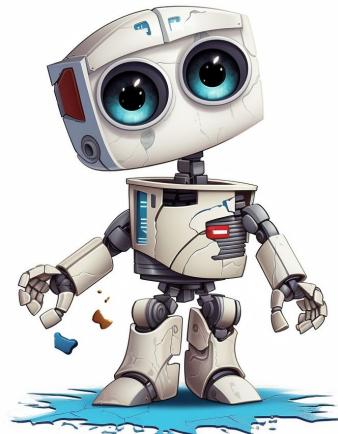
C'est un logiciel qui teste un logiciel.



Qu'est-ce qu'un test ?

C'est un logiciel qui teste un logiciel.

Spoiler alert : Le logiciel parfait n'existe pas, donc le test parfait n'existe pas non plus...





Pourquoi tester ?

Si aucun test n'est parfait ? Pourquoi écrire des tests ?

- Pas de test logiciel ➡️ Test humains
- Les humains sont :
 - Fatigables
 - Lents
 - Faillibles
 - Imprévisibles
 - Décevables

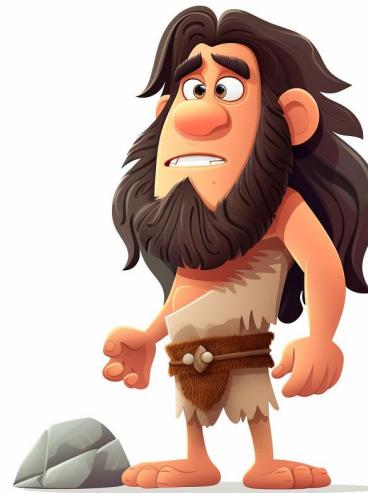


Pourquoi tester ?

Si aucun test n'est parfait ? Pourquoi écrire des tests ?

- Pas de test logiciel ➡️ Test humains
- Les humains sont :
 - Fatigables
 - Lents
 - Faillibles
 - Imprévisibles
 - Décevables

Mais on les aime quand même (pas tous)





Tester c'est douter ?

En 1945, Alan Turing a dit :

“L'un de nos problèmes sera le maintien d'une discipline suffisante, pour que nous ne perdions pas l'objectif que nous poursuivions.”



Peu importe que vous soyez le ou la meilleur(e) ingénieur du monde, vous ferez des erreurs, et d'autres en payeront les conséquences !



Une sonde “spéciale” (1998)

- NASA
- Mars Climate Orbiter
- Problème de distance
- Système impérial dans la sonde
- Système international dans la station



“Ce n'est pas un échec, mais ça n'a pas marché” - Chef de l'exploration du système solaire après le crash



Un “Excel” de zèle (2016)

- Microsoft
- Tableur Excel
- Recherche en biologie
- MARCH1 (Membrane Associated Ring-CH-Type Finger 1)
- 1er Mars



20% des papiers scientifiques en biologie affectés par cette “fonctionnalité”



Le vers et la pomme (2015)

- Apple
- Iphone SMS
- effective. Power لـ الصـبـلـاصـبـرـ
- Crash du téléphone
- Remise à zéro



Découvert en Mai et corrigé le 30 Juin 2015



Un peu de philosophie

- “Si vous n'aimez pas tester votre produit, il est certain que vos clients n'aimeront pas ça non plus.”



Un peu de philosophie

- “Si vous n'aimez pas tester votre produit, il est certain que vos clients n'aimeront pas ça non plus.”
- “Ce n'est pas parce que vous avez compté tous les arbres que vous avez vu la forêt”



Un peu de philosophie

- “Si vous n'aimez pas tester votre produit, il est certain que vos clients n'aimeront pas ça non plus.”
- “Ce n'est pas parce que vous avez compté tous les arbres que vous avez vu la forêt”
- “Le test logiciel prouve l'existence de bug, par leur absence”



Code
Garage



La vie d'un logiciel

La gestion de projet



Les différentes étapes

- L'analyse des besoins / spécifications
- La gestion de planning / ressources
- Le développement
- La vérification
- Le feedback
- La correction
- Le déploiement

Peu importe la méthode (cycle en V, en Y, méthode agile), ces étapes existent



Les différentes étapes

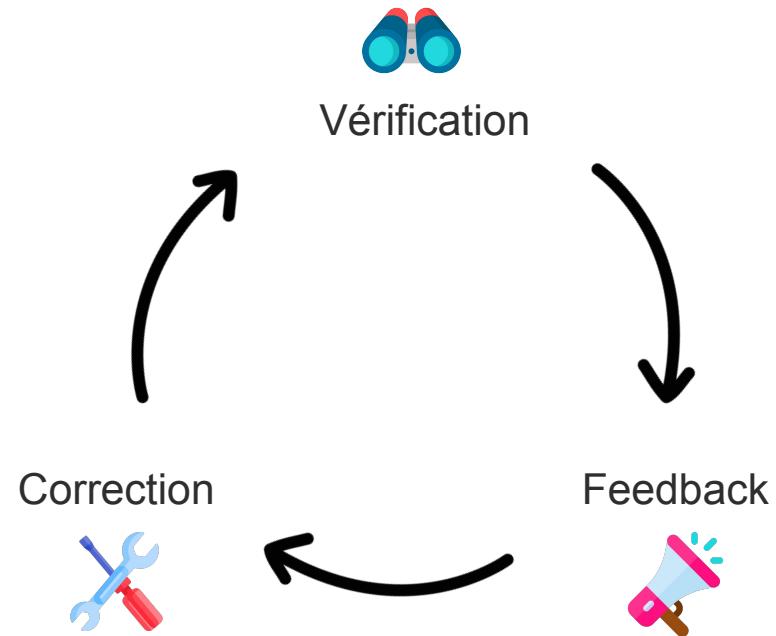
- L'analyse des besoins / spécifications
- La gestion de planning / ressources
- Le développement
- **La vérification**
- **Le feedback**
- **La correction**
- Le déploiement



Peu importe la méthode (cycle en V, en Y, méthode agile), ces étapes existent



Boucle de feedback

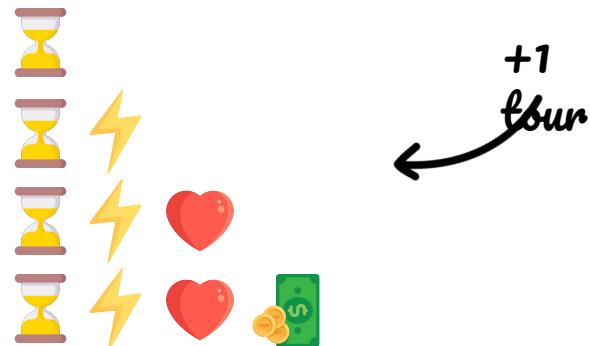




Impact de la boucle de feedback

À chaque tour de boucle :

- Dev
- Product owner
- Client.e
- Utilisateurs





Évolution du code

On a peur de :

- Renommer du code
- Changer une dépendance
- Restructurer des données
- Optimiser un algorithme



Ça pourrait changer le comportement de notre code



Évolution du code

En plus on va :

- Devoir tout retester à la main
- Perdre du temps
- Faire des erreurs
- Privilégier le “happy path”

Avec des tests, si la sortie du code correspond à ce que l'on attend, c'est bon !



Code

Garage



Les tests

Familles, types et rôles



Les 3 grandes familles

- **Les tests fonctionnels**
- Les tests de performance
- Les tests de sécurité



| Ce sont les tests fonctionnels qui nous intéressent aujourd'hui



Les tests fonctionnels

- **Les tests unitaires**
- Les tests d'intégration
- Les tests end-to-end (bout en bout)



Les tests unitaires

- Unité = fonction, méthode ou classe
- On teste la logique
- En isolation

C'est la pierre angulaire du test





Les tests d'intégration

- 2 composants ou plus testés en même temps
- Implémentation, architecture et fonctionnement
- Exemples :
 - Sauvegarde en base de données/fichiers
 - Envoi d'un email
 - Montage d'un composant front-end
 - ...





Les tests end-to-end

- Simuler l'usage d'un utilisateur et vérifier le comportement du logiciel
- Implémentation, architecture et fonctionnement
- Gros effort de préparation/configuration des outils
- Exemples :
 - Requête HTTP sur une API Rest
 - Appuis boutons sur interface
 - ...

Sur des systèmes critiques, ils sont indispensables !





Qu'est-ce qu'un test de non-régression ?

- C'est le “rôle” d'un test
- Peut être unitaire, intégration ou e2e
- Tester un morceau de code **déjà fonctionnel**
- Vérifier qu'il **continuera de fonctionner** même après une évolution

En réalité, un test de non-régression est simplement un test fonctionnel.



Code

Garage



Les tests unitaires

Outils et méthodes



Mon premier test

Direction : L'IDE !



Framework de test

- Permet d'exécuter le code en dehors de votre application
- Chaque fonction/classe est Importée, isolée du reste et exécutée

En JavaScript on utilisera **Jest**, et **NUnit** (intégré) avec Unity !

Pour rappel : un framework appelle le code du dev, alors que, c'est le dev qui fait appel au code de la bibliothèque



Framework de test

Il va en général :

- Chercher tous les fichiers de tests
- Les lancer de manière séquentielle
- Exécuter chaque test
- Afficher les résultats des tests individuellement
- Faire un récapitulatif
- Retourner une valeur selon si les tests sont tous passés avec succès ou non





Framework de test

```
GET /
  ✓ the status code should be 200
  ✓ the content type should be JSON
  ✓ the response should be a welcome message
```

```
GET /users
  ✓ the status code should be 200 (70ms)
  ✓ the content type should be JSON
  ✓ the response should contain 1 user
  ✓ the id of the user should 1
  ✓ the username of the user should john
```

8 passing (140ms)

Le résultat





Bibliothèque d'assertion

- Une règle dans les tests logiciels :
 - Le résultat d'un test doit être booléen
 - Vrai ou Faux (ou Exception)

```
return user.age === 18;
```

← Sans





Bibliothèque d'assertion

- Faciliter la gestion des conditions, et rendre la lecture et l'écriture des tests plus naturelle



```
return user.age.is.equal.to(18);
```

←
Avec





Les jeux de tests

Le “Happy Path”, c'est le chemin pris dans le code lorsque notre logiciel s'exécute correctement, et que l'utilisateur fait ce qu'on attend de lui.

Mais l'utilisateur peut être parfois créatif, maladroit, ou carrément malveillant !

On prépare donc plusieurs jeux de tests

- Le domaine valide
- Le domaine invalide



Le domaine valide

Le domaine valide représente l'ensemble des données que l'utilisateur lambda est censé pouvoir utiliser.

Au minimum un test pour le domaine valide

Exemple : Une fonction attend une chaîne de caractères, et on envoie “hello”.



Le domaine invalide

Le domaine invalide représente toutes les données que chaque fonction peut accepter, mais qui vont poser un problème.

En général, le domaine invalide va renvoyer null, undefined ou une exception

Exemple : Une fonction attend un email, et on envoie “hello.com”.



Tester les limites

- Pour les domaines valide et invalide
- Tester les valeurs limites
- Déetecter les potentiels comportements inattendus

On parle également de “**edge cases**”

On peut tester une infinité de valeurs limites, mais il faut **faire des choix raisonnables**, en fonction du code.



Tester les limites

Voilà un exemple :



```
function divide (a: number, b: number) {  
    return a/b;  
}
```

Que se passe-t-il si $a = 2$ et :

- $b = 0$
- $b = 0.1$
- $b = 1$
- $b = \text{Math.Infinity}$
- $b = \text{NaN}$



Les stubs

Stub (morceau) : objet ou une fonction fictive utilisée pour simuler le comportement d'une dépendance externe d'un composant à tester.

- Renvoie des **valeurs prédéfinies**
- Permet d'**isoler le composant**
- Vérifie son fonctionnement indépendant

Ils sont donc nécessaires pour respecter le concept d'isolation.



Les stubs : exemple

Utilisés lorsque l'accès à une dépendance externe n'est pas possible ou complexifie trop le test.

```
● ● ●  
  
class PaymentProcessor {  
    private paymentService: PaymentService;  
  
    constructor(paymentService: PaymentService) {  
        this.paymentService = paymentService;  
    }  
  
    processPayment(amount: number): boolean {  
        if(amount < 0){  
            throw new Error("cannot process payment for negative amount");  
        }  
        if(amount > 0){  
            try {  
                this.paymentService.charge(amount);  
            } catch(e){  
                return false;  
            }  
        }  
        return true;  
    }  
}
```



Création d'un stub

```
class PaymentServiceStub {  
    charge(amount: number): string {  
        // Simuler une réponse réussie du service de paiement  
        return true;  
    }  
}
```



Utilisation d'un stub



```
describe('PaymentProcessor', () => {
  it('should process a payment successfully', () => {
    const paymentServiceStub = new PaymentServiceStub();
    const paymentProcessor = new PaymentProcessor(paymentServiceStub);
    const result = paymentProcessor.processPayment(100);

    expect(result).toBe(true);
  });
});
```



Différences entre un stuck et un mock

- Stub : Un objet qui fournit des réponses prédéfinies aux appels des méthodes





Différences entre un stub et un mock

- Stub : Un objet qui fournit des réponses prédéfinies aux appels des méthodes
- Mock : Un objet qui simule la logique de la méthode d'origine mais de manière simplifiée





Code

Garage



Les bonnes pratiques

À ne pas oublier



La méthode F.I.R.S.T

Un ensemble de bonnes pratiques pour avoir des tests unitaires utiles et efficaces !

- Fast
- Isolated
- Repeatable
- Self-validating
- Timely





F.I.R.S.T : Fast

Si votre suite de test met trop longtemps à s'exécuter, alors vous n'aurez pas envie de la lancer.

Et des tests qui ne sont pas lancés ne servent à rien

On va d'ailleurs chercher à automatiser ces tests, l'objectif est de gagner du temps, pas d'en perdre



F.I.R.S.T : Isolated

C'est le principe même des tests unitaires !

Mais ça signifie aussi :

- Éviter les effets de bords
- Besoin de refactoriser si votre code n'est pas suffisamment isolé



F.I.R.S.T : Repeatable

Votre suite de test doit pouvoir :

- Être lancée plusieurs fois de tests
- Ne doit laisser aucune trace de données résiduelles
- Ne pas entraver une prochaine exécution



F.I.R.S.T : Self Validating

Chaque test doit :

- Fournir une sortie booléenne (assertion)
- Déterminer automatiquement si le système en cours se comporte correctement
- Pas d'intervention manuelle ou d'interprétation



F.I.R.S.T : Timely (opportun)

Chaque test doit être écrit :

- Rapidement
- En amont ou en parallèle du code (dans le meilleur des cas)
- On écrit un test en fonction des spécifications, pas du code



Préparer son code

Il est difficile de tester un code qui ne suit pas les bonnes pratiques de développement, notamment dans 4 cas particuliers :

- **Trop de logiques** différentes dans une même fonction
- Si une fonction ne prend **pas de paramètre**
- Si une fonction **ne renvoie pas de valeur**, et ne gère pas d'exceptions.
- Si une fonction prend **des paramètres trop complexes**, où difficilement prévisibles



Éviter les effets de bords

Un effet de bord, c'est lorsque qu'une méthode modifie la valeur d'une donnée, ou exécute une action, qui est **en dehors de ses préoccupations d'origine.**



```
function trackOrderInAnalytics(order: Order){  
    analytics.track('order-placed', order.id);  
    order.status = 'completed';  
}
```



La couverture de code

La couverture de code, c'est la représentation, en pourcentage, de **la proportion de code exécuté lors des tests.**

C'est la seule métrique objective que l'on possède lorsque l'on veut évaluer des tests

Même si ce n'est **pas une métrique parfaite.**





La couverture de code

Voici un exemple de rapport de “coverage” généré par la librairie JS appelée Istanbul :

- tous les tests sont passés avec succès
- la couverture de code est plutôt bonne
- certains fichiers restent à tester

File	%Stmts	%Branch	%Funcs	%Lines	Uncovered Line #s
All files	80.65	64	53.85	80.65	
ffw-api	70	16.67	0	70	
app.js	70	16.67	0	70	13,24,30,31,34,35
ffw-api/controllers	50	100	33.33	50	
users.js	50	100	33.33	50	5,9,10,16
ffw-api/models	91.3	78.95	100	91.3	
index.js	90	78.95	100	90	10,36
user.js	100	100	100	100	
ffw-api/routes	100	100	100	100	
index.js	100	100	100	100	
users.js	100	100	100	100	



La couverture de code

- Avoir un “Code Coverage” de 100% **ne signifie pas que les tests sont bien écrits**
- Ni que le code est robuste
- Mais cela permet d'avoir une vision sur des parties du code avec des tests restants à rédiger



Code

Garage



Autour des tests

Automatisation & Coverage



L'automatisation

Il y a deux sortes d'automatisation :

- Le lancement en local
- La CI/CD





Automatisation locale

Pour automatiser le lancement des tests unitaires avant un commit, il est **possible d'utiliser un “hook” git.**

Un hook est un évènement git auquel on va pouvoir attacher un script.

Par exemple on peut **exécuter un script avant un commit**, après un commit, avant un push, avant un pull, etc...

Ici notre objectif va plutôt être d'exécuter nos tests juste avant de commit.



Husky (Git Hook)

Le problème avec les hooks git, c'est que **leur configuration est locale** et ne peut pas être partagée en même temps que la base de code.

Husky est un outil qui permet de **gérer les hooks git directement depuis le fichier package.json**, et d'exécuter des commandes bash, npm, etc...





Husky (Exemple)

```
● ● ●  
{  
  "name": "mon-projet",  
  "version": "1.0.0",  
  "scripts": {  
    "test": "jest"  
  },  
  "husky": {  
    "hooks": {  
      "pre-commit": "npm test"  
    }  
  },  
  "devDependencies": {  
    "husky": "^7.0.4",  
    "jest": "^27.4.1"  
  }  
}
```



Github Action

- L'outil d'automatisation de GitHub
- CI/CD
- Permet de lancer ses tests sur un serveur
- Fonctionne également pour les tests d'intégration ou end-to-end



Exemple de workflow

```
name: Tests

on: [push]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: 14

      - name: Install dependencies
        run: npm test
```



TDD : Test-Driven Development

- Approche de développement logiciel où les tests unitaires sont écrits avant même d'écrire le code de production.
- Plusieurs étapes :
 - écrire un test
 - voir le test échouer
 - écrire le code minimal pour faire passer le test
 - voir le test réussir
 - puis refactoriser le code si nécessaire



TDD : Les avantages

- réduire les défauts dès le début du processus de développement, **d'obtenir une couverture de tests plus complète**
- faciliter la collaboration entre les membres de l'équipe
- **d'améliorer la confiance** dans la qualité du code produit



Test-first vs Test-last

Faut-il tester avant ou après ?

Il y a une très forte croyance sur le fait que le TDD aurait un impact positif sur la robustesse d'un logiciel (bien plus que le "*test last*").

En réalité, il y a eu des recherches relativement poussées sur le sujet, et **les deux méthodes se valent**, ce qui importe, c'est d'écrire des tests et de les lancer de manière automatique.

L'un des facteurs clé quant au nombre de bugs décelé dans un code réside plutôt dans **la complexité sociale de l'équipe de développement** (à nombre de lignes de codes écrites égales, il y aura plus d'anomalie si la taille de l'équipe augmente).



Générer ses tests avec ChatGPT

- La plupart des devs n'aiment pas écrire des tests
- **Bonne nouvelle :**
 - Plus forcément obligé de les écrire (ou presque)
 - on peut les générer





La question magique

“Generate a unit test using **[testing framework]** and **[assertion library]** for the following **[language]** function, including valid AND invalid input data.

Additional guideline : The test can fail if you think it should because the tested method is not written correctly. If so, please add your refactor recommendation in a comment above the failing test.

Here is the function : **[code]**”



La question magique



```
export function calculateDiscountedPrice(price: number, discount: number): number {
    if (price <= 0) {
        throw new Error('Invalid price');
    }

    if (discount < 0 || discount > 100) {
        throw new Error('Invalid discount percentage');
    }

    const discountAmount = (price * discount) / 100;
    const discountedPrice = price - discountAmount;

    return discountedPrice;
}
```



La réponse magique

```
import { calculateDiscountedPrice } from './your-module';

describe('calculateDiscountedPrice', () => {
  // Valid inputs
  test('returns discounted price for valid inputs', () => {
    const price = 100;
    const discount = 20;
    const expected = 80;

    const result = calculateDiscountedPrice(price, discount);

    expect(result).toEqual(expected);
  });
  //...
});
```



La réponse magique

```
// Invalid price
test('throws error for zero price', () => {
  const price = 0;
  const discount = 20;

  expect(() => {
    calculateDiscountedPrice(price, discount);
  }).toThrow('Invalid price');
});

test('throws error for negative price', () => {
  const price = -100;
  const discount = 20;

  expect(() => {
    calculateDiscountedPrice(price, discount);
  }).toThrow('Invalid price');
};

//...
```



La réponse magique



```
// Add more test cases as needed
// Uncomment the following test case if you think the function should be
// refactored to handle decimal discounts

test('returns discounted price for decimal discount percentage', () => {
  const price = 100;
  const discount = 10.5;
  const expected = 89.5;

  const result = calculateDiscountedPrice(price, discount);

  expect(result).toEqual(expected);
});
```



Code

Garage



Mise en pratique

Exemples et cas concrets



Dépôts Github

- JS : <https://github.com/CodeGarageFrance/unit-testing-js-basic>
- Unity : <https://github.com/CodeGarageFrance/unit-testing-unity-basic>

Ce sont les mêmes exercices, dans deux langages différents, plus un exemple supplémentaire pour les contrôleurs avec Unity.



Guide Unity

Lancer les tests :

- Window > General > Test Runner
- Cliquer sur “PlayMode”
- Puis sur “Run All”

Code Coverage :

- Window > Analysis > Code Coverage
- Cliquer sur “Enable”