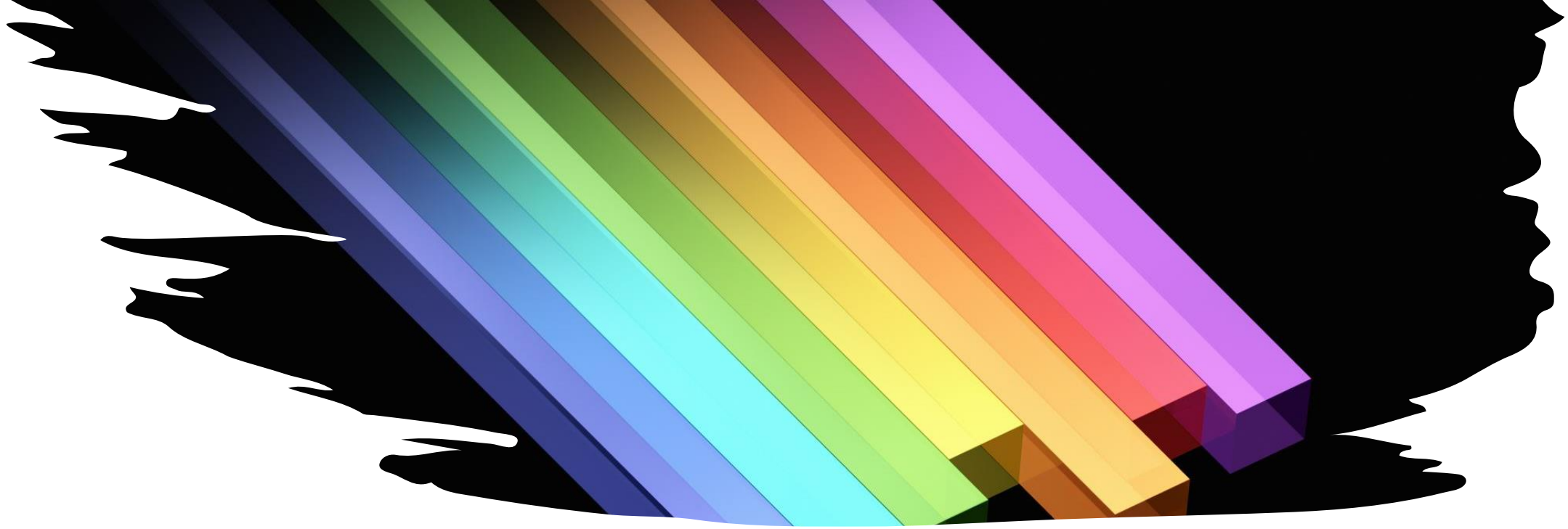


Tests unitaires Java avec JUnit 5

DÉVELOPPEMENT PILOTÉ PAR LES TESTS AVEC JUNIT 5

Généralités

L'informatique : domaine d'activité industriel ?

- À de nombreux égards, l'informatique s'inspire de l'industrie : mise en place de processus standardisés, structuration des équipes, notion de produit livré, etc. Il convient d'aller au bout de la démarche, et de considérer, aussi, les problématiques de qualité.

Qu'est-ce qu'un programme qui marche, au juste ?

- Un programme qui « marche » est un programme **qui produit les résultats attendus en fonction des données qui lui sont fournies** en entrée.

- Mais comment le vérifier, sachant qu'un programme est en fait constitué de plusieurs programmes, eux-mêmes constitués d'autres programmes, etc. : que doit-on tester, le programme principal ou les sous-programmes ? La réponse à cette question fait débat depuis très longtemps, mais des éléments de réponse se situent dans l'étude des procédés industriels !

Industriel et industrialisation…

Au delà du fait d'écrire des lignes de code et d'automatiser des traitements, les principaux acteurs du domaine de l'informatique ont œuvré, depuis des décennies, à en faire un domaine d'activité industriel.

- Ils en ont repris les objectifs : fournir des produits, à des clients, dans le but de conquérir des marchés.

- Ils en ont repris les (ou plutôt certaines des) méthodes : MOE/MOA, qualité, architecture, gestion de projets, etc.

Figure – Pont de l'île de Ré, testé et approuvé (avant de faire monter des gens dessus…)

De la qualité intrinsèque des composants d'un produit industriel…

- **Très important** – *Pour réaliser le pont de l'île de Ré, chaque pièce importante a été mise à l'épreuve individuellement pour garantir qu'elle remplisse sa propre fonction au sein de l'ouvrage final, sans défaillance. Puis, des tests ont été réalisés sur des parties assemblées de plus grande taille, et enfin sur le pont terminé.*

- Le bâtiment c'est le bâtiment, l'informatique, c'est l'informatique. Mais le parallèle est simple : si vous avez compris et admis de qui est mentionné précédemment, vous avez déjà intégré la notion de test **unitaire**, de test **d'intégration**, de **recette**…

La qualité logicielle

Qualité logicielle : définition

Difficile de définir précisément et de manière consensuelle la notion de qualité logicielle, tellement les indicateurs mis en jeu sont nombreux et variés. Deux approche fortes peuvent être identifiées :

- La **qualité du produit**, autrement dit la fin, le résultat produit.

- La **qualité des processus**, autrement dit les moyens, la façon dont le produit est conçu et créé.

Ces deux axes sont complémentaires, et donnent lieu à différents actions, normalisations, indicateurs de suivi, etc.

Principaux facteurs ou indicateurs de qualité

Parmi les principaux indicateurs de qualité, on peut notamment cite : la **fiabilité** (*qui peut être caractérisée par la robustesse du logiciel et l'exactitude des résultats produits*), la **performance**, la **tolérance aux pannes**, la **complétude** des fonctionnalités…

Tests techniques

Cinématique de test

En matière de testing, beaucoup de monde s'excite avec ce que l'on appelle les « tests fonctionnels » ou encore (souvent à tort) la « recette ». On parle aussi parfois de « tests end to end » ou « tests de bout en bout » pour qualifier les tests de l'ensemble des intégrations nécessaires pour la mise en œuvre d'un service logiciel répondant à une problématique fonctionnelle (*commande d'un article sur un site marchand, réservation d'un billet de train, etc.*).

En réalité, pour fournir un produit de qualité, un seul chemin est possible, exposé de manière schématique par la figure ci-suivante :

Figure – Processus de test : l'enchaînement recommandé

Important – Toutes les étapes décrites par la figure ci-dessus sont obligatoires (tests unitaires, tests d'intégration, tests de bout en bout) et l'ordre compte !

Tests unitaires

Définition : qu'est-ce qu'un test unitaire ?

Comme beaucoup de concepts en informatique, le concept de test unitaire n'est pas simple à définir. Non pas qu'il soit réellement complexe, mais plutôt parce que beaucoup de développeur se font leur propre idée de ce qu'est un test unitaire…

- Un **test unitaire** est un procédé permettant de **s'assurer du bon fonctionnement d'une unité de programme**.
- OK, cette définition est assez générale. Certes, mais il est vrai que le terme « **unitaire** » peut lui-même renvoyer à ces réalités assez différentes. Une unité de programme est-elle une fonction ? Une classe ? Une instruction ?

Au fond, à quoi sert un test unitaire ?

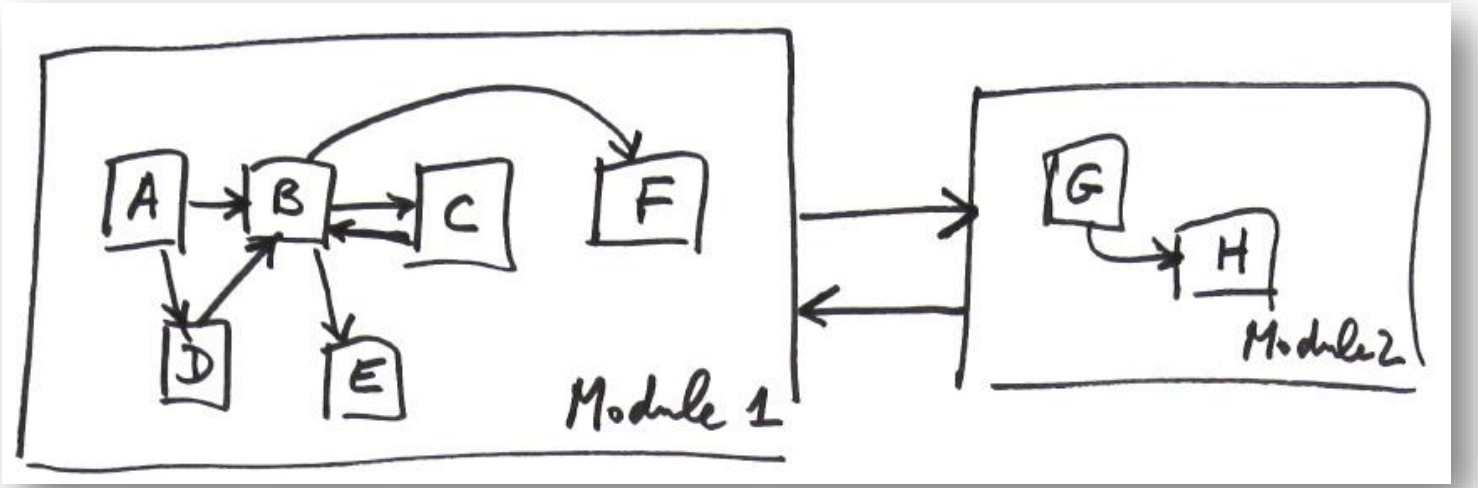
Au delà des définitions, une bonne manière de comprendre le testing unitaire est d'en comprendre le but, de façon pragmatique.

Il s'agit simplement de vérifier, en fonction de certaines données fournies en entrée d'un module de code (*on parle parfois d'unité fonctionnelle testée, ou SUT pour **S**ystem **U**nder **T**est*), que les données qui en sortent ou les actions qui en découlent sont conformes aux spécifications du module.

En d'autres termes, il s'agit de vérifier le respect du **contrat de service** du module.

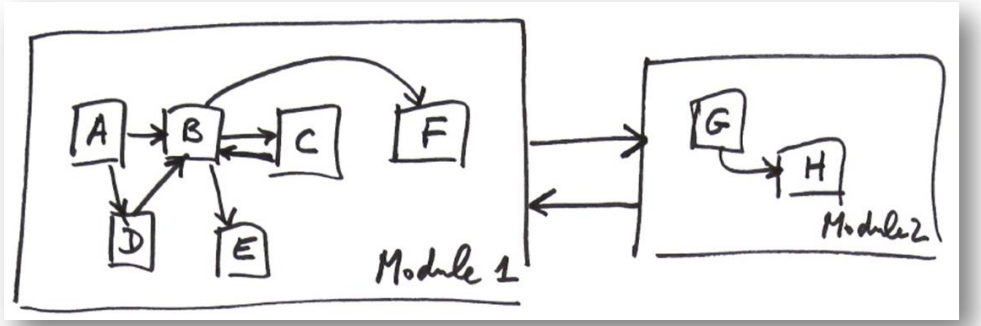
Testing unitaire : un exemple

Considérons un système informatique composé de deux gros modules interdépendants.



Dans chacun de ces modules, nous trouvons différents composants, eux-mêmes dépendants entre eux.

Testing unitaire : un exemple



Sur notre exemples,

- le module **D** accepte en entrée des données qui lui sont fournies par le composant **A**, et produit des données en résultat qui sont utilisées par le composant **B**. OK ?
- Imaginons maintenant que le développeur Kim soit responsable du développement et de la maintenance du composant **A**, que Brigitte soit responsable du composant **B**, et que Daniel soit responsable du composant **D**.

Respecter son contrat…

Si Arnaud fait mal son boulot en déployant un composant A buggué qui n'implémente pas correctement la spécification qui lui a été fournie, il y a fort à parier que le composant D du pauvre Daniel plantera lamentablement, même s'il est correctement développé.

C'est alors que commencent de longues et pénibles discussions, du genre « *Non, mais c'est ton code qui est faux, non c'est le tien, bla, bla, bla…* ».

Une solution simple pour s'assurer que chacun fait correctement son travail est tout simplement d'implémenter des tests unitaires sur chacun des composants..

Le cas de Daniel

Dans le cas de Daniel, il s'agit de s'assurer que si le composant **D** reçoit les données qu'il attend (i.e. conformes à la spec), alors il produit les résultats qu'il est censé produire (i.e. conformes à la spec). Ni plus, ni moins.

Les tests unitaires de **D** auront donc grosso-modo cette forme :

En d'autres termes : si je fais entrer le jeu de données x_1 dans la moulinette D, alors il sort y_1. Si je fais entrer le jeu de données x_2 dans la moulinette D, alors il sort y_2.

Conclusion : chacun respecte son contrat et tout le monde est content !

À retenir

Le testing unitaire repose au fond sur un principe très simple :

- un système aussi gros et complexe qu'il soit est toujours composé différentes parties plus petites que lui.

- Il est indispensable (mais pas suffisant) que chacune de ces parties remplisse correctement sa fonction pour que l'ensemble soit fonctionnel et fiable.

Test-Driven Development

Introduction

Vous souvenez-vous de la dernière fois où vous avez voulu apporter une modification pour corriger un défaut critique découvert par un client ? Étiez-vous certain que votre correctif n'introduirait pas d'erreur de régression ?

Pensez également à la dernière fois où vous avez voulu refactoriser votre code, mais vous craigniez que votre changement ne casse quelque chose d'autre ? Cela se produit souvent dans le développement de logiciels - la peur de casser un logiciel. Même les meilleurs programmeurs commettent des erreurs et introduisent des défauts.

Une solution simple pour s'assurer que chacun fait correctement son travail est tout simplement d'implémenter des tests unitaires sur chacun des composants.

Théorie

Le développement piloté par les tests, redécouvert par **Kent Beck** en 2003, est une pratique de développement qui augmente la confiance des développeurs en préconisant des tests pour toutes les exigences logicielles.

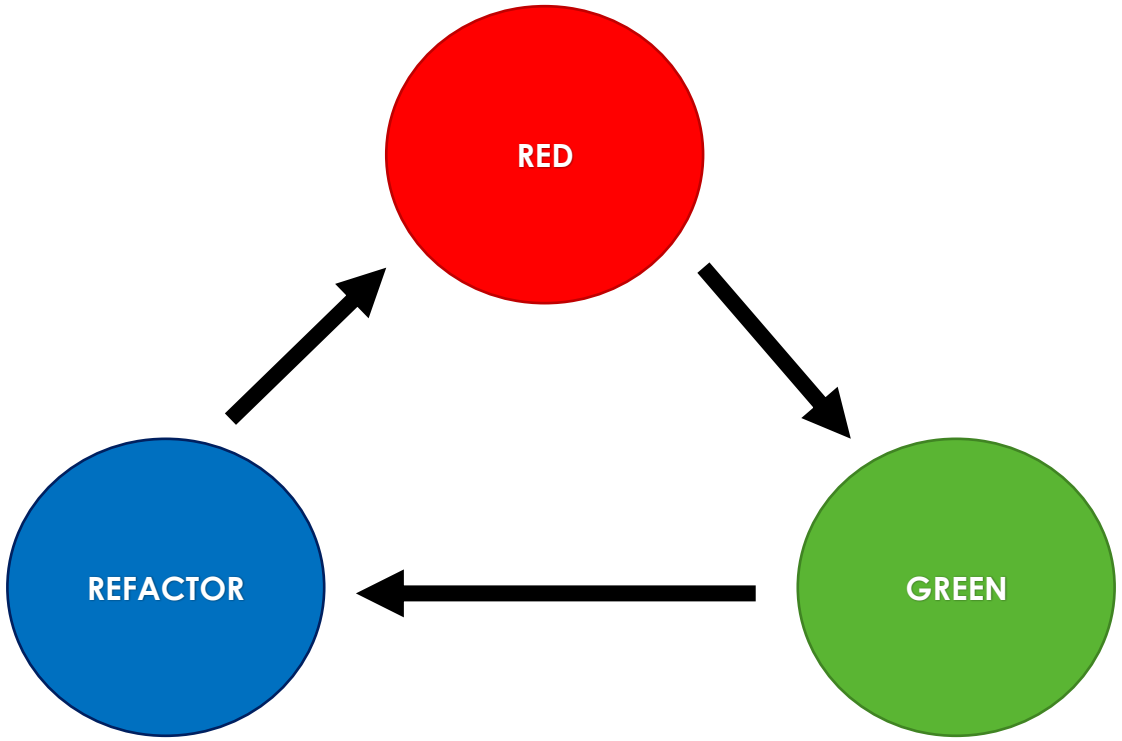
Cela nous fait travailler dans des cycles de développement incrémentaux courts (quelques minutes), fournissant ainsi un retour rapide sur nos progrès.

TDD nous oblige à écrire un test d'échec avant d'écrire le code de production. Le processus complet se présente comme suit :

Etapes

1. Ajoutez un test pour la nouvelle fonctionnalité ou le nouveau comportement.
2. Le voir échouer.
3. Écrivez suffisamment de code pour que le test réussisse.
4. Assurez-vous que tous les tests précédents réussissent également.
5. Refactorisez le code.
6. Répétez jusqu'à ce que vous ayez terminé.

Schématisation



Kata FizzBuzz

1. Ecrire un programme qui retourne sous forme de chaine de caractères les nombres de 1 à 100.
   a) *Les nombres multiples de 3 sont remplacés par « **Fizz** »*
   b) *Les nombres multiples de 5 sont remplacés par « **Buzz** »*
   c) *Tout nombre multiple de 15 est remplacé par « **FizzBuzz** »*

2. Exemple d'exécution.

   « *12Fizz4BuzzFizz78FizzBuzz … Buzz* »

Kata FooBarQix

1. Vous devez implémenter une fonction ***String*** **compute**(***String***) qui respecte les règles suivantes.
    1. **Étape 1**
        1. Si le nombre passé en paramètre est divisible par 3, on le remplace par « Foo » ,
        2. Si le nombre passé en paramètre est divisible par 5, on le remplace par « Bar »
        3. Si le nombre passé en paramètre est divisible par 7, on le remplace par « Qix »
        4. Pour chaque chiffre 3, 5, 7, ajoutez « Foo », « Bar », « Qix » dans l'ordre des chiffres.

Kata FooBarQix

1. Exemples
   - 1 ➔ 1
   - 2 ➔ 2
   - 3 ➔ FooFoo (divisible par 3, contient 3)
   - 4 ➔ 4
   - 5 ➔ BarBar (divisible par 5, contient 5)
   - 6 ➔ Foo (divisible par 3)
   - 7 ➔ QixQix (divisible par 7, contient 7)
   - 8 ➔ 8
   - 9 ➔ Foo
   - 10 ➔ Bar
   - 13 ➔ Foo
   - 15 ➔ FooBarBar (divisible par 3, divisible par 5, contient 5)
   - 21 ➔ FooQix
   - 33 ➔ FooFooFoo (divisible par 3, contient deux 3)
   - 51 ➔ FooBar
   - 53 ➔ BarFoo

Kata FooBarQix

Étape 2

Nous avons une nouvelle exigence métier : il faut garder une trace des 0 dans les chiffres, chaque 0 doit être remplacé par le caractère « * ».

Exemples

- 101 ➔ 1*1
- 303 ➔ FouFou*Fou
- 105 ➔ FooBarQix*Bar
- 10101 ➔ FooQix**

Kata Années bissextiles

Ce Kata court et simple doit être exécuté par paires en utilisant le Test Driven Development (TDD).

Avant 1582, le calendrier julien était largement utilisé et définissait les années bissextiles comme chaque année divisible par 4. Cependant, il a été constaté à la fin du XVIe siècle que l'année civile s'était éloignée de l'année solaire d'environ 10 jours. Le calendrier grégorien a été défini afin de réduire le nombre d'années bissextiles et d'aligner plus étroitement l'année civile sur l'année solaire. Il a été adopté dans les pays pontificaux le 15 octobre 1582, sautant 10 jours à partir de la date du calendrier julien. Les pays protestants ont adopté le calendrier grégorien après un certain temps.

Kata Années bissextiles

Le calendrier grégorien est assez précis, mais pourrait être rendu plus précis en ajoutant une règle supplémentaire qui élimine les années divisibles par 4000 comme années bissextiles. Mais je suppose que nous traverserons ce pont quand nous y arriverons. Envisagez d'ajouter cette règle en tant que deuxième « user story » en tant que prolongement de l'exercice.

User Story :

- *En tant qu'utilisateur, je souhaite savoir si une année est une année bissextile, afin de pouvoir prévoir une journée supplémentaire le 29 février au cours de ces années.*

Kata Années bissextiles

Critères d'acceptation *:*

1. *Toutes les années divisibles par 400* ***SONT*** *des années bissextiles (ainsi, par exemple, 2000 était bien une année bissextile),*
2. *Toutes les années divisibles par 100 mais pas par 400 ne sont* ***PAS*** *des années bissextiles (ainsi, par exemple, 1700, 1800 et 1900 n'étaient PAS des années bissextiles, NI 2100 ne sera pas une année bissextile),*
3. *Toutes les années divisibles par 4 mais pas par 100 SONT des années bissextiles (par exemple, 2008, 2012, 2016),*
4. *Toutes les années non divisibles par 4 ne sont PAS des années bissextiles (par exemple 2017, 2018, 2019).*

Kata String Calculator

https://codingdojo.org/kata/StringCalculator/