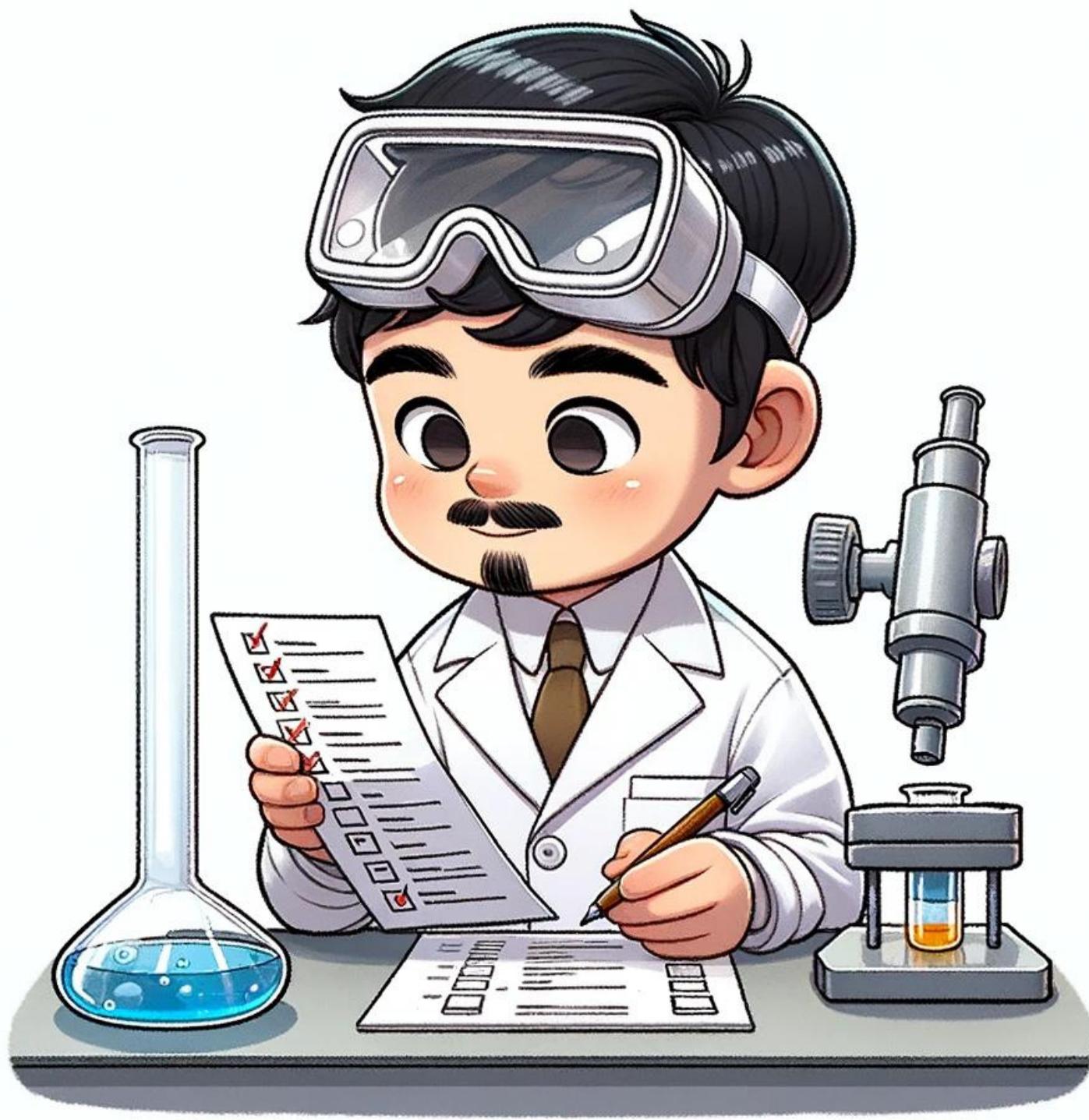


# ZOOM SUR



# LES TESTS



## DÉFINITION

Un test consiste à jouer une portion de code dans un contexte donné puis vérifier son comportement via des assertions.

Il permet de savoir si le code fonctionne tel qu'on le souhaite, c'est à dire tel que cela est spécifié dans le test. Ce retour est appelé par la suite “feedback”.



## BÉNÉFICES

Les tests me donnent un feedback rapide tout au long de mon développement sur le fait que mon code respecte l'ensemble de mes assertions. Ce qui engendre de la confiance pour délivrer du code et de la flexibilité car le filet de sécurité qu'ils constituent facilite la modification de code existant.

Ils me servent aussi de documentation vivante.

Ils m'aident à conserver une productivité haute.



## AUTOMATISÉS

Parce que je veux un grand nombre de tests et que je souhaite pouvoir les rejouer facilement et rapidement, il est primordial que ces tests soient automatisés.

Il m'arrive de faire des tests manuels pour explorer un comportement, mais dès que possible, je bascule sur l'écriture d'un test automatisé.



## F.I.R.S.T.

FIRST est un acronyme désignant cinq propriétés que je respecte lorsque je rédige un test :

Fast, Independant, Repeatable,  
Self-validating et Thorough

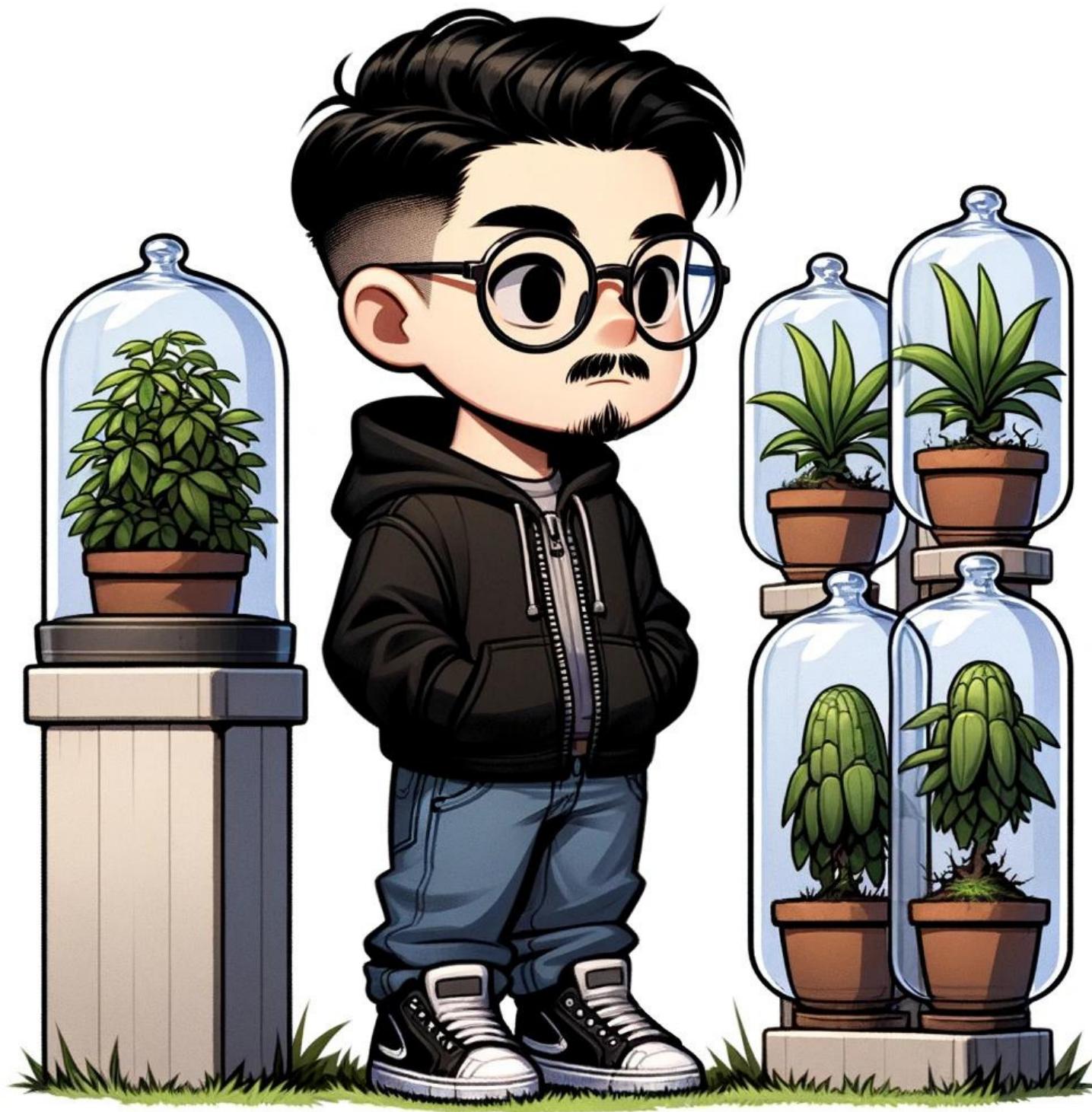


## FAST

Un test devrait être aussi rapide que possible.

A titre d'exemple, pour un test unitaire, je souhaite un feedback de l'ordre de la milliseconde.

Pour cela, j'évite les timers/sleep, je ne teste pas la même chose plusieurs fois, et j'évite de dépendre de l'infrastructure réelle (base de données, API tierce...).



## INDEPENDANT

Un test devrait être indépendant des autres tests et de l'environnement dans lequel on le lance.

Je m'assure d'isoler mes tests de manière à pouvoir les lancer en parallèle, sans que cela n'ai d'impact sur le résultat de chacun d'entre eux.



## REAPETABLE

Un test devrait être répétable.

On devrait pouvoir répéter un test autant de fois qu'on le souhaite sans que cela n'ai d'impact sur le résultat du test.



## SELF-VALIDATING

Un test devrait indiquer lui-même si il est en réussite ou en échec. Pour cela, je m'assure d'écrire au minimum une assertion par test.

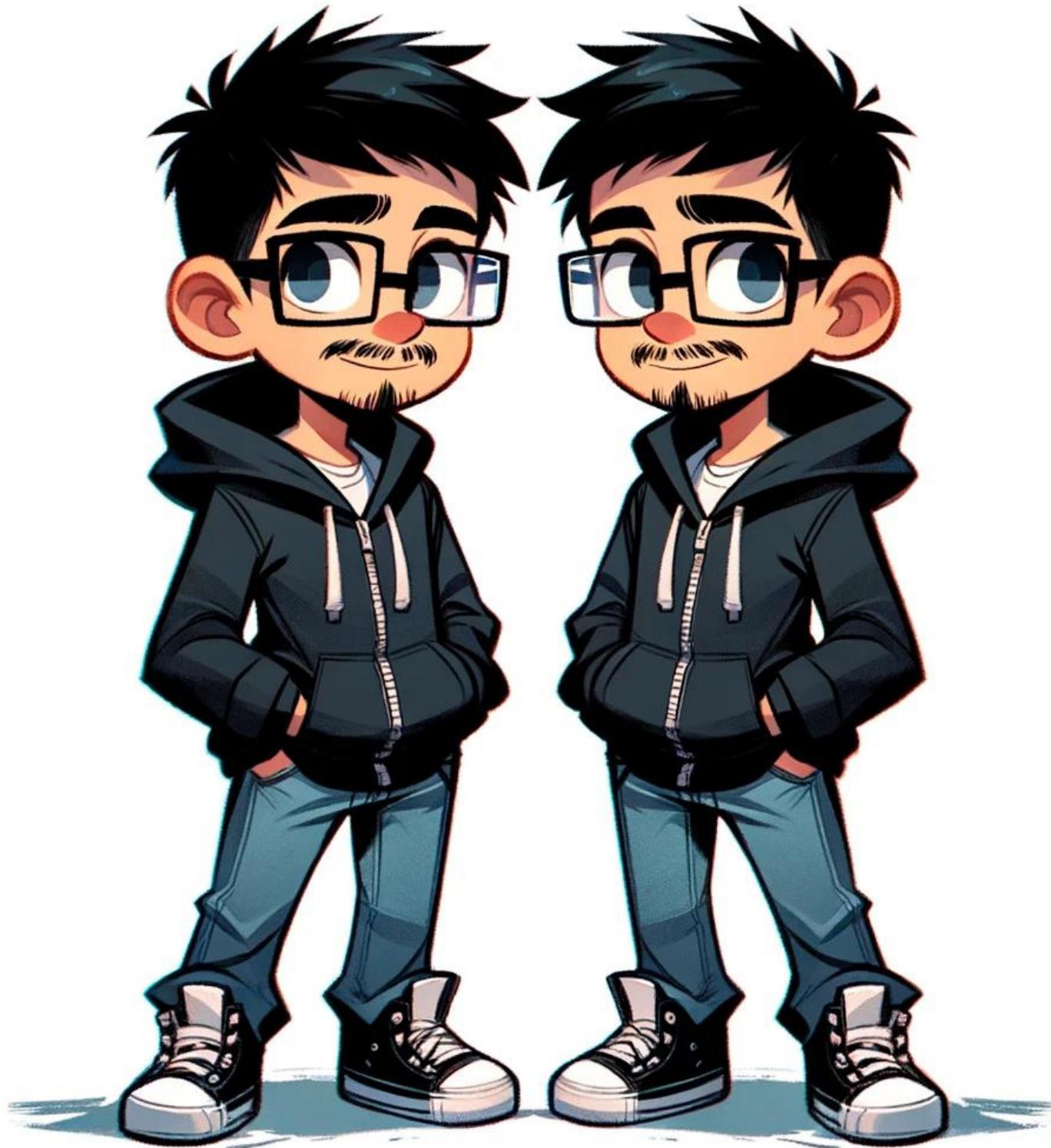
On ne devrait pas avoir à faire d'opération manuelle pour vérifier un test (par exemple regarder des logs ou le contenu d'une base de données).



## THOROUGH

**Les tests devraient couvrir une fonctionnalité dans son ensemble.**

**Je teste en premier lieu le cas nominal, mais aussi les cas aux limites, les cas d'erreur, les comportements inattendus (undefined / null) et les cas relatifs à la sécurité.**



## DOUBLURES DE TEST

Ce terme vient du cinéma où l'on utilise des doublures lors de scènes d'acrobatie dangereuses qui ne sont alors pas effectuées par les "vrais" acteurs.

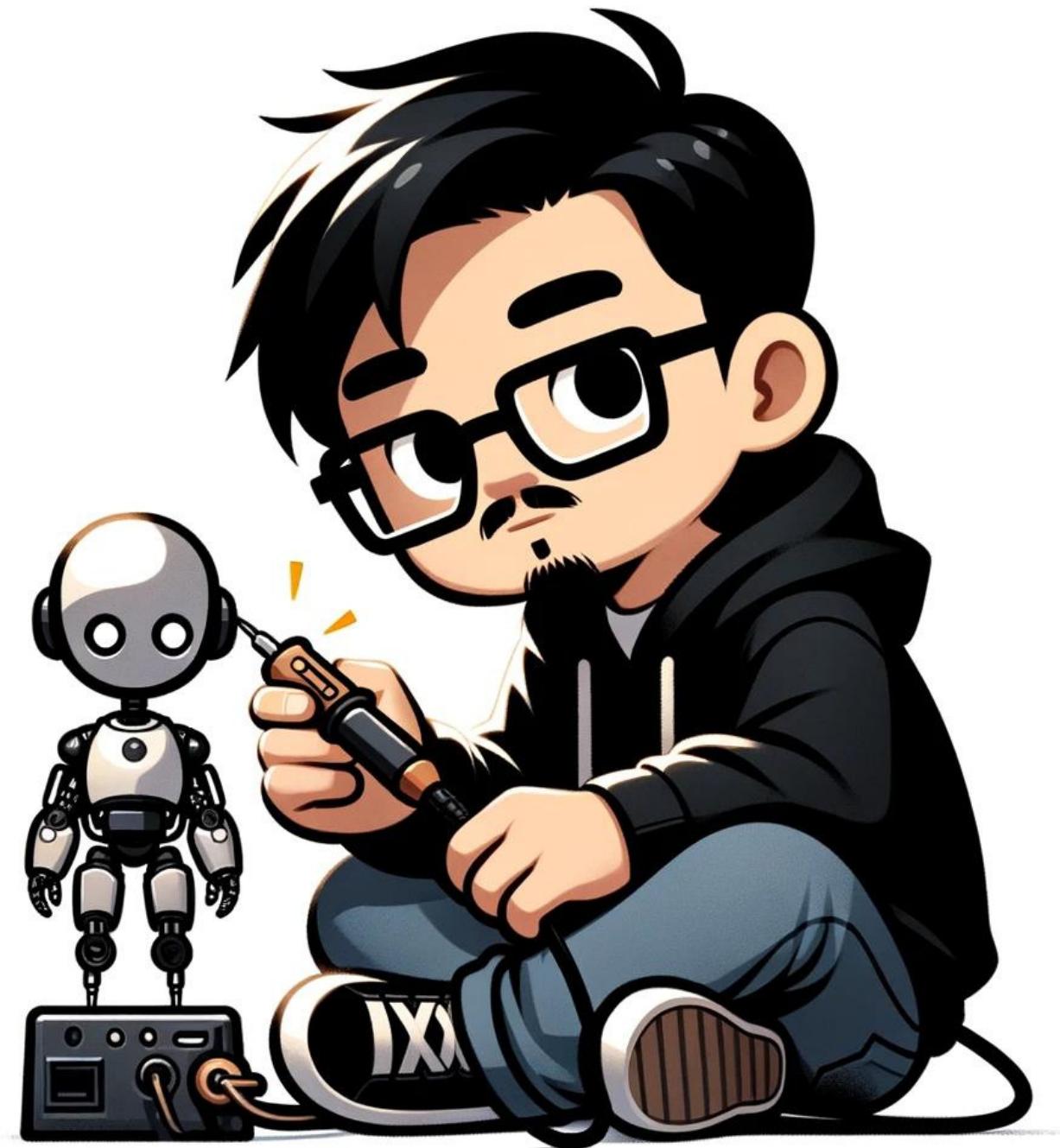
Dans le monde du dev, il existe plusieurs types de doublures de tests : dummy, stub, spy, mock et fake.



## DUMMY

Un dummy est une implémentation minimaliste d'une doublure de test.

Je l'utilise quand un objet est nécessaire pour créer un test mais que cet objet ne sera pas utilisé.



## STUB

Le stub est un dummy qui permet en plus de pouvoir définir le comportement de la doublure de test.

On peut indiquer le retour d'une fonction doublée, exécuter une portion de code à la place de la fonction originale ou encore lever des exceptions.

Il m'est très utile pour améliorer les performances de mes tests et réduire leur complexité.

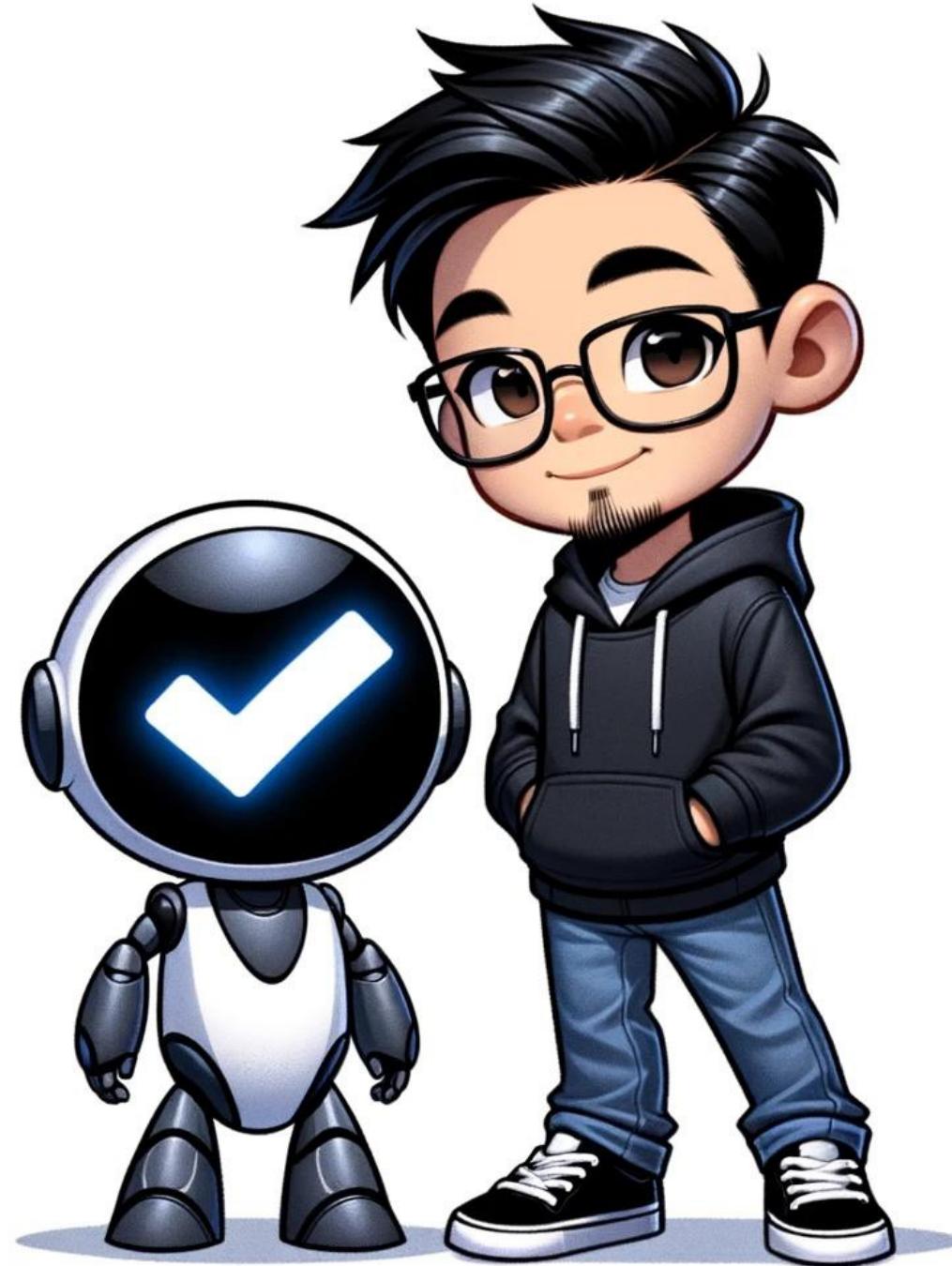


## SPY

**Le spy est un stub qui permet en plus d'observer l'utilisation de la doublure de test.**

**On peut savoir combien de fois une fonction doublée a été appelée et avec quels arguments.**

**Je m'en sers pour vérifier que j'agis correctement sur une autre couche applicative dont dépend le bout de code que je teste.**



## MOCK

Le mock est un spy qui produit implicitement des assertions sur les comportements qui lui ont été définis.

À la fin de l'exécution d'un test, si une fonction "mockée" n'a pas été appelée, le test sera en échec.

Je n'utilise que très rarement cette doublure car elle est un antipattern du GWT/AAA (voir plus loin).

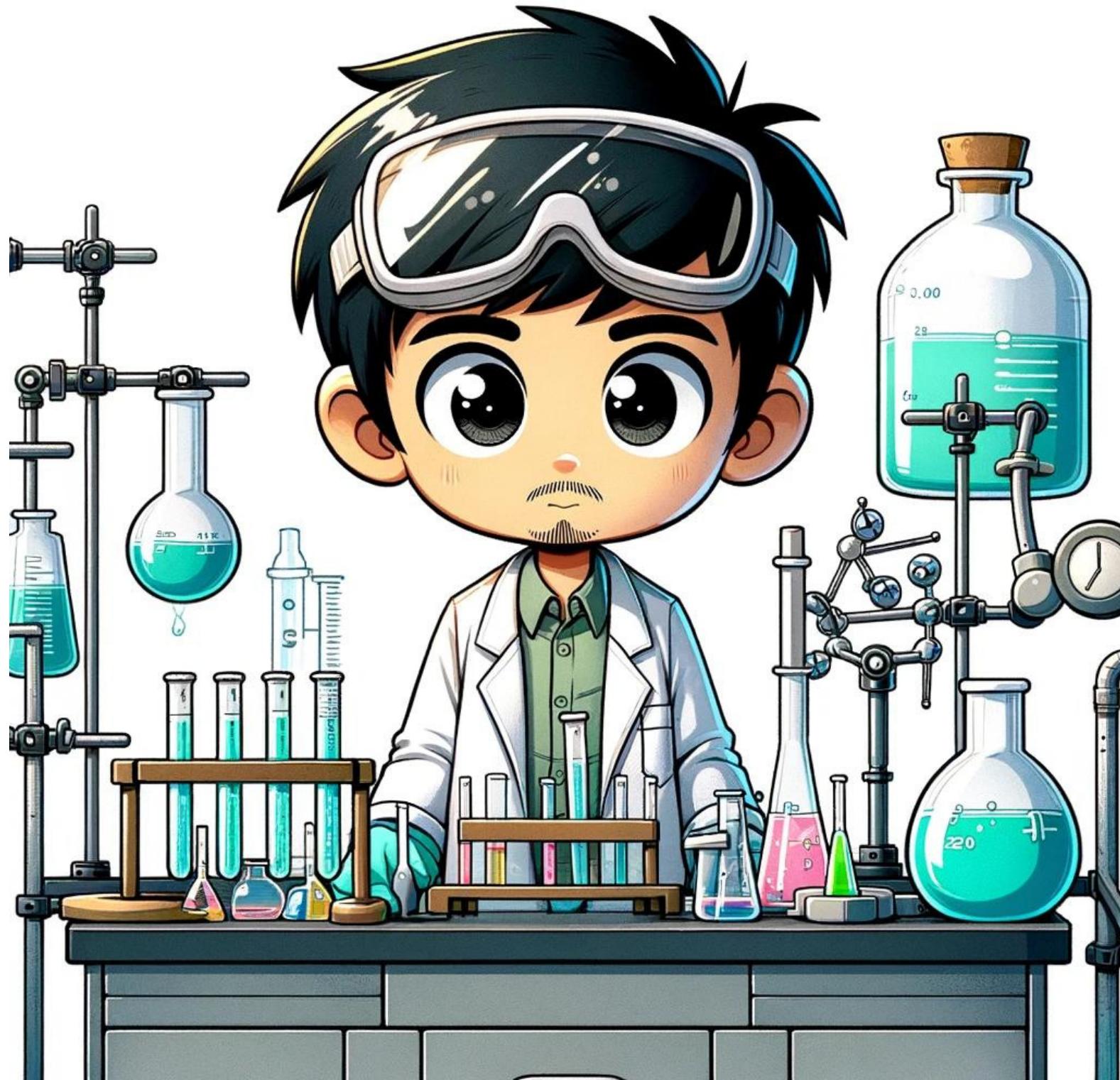


# FAKE

Implémentation manuelle d'une doublure de test.

Il s'agit de la première forme de doublure de test, avant l'apparition de ces nombreuses librairies qui nous offrent aujourd'hui les stubs, spies et mocks.

Le fake me permet de réaliser une implémentation très basique et rapide d'un composant lourd pour simplifier certains tests. Par exemple, je peux créer une base de données “in memory” plutôt qu’utiliser un vrai connecteur.



## TYPES DE TESTS

Selon ce qu'ils permettent de tester, il est possible de classer les tests par type ou par catégorie.

Attention : les nomsages qui vont suivre sont ceux que je suis habitué à employer. Mais ces noms peuvent être utilisés par d'autres personnes pour désigner d'autres types de tests.



## TESTS UNITAIRES

Les tests unitaires me permettent de tester une unité fonctionnelle. Ils n'ont pas de dépendance lourde et sont donc très rapides à exécuter.

Comme ils sont simple à écrire et qu'ils me donnent un feedback de l'ordre de la milliseconde, je vise à maximiser leur nombre. Il me permettent de détecter la majorité des problèmes rapidement.



## UNITÉ FONCTIONNELLE

Je teste le comportement d'une unité fonctionnelle.

Je n'écris pas nécessairement un test par fonction ou méthode publique.

De plus, mon test peut faire appel à plusieurs fonctions ou méthodes pour vérifier le comportement de cette unité fonctionnelle.



## TESTS TRAVERSANTS

J'extrais parfois une classe ou une fonction pendant un refactoring. Cette nouvelle entité ne nécessite pas de test dédié puisque son code est déjà couvert par un test existant, au travers du code dont elle est extraite.

En ne couplant pas mes tests aux détails d'implémentation, je me laisse la possibilité de refactorer le code sans risquer de devoir réécrire mes tests.



## COMPORTEMENT

Je m'assure de tester un comportement plutôt qu'une implémentation. Cela me permet d'avoir des tests plus solides.

Je n'ai pas besoin de savoir comment est/sera implémentée une fonctionnalité pour pouvoir rédiger un test.



## DÉPENDANCES

J'utilise des doublures de tests lorsque le code testé dépend d'implémentations lourdes qui ne sont pas l'objet du test ou lorsque je souhaite découpler le code de ces implémentations.



## USAGES ET OUTILS

Dans un contexte d'architecture hexagonale ou de clean architecture, les tests unitaires me permettent de valider le “domain” et les transformations effectuées dans les “adapters”.

Pour les patterns MVP et MVVM, ils me permettent de tester tout ce qui n'est pas de l'UI (la couche “V”).

Outils : JUnit, Vitest, Jest, Mockito, Sinon...



## TESTS D'INTÉGRATION

Les tests d'intégrations me permettent de vérifier les interactions avec du code externe à mon système, par exemple un connecteur de base de données, une librairie tierce ou l'appel à un service tiers.

Je ne teste pas ce code externe car je suppose que je peux lui faire confiance. Par contre, je teste que je l'utilise correctement dans le cadre de mes besoins.



## TESTS D'INTÉGRATION

Ils peuvent aussi me permettre de découvrir le fonctionnement de ce code externe avec une boucle de feedback bien plus courte que si je devais démarrer le produit dans son ensemble.

Ces tests peuvent être relativement coûteux, j'essaie donc de restreindre le code au strict minimum et de m'assurer de ne pas lier de logique métier ou d'adaptation.



## USAGES ET OUTILS

Dans un contexte d'architecture hexagonale ou de clean architecture, les tests d'intégration me permettent de valider tout appel direct à de “l'infrastructure”.

Outils : JUnit, Vitest, Jest, TestContainers...



## TESTS DE CONTRAT

Les tests de contrat sont utiles dès lors que j'expose une API publique.

Ils me permettent de m'assurer que je n'apporte pas de régression dans les contrats d'API définis lorsque je modifie la base de code.



## TESTS DE COMPOSANT

Les tests de composant me permettent, moyennant des cas d'usage, de tester l'application dans son ensemble mais isolée de tout lien vers l'extérieur.

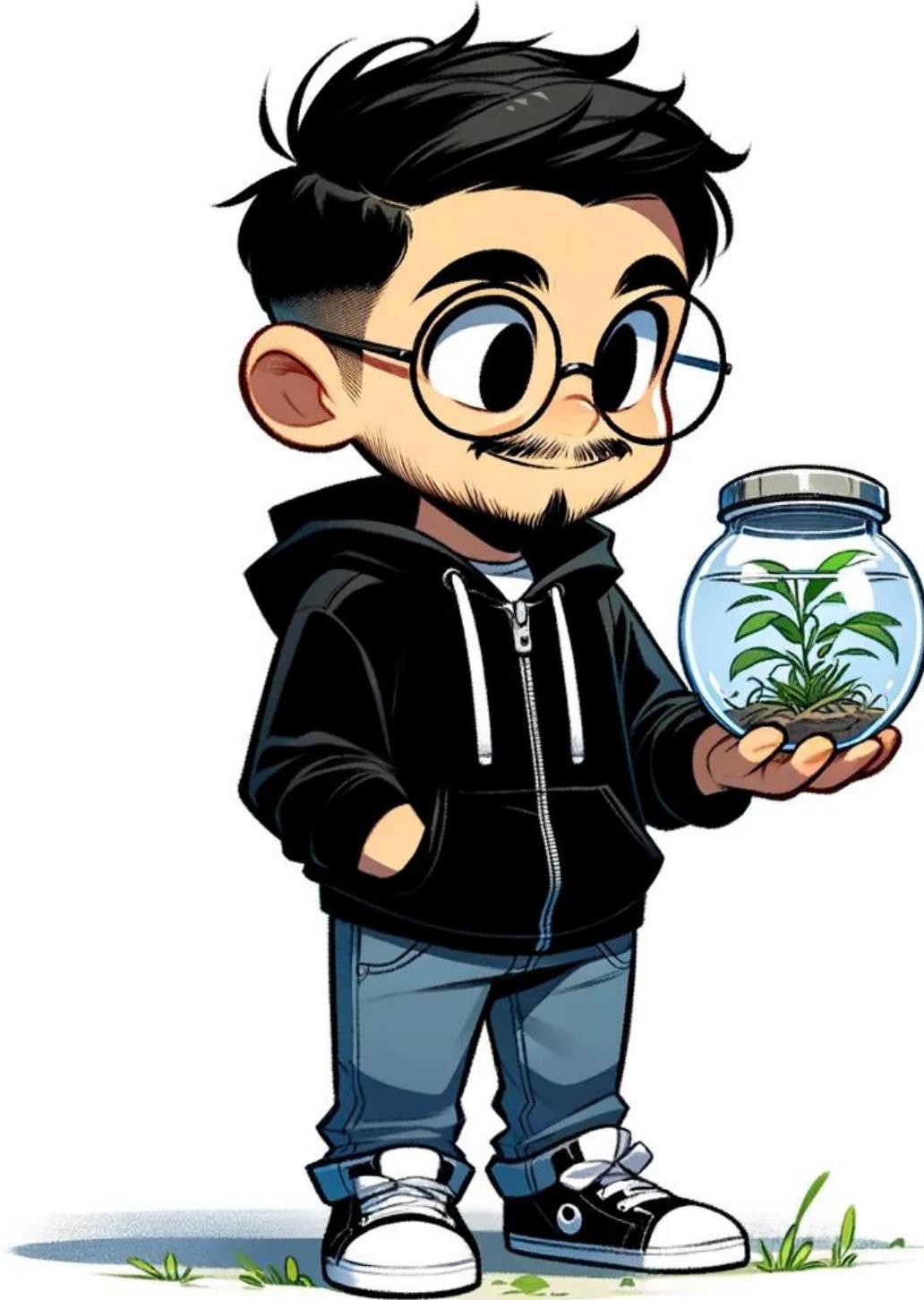
Pour cela, j'utilise soit des doublures de tests pour des tests en boîte blanche, soit de l'interception de flux sortant ou des faux composants externes pour des tests en boîte noire.



## TESTS DE COMPOSANT

Ces tests sont assez coûteux et n'ont pas vocation à couvrir tous les cas, ceux-ci seront couvert par d'autres types de tests. Je les utilise pour valider les cas les plus important du métier.

Je sollicite le métier pour définir ensemble les scénarios qui seront traduits en tests de ce type.



## USAGES ET OUTILS

Mes tests de composant manipulent “l’api” pour l’architecture hexagonale et “l’infrastructure” pour la clean architecture. J’utilise des stubs ou de l’interception de flux pour isoler mon application.

Outils : Gherkin, Cucumber, Karate, Supertest, Cypress, Mockito, Sinon, Nock...



## TESTS END-TO-END

Les tests end-to-end sont des tests en conditions réelles. Je teste mon produit déployé en pré-prod et connecté à l'ensemble de ses dépendances externes.

Ces tests sont très lourds, coûteux et fragiles. C'est pourquoi je me limite à ne tester que les cas les plus critiques définis avec le métier.



## USAGES ET OUTILS

Mes tests end-to-end manipulent le produit via  
“l’api” pour l’architecture hexagonale et via  
“l’infrastructure” pour la clean architecture.

Outils : Gherkin, Cucumber, Karate, Cypress,  
TestContainers, Docker



## TESTS NON FONCTIONNELS

Ces tests me permettent de vérifier des aspects non fonctionnels du produit mais nécessaires à son bon fonctionnement.

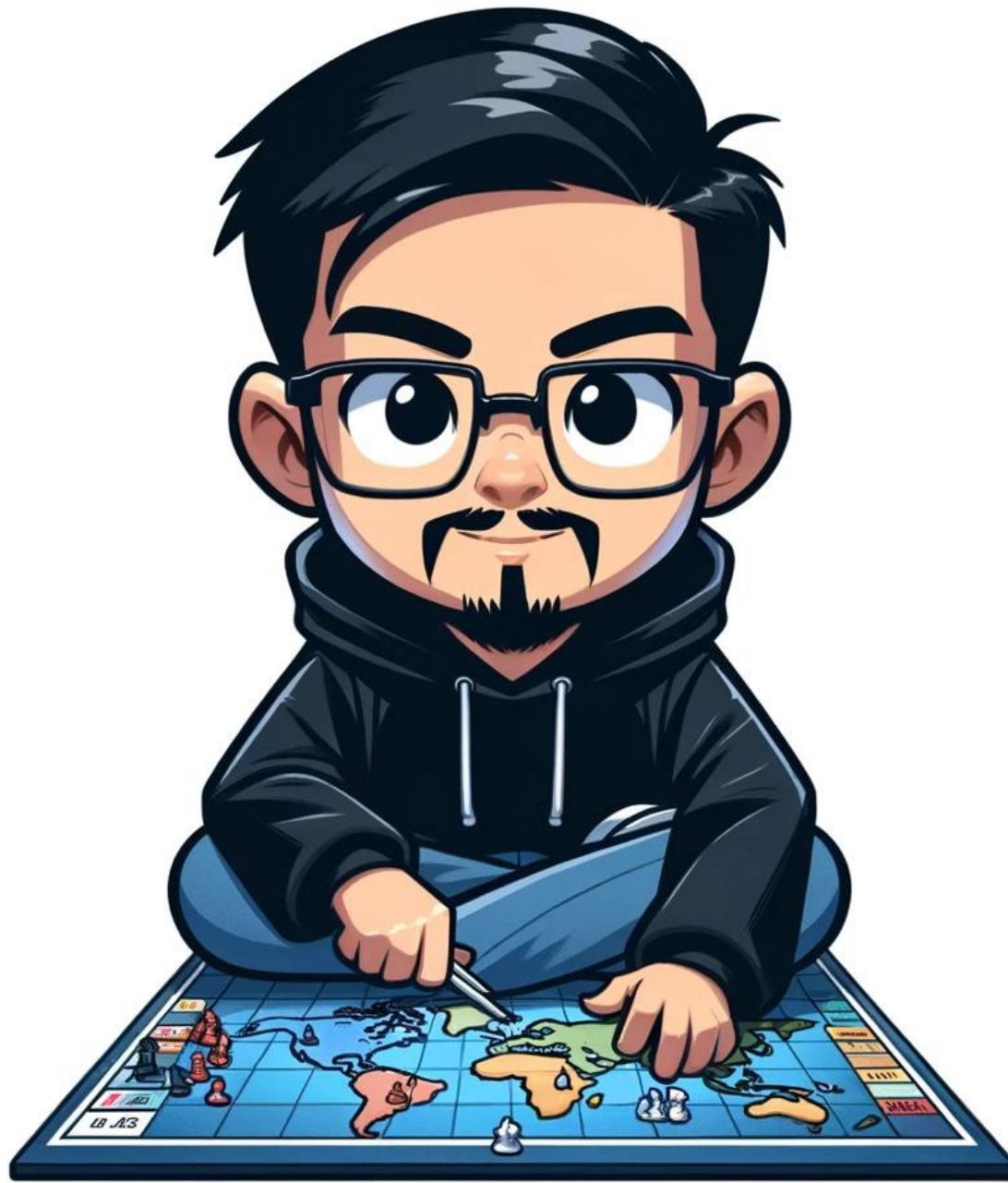
Je peux tester par exemple le temps de réponse, la disponibilité, la résilience ou encore la sécurité d'un produit.



## USAGES ET OUTILS

Les tests à mettre en œuvre sont grandement dépendants du type d'application développé.

Outils : JMeter, Gatling, Burp, SonarQube,  
SAST/DAST/IAST...



# STRATÉGIE DE TEST

Pour plus d'efficacité, j'adopte une stratégie de tests.

J'utilise depuis de nombreuses années la “pyramide de tests” et le “diamant”. Combinée à une architecture centrée métier, l'implémentation de ces stratégies est simple et limpide.

Il existe d'autres stratégies de tests comme le nid d'abeilles, le trophée, le cône de glace ou encore le crabe.



## QUALITÉ DU CODE

Le code relatif aux tests est pour moi aussi important que le code de prod.

Il doit être bien organisé, lisible et facile à retravailler.

En d'autres termes, il doit bénéficier des mêmes attentions que celles portées au code de prod.



## GWT & AAA

GWT = Given / When / Then

AAA = Arrange / Act / Assert

Ces deux façons d'organiser le code d'un test reposent sur le même principe : je commence par établir un contexte, je déclenche ensuite un comportement, puis je vérifie le résultat de ce comportement.



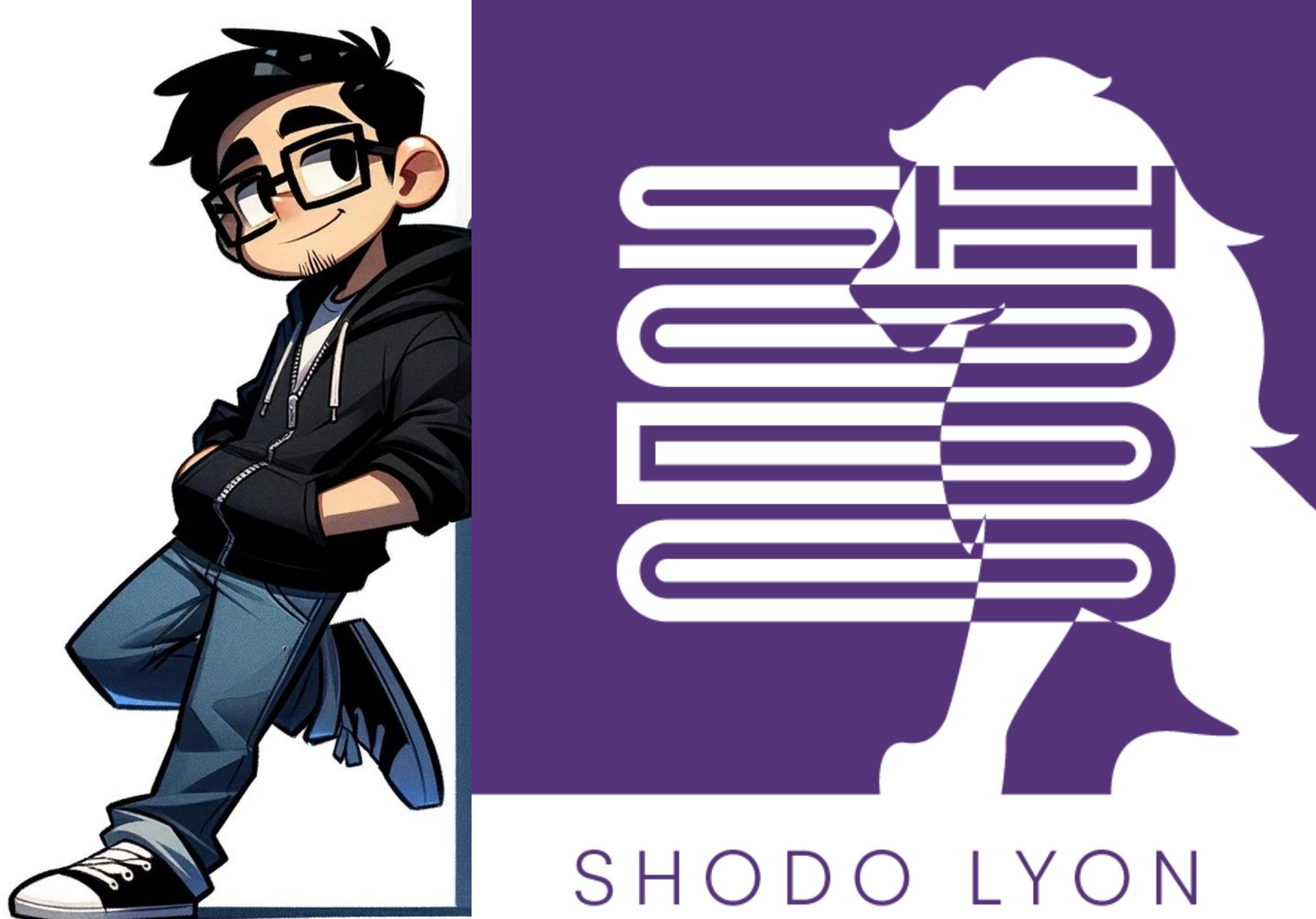
## COUVERTURE DE TESTS

La couverture de tests est une métrique qui m'indique le taux de code couvert par les tests. Elle me permet de révéler des manquements dans les tests.

Elle peut toutefois être erronée si un test ne fait pas d'assertion ou de “mauvaises” assertions.

L'utilisation de cet outil ne remplace donc pas le soin que j'apporte lors de l'écriture des tests.

Jérémy Sorant  
ingénieur logiciel senior  
chez SHODO LYON



I'ESN craft militante  
#justicesociale