CICD

Intégration Continue

I. Tester

II. Types de Test

III. Qualité

Intégration Continue

- *processus liant pratiques et solutions* concourant à
  - éliminer les erreurs - **« régressions »**
  - amener un code à l'état de **livrable**, i.e *démontrable à un client*
- processus exécuté **périodiquement** sur une *branche git commune*
- exemples:
  - tests
  - qualité
  - sécurité

I. Tester

testCase

- fonction ou classe effectuant un test
- les **testCase** sont *découverts par un algorithme fonction* de
  - nom d'un dossier parent - *Ex: tests/*
  - nom du fichier - Ex: *test_*.py*
  - nom de classe ou de méthode/fonction ...
- l'alogrithme de découverte est configuré dans un fichier
- les **testCase** peuvent être aussi collectés par un objet **TestSuite**

procédure de test

- **A**rrange: créer le contexte de test
  - instanciations, connexions, descripteurs de fichier, ...
- **A**ct: exécuter le code à tester
- **A**ssert: **Δ** entre les valeurs *retournée et attendue*
- **C**leanup : libération de variables, fermeture de connexions ...

fixtures

- toute ressource et par prolongement une **fonction/méthode** qui retourne une **ressource nécessaire au test**
- les fixtures peuvent être **paramétrisées**
- les valeurs des paramètres peuvent être injectés depuis des *jeux de données* ou **Providers**
- un objet **Mock** ou **MonkeyPatch** peut *simuler rapidement* la réponse d'un *appel lent* nécessaire au test

assertions

- la grande majorités sont des évaluations booléennes dont le résultat signifie
  - true => **success** *"."*
  - false => **fail** *"F"*
- **xfail:** dans certains cas on peut inverser la logique
  - false => **success** *« eXpected to Fail !! »*
- le retour attendu peut être une **exception levée**

rapport de test

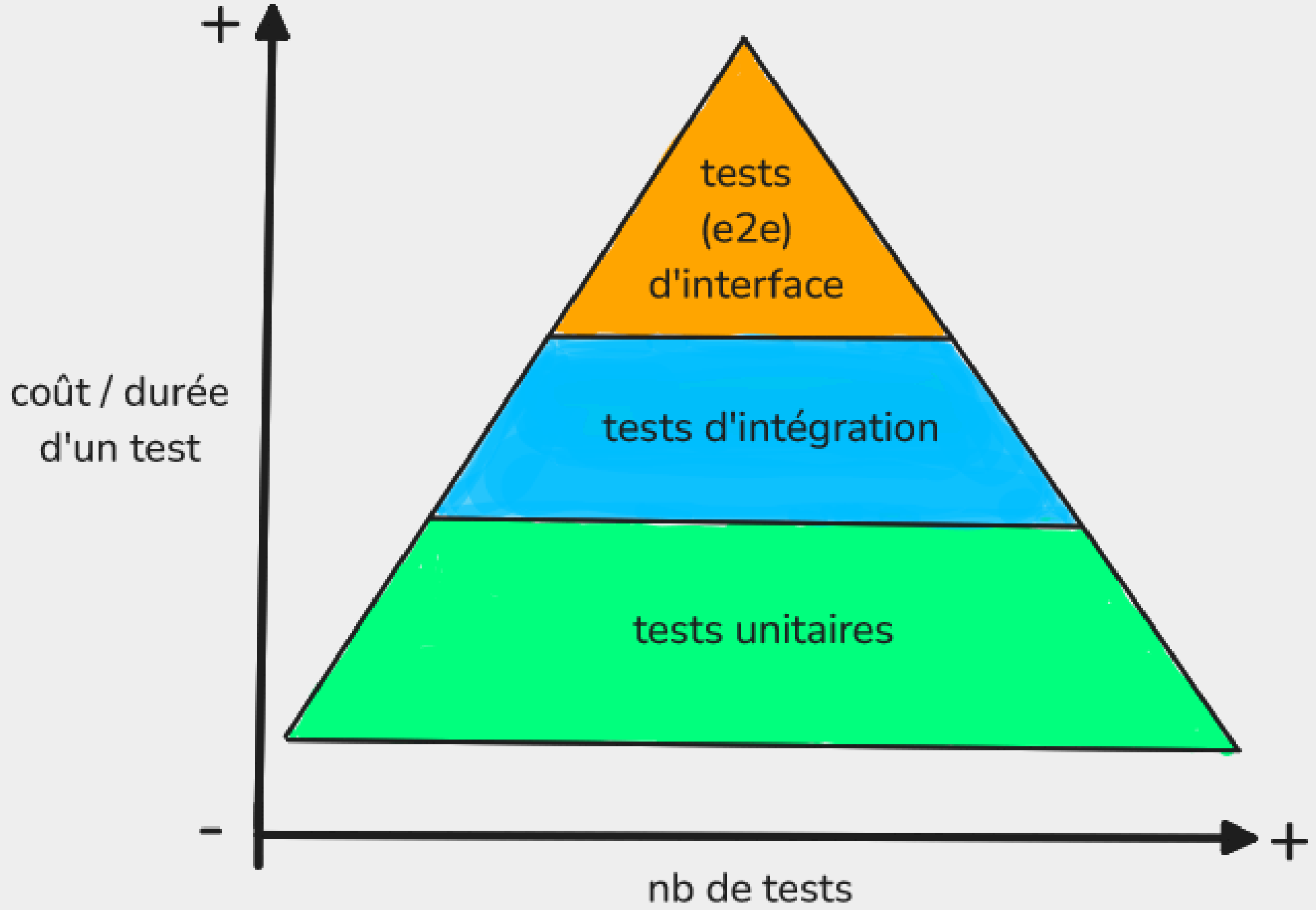
- par défaut, le processus de test retourne une *sortie en texte*
- on peut demander cette sortie dans différents formats
  - html - *visualisation*
  - **xml au format JUnit** - *interface avec un outil CICD*

couverture de code

> *ISTQB:* « Degré, exprimé en pourcentage, selon lequel un élément de couverture spécifié a été exécuté lors d'une suite de test »

- ratio: `(nb d'éléments testés / nb d'éléments testables) * 100`
- type d'éléments, i.e type de *couverture de code par*
  - les **méthodes**: méthode rencontrée *au - 1x par le test*
  - les **instructions**: ligne de code // => **simple et visualisable**
  - les **chemins**: flux de lignes de codes possibles - *trop complexe*

II. Types de Test


+
tests
(e2e)
d'interface
tests d'intégration
tests unitaires
coût / durée
d'un test
-
+
nb de tests

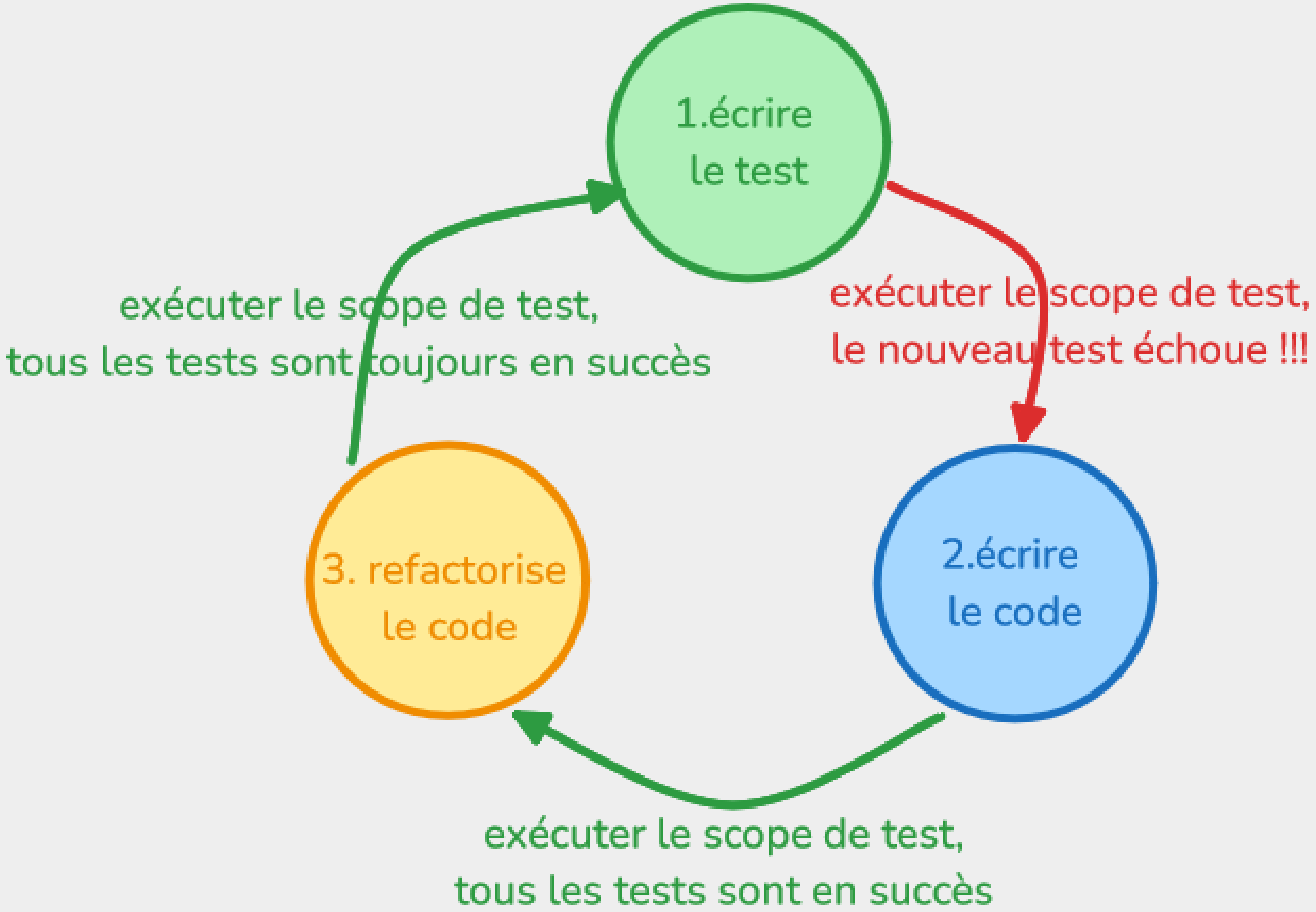
tests unitaires

- les **unités** sont les éléments de code les plus fins
  - en impératif/fonctionnel: **les fonctions**
  - en objet: **les méthodes**
- traditionnellement on écrit un *code à partir d'une spécification*
  => *on en déduit un test*
- **en TDD**, on écrit un **test à partir d'une spécification**
  => **on en déduit le code** de l'unité

> écrire les tests au plus tôt est un élément emlématique d'une démarche agile dite *« Shift Left »*

TDD

- ou « **T**ests **D**riven **D**evelopment »: développemnt pioté par les tests
- « *scope* » ou périmètre => ensemble de *tests existants*
- pour chaque *cas d'utilisation* de la fonctionnalité
  1. écrire le test + exécuter le scope => le **test échoue !**
  2. écrire le cas + exécuter le scope => le *test est en succès*
  3. réfactoriser le code + exécuter le scope => le *test est en succès*


1.écrire
le test
exécuter le scope de test,
le nouveau test échoue !!!
2.écrire
le code
exécuter le scope de test,
tous les tests sont en succès
3. refactorise
le code
exécuter le scope de test,
tous les tests sont toujours en succès

tests d'intégration

- tests des flux informatiques individuels hors exécution de l'application.
- ces flux sont décrits par des **spécifications fonctionnelles**
  - => *vision métier !*
  - écrites sous forme d'*« User Story »*
- séquencent les fonctionnalités unitaires déjà *testés en TDD*

> on veut donc tester l'assemblage des unités !!

BDD

- « **B**ehaviour **D**riven **D**ev. » Dev piloté par les comportements
- un comportement est une **spécification fonctionnelle** écrite sous forme d'*User Story US*

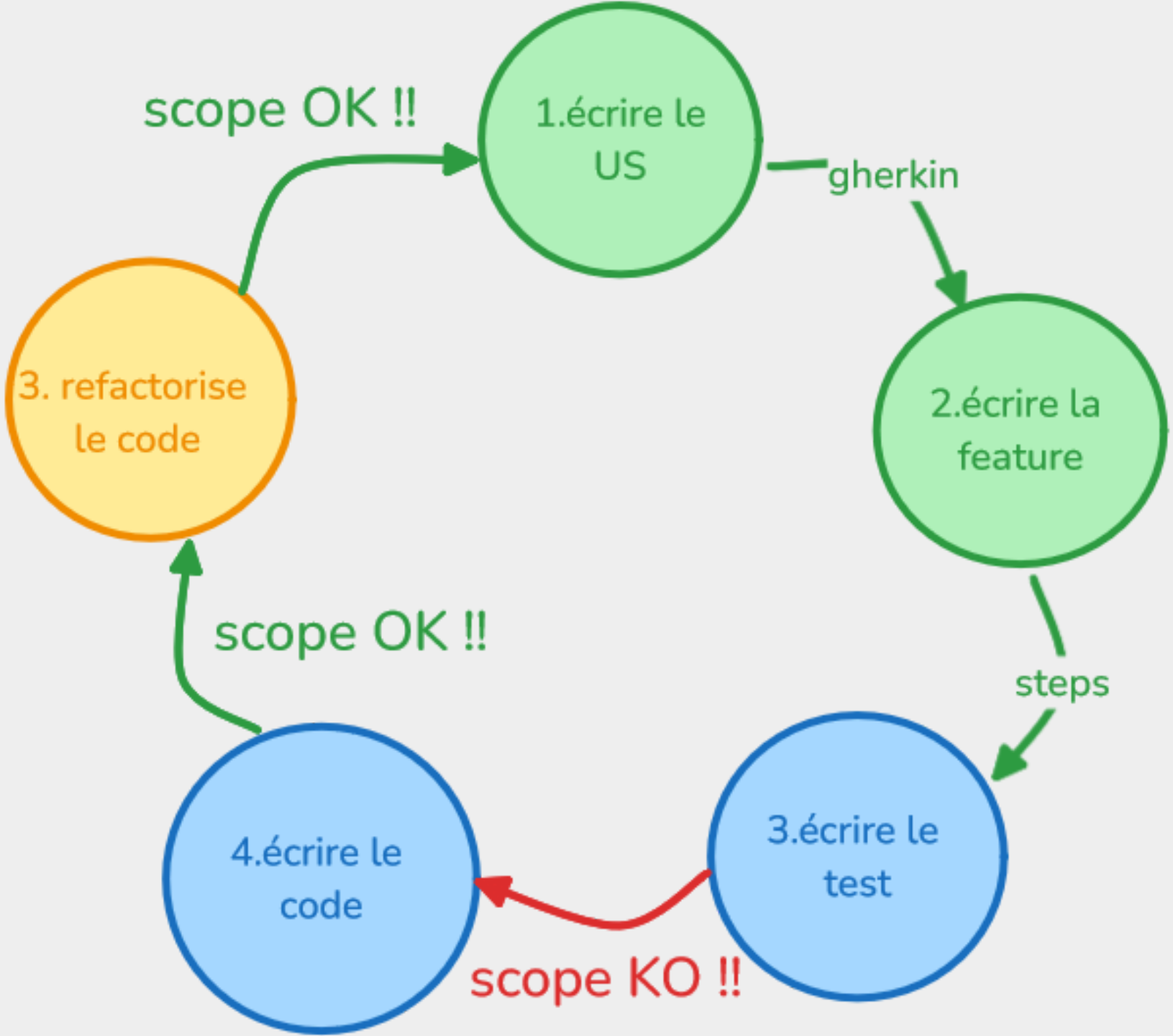
1. l'US est formalisée en utilisant le **langage Gherkin** dans un fichier *.feature*
   - une feature est composée de **scénarios** eux mêmes composés de **triplets Given | When | Then**
2. l'écriture du test reprend la composition de la feature
3. le code écrit et refactorisé *comme en TDD*

exemple de Gherkin

```
# withdraw.feature
Feature: un client retire une somme
   En tant que client
   Je veux retirer une somme de mon compte dans un distributeur
   même quand la banque est fermée

   Scenario Outline: le compte est solvable
      Given le compte a <balance> euros
      And le distributeur a au - <amount> euros
      When je demande <amount> euros
      Then j'obtients <amount> euros
      And mon solde est à <new-balance> euros
      Examples:
      | balance | amount | new-balance
      | 100     | 20     | 80
```

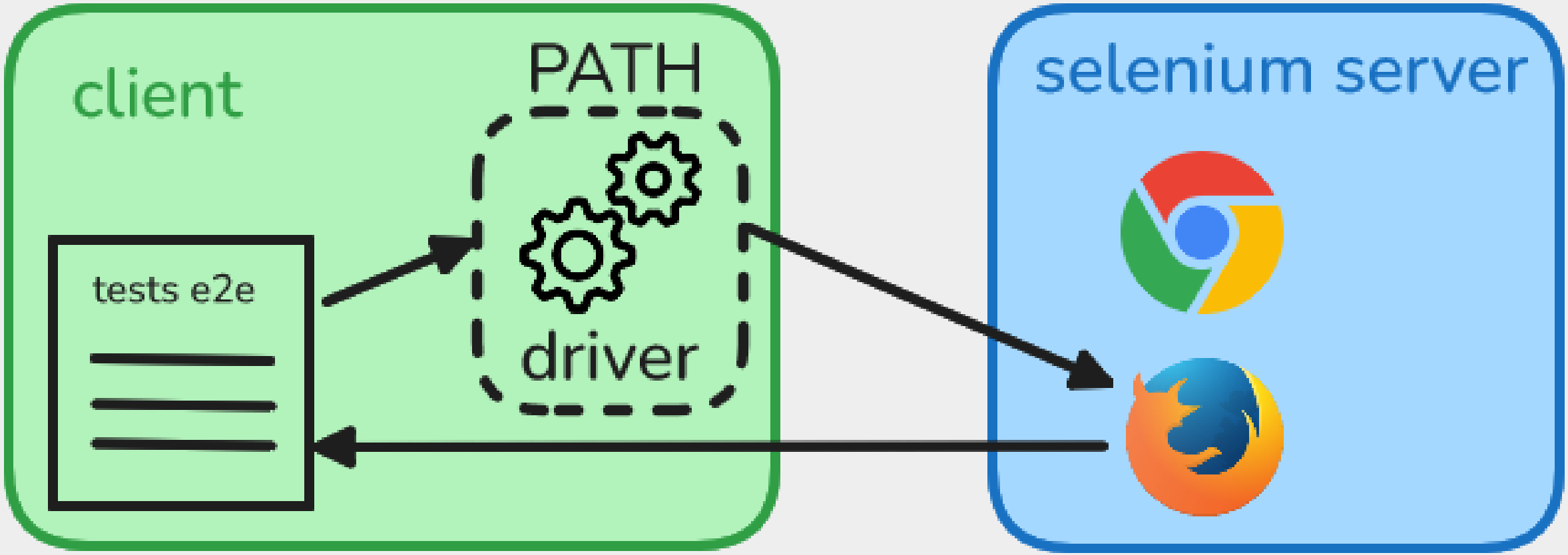
décomposition du test en « steps »

```
@scenario('withdraw.feature', 'le compte est solvable')
def test_withdraw(): pass

@given(parsers.parse("le compte a {balance:int} euros"))
def user_account(account, balance): account["balance"] = balance

@given(parsers.parse("le distrib a au - {amount:int} euros"))
def atm_balance(atm, amount): atm["qty"] = amount

@when("je demande ...")
def witdraw(account, atm, amount): account.withdraw(amount, atm)

@then("mon solde ...")
def assert_new_balance(account, new_balance):
    assert account.balance == new_balance
```


scope OK !!
1.écrire le US
gherkin
2.écrire la feature
steps
3.écrire le test
scope KO !!
4.écrire le code
scope OK !!
3. refactorise le code

tests d'interfaces

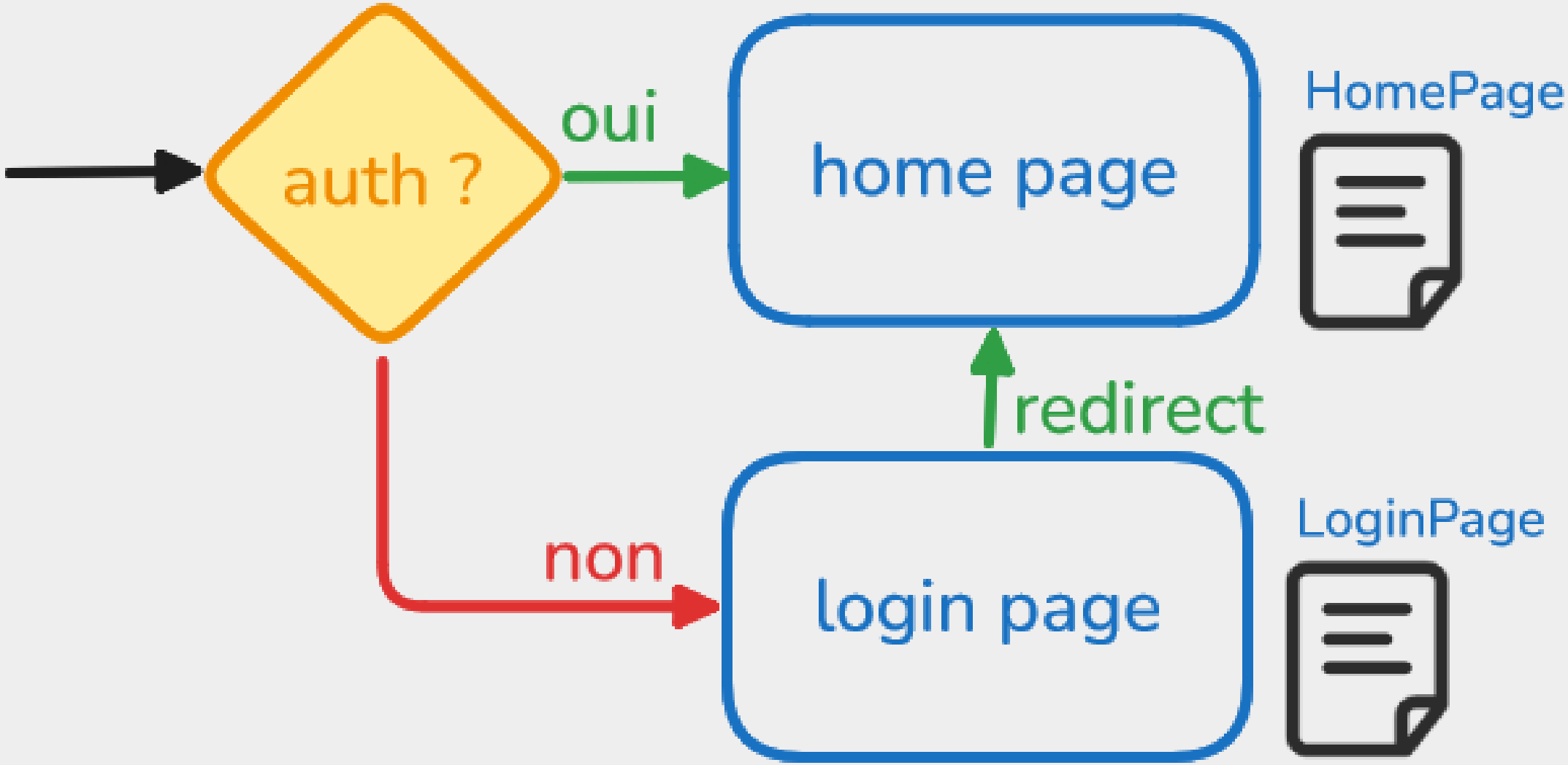
- ou tests E2E « **E**nd To **E**nd »
- consistent à *simuler la navigation* des utilisateurs sur l'application **en exécution**
- les tests doivent avoir accès à un ou plusieurs *moteurs de navigateurs*, au travers d'un **driver**
- plusieurs solutions comme **Selenium** présente une *solution client / serveur* dont le serveur contient des *moteurs configurables*


client
PATH
driver
tests e2e
selenium server

« Page Object Model »

POM est un *design pattern* utilisé par les tests e2e

- une navigation est une séquence d'accès à des **pages html** liées par des **liens http**
- un test e2e peut être structuré en **encapsulant** ces accès successifs dans des *classes représentant les pages*
- puisque une navigation est un *flux métier / utilisateur*, on peut utiliser le **formalisme bdd avec le POM**


auth ?
oui
home page
HomePage
redirect
non
login page
LoginPage

III. Qualité

conventions de code

- **règles** + ou - arbitraires relatives à un langage, une organisation
- **vars:** PascalCase, camelCase, snake_case, kebab-case, ALL_CAPS
- **aération:** saut de lignes, indentations, espaces, marge à droite
- documentation, commentaires, TODOs
- Règles *empiriques* : pas + que
  - 80 caractères par ligne
  - 25 lignes par méthode

formatters

- outil capable de **reformater un code** selon les *conventions de code*
- conventions de code *configurables par un fichier*
- usage commun: exécuter un *formatter avant un commit git*
  - utilisation d'un git **hook pre-commit**

SAST

- ou « **S**tatic **A**pplication **S**ecurity **T**esting »
- analyse du code source de l'application pour trouver des:
  - *Bugs*
  - *Code Smells*
  - *Security Hotspots*
  - *Vulnérabilités*