Engineer

Test unitaire

vérification automatisée d'une petite portion de code isolée

Ce que c’est

Un test unitaire est une petite vérification automatisée d’un élément de code pris isolément : il appelle une fonction, une classe ou un petit module avec des entrées choisies et vérifie que le résultat est bien celui attendu. Ces tests sont généralement conçus pour s’exécuter dans le même processus que le code vérifié, finir rapidement et ne nécessiter ni réseau, ni base de données, ni navigateur. Ce sont des choix utiles en matière de ressources et d’isolation, pas des garanties qui découlent du seul périmètre réduit du test. La pratique qui consiste à les écrire est généralement appelée test unitaire, et leur ensemble dans un projet constitue sa suite de tests unitaires.

Ce qui constitue une unité est moins établi que son nom ne le suggère. Dans le chapitre sur les tests unitaires de *Software Engineering at Google*, Erik Kuefler emploie le terme pour des tests de portée relativement réduite, comme une classe ou une méthode, et ajoute qu’une unité peut aller d’une fonction à plusieurs paquets liés. La frontière utile se situe à l’interface appelée de l’extérieur : une unité fournit une interface dont dépendent des appelants extérieurs, et ses tests utilisent cette interface comme le font ces appelants.

Les frameworks modernes ont une ascendance commune. L’article de Kent Beck, *Simple Smalltalk Testing: With Patterns*, décrit un petit framework composé de fixtures, de cas de test, de vérifications et de suites, et recommande aux développeurs d’écrire eux-mêmes leurs tests unitaires, un par classe. La documentation de `unittest` en Python indique qu’il s’inspire à l’origine de JUnit et conserve les quatre mêmes idées : la fixture de test, préparation nécessaire au test et nettoyage qui le suit ; le cas de test, unité individuelle de test ; la suite de tests ; et l’exécuteur de tests. Les deux tracent une distinction à conserver : un échec est une vérification qui a reçu une mauvaise réponse, tandis qu’une erreur est un événement que le test n’avait pas du tout prévu, comme une exception levée au fond du code.

Les tests unitaires sont les tests au périmètre le plus restreint. Le chapitre d'Adam Bender sur les tests dans le même livre les classe selon leur portée : les tests de portée réduite sont couramment appelés tests unitaires, les tests de portée moyenne couvrant plusieurs composants ensemble sont des tests d'intégration, et les tests de grande portée qui pilotent tout le système sont des tests de bout en bout.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce qu'un test unitaire ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. Une seule équipe transforme une idée en un produit auquel vos clients peuvent se connecter, sans le faire passer entre trois prestataires.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. Application GIS : revenus multipliés par 10 — 10x croissance du chiffre d'affaires portée par le produit cartographique

Application GIS : revenus multipliés par 10 Contactez-nous →

Pourquoi c'est important

La rapidité est la première raison. Un petit test unitaire isolé s'exécute dans le même processus que le code vérifié : un ingénieur peut donc en exécuter des milliers dans une journée ordinaire, et une erreur apparaît alors que la modification qui l'a causée est encore à l'écran. Les règles de Google sont nettes sur ce qu'elles autorisent : un petit test ne doit ni suspendre son exécution pour attendre, ni effectuer d'entrées-sorties, ni accéder au réseau ou au disque. Ces contraintes facilitent la construction de tests rapides et répétables, et une équipe apprend à se fier à une suite qui donne la même réponse à chaque fois.

La deuxième raison est le diagnostic. Lorsqu'un test de portée réduite échoue, le défaut se trouve dans une petite partie du code, généralement celle qui vient de changer. Lorsqu'un test pilotant tout le système échoue, le défaut peut être n'importe où entre le navigateur et la base de données. C'est pourquoi *Software Engineering at Google* présente une pyramide comme une indication approximative : environ 80 % de tests de portée réduite, 15 % de tests d'intégration et 5 % de tests de bout en bout.

La troisième raison est le changement. Une suite qui vérifie le comportement à travers une interface publique permet de restructurer le code derrière cette interface sans crainte : si les tests réussissent encore, les besoins des appelants

restent satisfaits. L'idéal de Kuefler est un test qui ne doit jamais changer sans changement des exigences ; une simple refonte interne ne touche donc aucun test.

Les tests unitaires servent aussi de documentation. Un test nommé d'après un comportement, avec des entrées simples et un résultat attendu écrit en entier, montre au prochain lecteur comment appeler le code et ce qu'il promet ; contrairement à un commentaire, il échoue lorsque cela cesse d'être vrai.

À quoi cela ressemble en pratique

Prenons une fonction qui lit des durées comme `1h30m` dans un fichier de configuration et les convertit en secondes :

```
# durations.py
UNITS = {"h": 3600, "m": 60, "s": 1}


def to_seconds(text: str) -> int:
    """Turn "1h30m" or "45s" into a whole number of seconds."""
    total, digits = 0, ""
    for char in text:
        if char.isdigit():
            digits += char
        elif char in UNITS and digits:
            total += int(digits) * UNITS[char]
            digits = ""
        else:
            raise ValueError(f"not a duration: {text!r}")
    if digits or not text:
        raise ValueError(f"not a duration: {text!r}")
    return total
```

Ses tests unitaires, écrits pour pytest, vérifient deux comportements : ce qu'elle accepte et ce qu'elle refuse.

```
# test_durations.py
import pytest

from durations import to_seconds


@pytest.mark.parametrize(
```

```
    "text,expected",
    [("45s", 45), ("2m", 120), ("1h30m", 5400), ("0s", 0)],
)
def test_counts_each_unit(text, expected):
    assert to_seconds(text) == expected


@pytest.mark.parametrize("text", ["", "90", "h", "1x", "1.5h"])
def test_refuses_what_it_cannot_read(text):
    with pytest.raises(ValueError):
        to_seconds(text)
```

Dans le répertoire, `pytest` repère tous les fichiers nommés `test_*.py` ou `*_test.py` ; les deux fonctions donnent neuf cas, un par ligne de paramètres. La documentation de pytest décrit chaque test comme une préparation, une action, une assertion et un nettoyage ; ici, la préparation est la ligne de paramètres, l'action est l'appel unique, et il n'y a rien à nettoyer puisque rien en dehors de la fonction n'est touché. Si quelqu'un fixe ensuite par erreur la minute à 3 600 secondes, deux cas échouent et le rapport nomme chacun par ses paramètres : `test_counts_each_unit[2m-120]` affiche 7 200 au lieu des 120 attendus, et `test_counts_each_unit[1h30m-5400]` affiche 111 600 au lieu de 5 400. Les valeurs attendues sont calculées à la main, pas copiées sur ce qu'a renvoyé le code à sa première exécution.

Les mêmes vérifications dans le `unittest` de la bibliothèque standard seraient une sous-classe de `unittest.TestCase` avec des méthodes dont le nom commence par `test`, utilisant `assertEqual` et `assertRaises` ; `python -m unittest` trouve par défaut les fichiers correspondant à `test*.py`. Go intègre cette convention dans sa chaîne d'outils : une fonction `func TestXxx(t *testing.T)` dans un fichier finissant par `_test.go` est exécutée par `go test`.

L'augmentation du volume change le suivi, pas le principe. Le générateur de base de données soumis à 981 cas de test, dans le cadre d'un travail réalisé en juillet 2026, compte 383 fonctions de test qui deviennent 981 cas une fois les tests paramétrés développés, répartis dans 45 fichiers et 7 944 lignes. Sa compilation impose un plancher de couverture des branches de 92 %, avec environ 95 % mesurés, et traite les avertissements comme des échecs. La page expose franchement la limite du chiffre : deux défauts découverts plus tard se trouvaient dans du code que le rapport de couverture indiquait comme couvert.

Où les problèmes surviennent

Les tests reproduisent l'implémentation. Un test qui accède à des fonctions auxiliaires privées ou vérifie la séquence exacte des appels internes casse à chaque restructuration, alors que rien de visible pour un appelant n'a changé. Kuefler parle de test fragile et désigne le recours excessif aux frameworks de simulation comme la cause la plus fréquente des problèmes de tests d'interactions. Testez par l'interface publique, vérifiez l'état obtenu plutôt que les appels qui y ont mené et utilisez de vrais objets lorsqu'ils sont peu coûteux.

Les tests dépendent les uns des autres. Un test crée un enregistrement, le suivant le lit, et la suite ne réussit que dans cet ordre. Exécutez un seul test, ou les mêmes dans un autre ordre, et elle échoue sans raison visible pour le lecteur. Le framework de Beck créait puis jetait ses données pour chaque test, et la documentation de `unittest` demande qu'un cas de test soit entièrement autonome. Donnez à chaque test sa propre fixture et nettoyez après son exécution.

La couverture devient le but. La couverture montre qu'une ligne s'est exécutée, pas qu'une vérification a contrôlé son effet. Un test sans assertion augmente le chiffre et ne prouve rien ; *Software Engineering at Google* avertit qu'une barre de 80 % tend à devenir un plafond, avec des modifications qui arrivent exactement au minimum. Traitez le chiffre comme un plancher et lisez les assertions en revue.

Un test par méthode. Associer chaque méthode de production à une méthode de test semble ordonné, mais le test grossit à chaque nouvelle branche de la méthode, jusqu'à ce que personne ne sache dire ce qu'il vérifie. Kuefler conseille de tester les comportements : un comportement par test, nommé selon ce qu'il prouve, pour que le rapport d'échec se lise comme une phrase décrivant ce qui a cassé.

Le test attend que le temps passe. Un `sleep()` pour laisser quelque chose finir ralentit la suite à chaque exécution et échoue encore sur une machine chargée. Dans un test unitaire, le temps est une entrée : passez une horloge ou un horodatage, pour que le test choisisse lui-même l'instant voulu au lieu de l'attendre.

Pour aller plus loin

- unittest : framework de tests unitaires — le framework de la bibliothèque standard Python : fixtures, cas de test, suites, exécuteurs, découverte et distinction entre échec et erreur.
- Anatomie d'un test — l'explication de pytest sur la préparation, l'action, l'assertion et le nettoyage.
- Premiers pas — l'introduction de pytest : simple `assert`, découverte des tests et `pytest.raises`.
- Tests unitaires — le chapitre d'Erik Kuefler dans *Software Engineering at Google* sur les tests fragiles, les API publiques, l'état plutôt que les interactions et les tests de comportement.
- Simple Smalltalk Testing: With Patterns — l'article de Kent Beck sur le framework, l'argument en faveur d'un test unitaire par classe et la distinction entre échecs et erreurs, dans une copie archivée.
- testing — le paquet de test standard de Go : fonctions `TestXxx`, fichiers `_test.go` et `go test`.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce qu'un test unitaire ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/unit-test/