**Engineer**

# Модульний тест

автоматична перевірка малого ізольованого фрагмента коду

## Що це таке

Модульний тест — це невелика автоматична перевірка одного фрагмента коду окремо від решти: він викликає функцію, клас або невеликий модуль із підібраними вхідними даними й перевіряє, що результат саме такий, яким має бути. Такі тести зазвичай проєктують так, щоб вони виконувалися в тому самому процесі, що й код, який перевіряють, швидко завершувалися й не потребували мережі, бази даних або браузера. Це корисні рішення щодо ресурсів та ізоляції, а не гарантії, що випливають лише з вузького обсягу тесту. Такий тест також називають юніт-тестом, а саму практику — модульним тестуванням. Сукупність таких тестів у проєкті — його набір модульних тестів.

Що саме вважати модулем, не так однозначно, як підказує назва. У розділі про модульне тестування книжки *Software Engineering at Google* Erik Kuefler називає так тести відносно вузького обсягу, наприклад одного класу чи методу, і додає, що модуль може бути завбільшки з одну функцію, а може охоплювати кілька пов'язаних пакетів. На практиці корисна межа проходить там, де код викликають інші: модуль — це те, що надає інтерфейс, на який хтось зовні покладається, а його тести користуються цим інтерфейсом так само, як ті, хто його викликає.

Сучасні фреймворки мають спільний родовід. Стаття Кента Бека *Simple Smalltalk Testing: With Patterns* описувала невеликий фреймворк із фікстур, тестових випадків, перевірок і наборів тестів та радила розробникам самим писати модульні тести, по одному на клас. Документація модуля `unittest` зі стандартної бібліотеки Python прямо каже, що його спочатку надихнув JUnit, і в ньому ті самі чотири поняття: тестова фікстура (підготовка, якої потребує тест, і прибирання після нього), тестовий випадок (окрема одиниця тестування), набір тестів і запускач тестів. Обидва проводять межу, яку варто

пам’ятати: збій (failure) — це перевірка, що отримала неправильну відповідь, а помилка (error) — те, чого тест узагалі не передбачав, наприклад виняток, викинутий глибоко всередині коду.

Модульні тести лежать на вузькому краї шкали. У розділі про тестування в тій самій книжці Adam Bender упорядковує тести за обсягом: вузькі зазвичай називають модульними, середні, що охоплюють кілька компонентів разом, — інтеграційними, а великі, що проганяють усю систему, — наскрізними тестами.

Застосуйте поняття до рішення

Застосуйте «Що таке модульний тест?» до конкретного рішення. Ми можемо пов'язати поняття із системою, яку ви створюєте або купуєте. Одна команда веде ідею від ескізу до продукту, у який ваші клієнти вже можуть увійти, а не перекидає роботу між трьома підрядниками.

**Переглянути наші послуги** Зв'язатися з нами →

Перевірте припущення за терміном

Використайте пояснення та пов'язані роботи, щоб сформулювати запитання для проєкту. Ми можемо перевірити припущення разом до вибору інструмента. Збільшила дохід GIS-застосунку в 10 разів — 10x зростання виторгу завдяки картографічному продукту

**Збільшила дохід GIS-застосунку в 10 разів** Зв'язатися з нами →

## Чому це важливо

Перша причина — швидкість. Малий ізольований модульний тест виконується в тому самому процесі, що й код, тож розробник може запустити тисячі таких тестів за звичайний робочий день, і помилка виявляється, поки зміна, що її спричинила, ще відкрита на екрані. Настанови Google тут однозначні: малий тест не може засинати, виконувати введення-виведення чи звертатися до мережі або диска. Ці обмеження полегшують створення швидких тестів із повторюваними результатами, і команда вчиться довіряти набору, який щоразу дає ту саму відповідь.

Друга — діагностика. Коли падає вузький тест, вада сидить у невеликому фрагменті коду, найчастіше в щойно зміненому. Коли падає тест, що проганяє всю систему, причина може бути будь-де між браузером і базою даних. Тому *Software Engineering at Google* подає піраміду як приблизну настанову: близько 80% вузьких тестів, 15% інтеграційних і 5% наскрізних.

Третя — зміни. Набір, що перевіряє поведінку через публічний інтерфейс, дає змогу без страху перебудовувати код за цим інтерфейсом: якщо тести досі проходять, ті, хто викликає код, і далі отримують те, що їм потрібно.

Ідеал, який описує Kuefler, — тест, який не доводиться змінювати, доки не зміняться вимоги, тож чистий рефакторинг не зачіпає жодного тесту.

Модульні тести ще й документують. Тест, названий за поведінкою, з простими вхідними даними й повністю виписаним очікуваним результатом показує наступному читачеві, як цей код викликати й що він обіцяє, а на відміну від коментаря тест падає, щойно перестає відповідати дійсності.

## Як це виглядає на практиці

Візьмімо функцію, яка читає тривалості на кшталт `1h30m` із конфігураційного файлу й переводить їх у секунди:

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


def to_seconds(text: str) -> int:
    """Переводить "1h30m" або "45s" у ціле число секунд."""
    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
```

Її модульні тести, написані для pytest, перевіряють дві поведінки: що функція приймає і що відкидає.

```
# 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)
```

Команда `pytest`, запущена в каталозі, збирає всі файли з назвами `test_*.py` або `*_test.py`, і дві функції розгортаються в дев'ять випадків, по одному на кожен рядок параметрів. Документація pytest описує кожен тест як arrange, act, assert і cleanup, тобто підготовку, дію, перевірку й прибирання; тут підготовка — це рядок параметрів, дія — єдиний виклик, а прибирати нічого, бо поза функцією нічого не зачеплено. Якщо згодом хтось помилково задасть для хвилини `3600` секунд, два випадки не пройдуть, і звіт назве кожен за його параметрами: `test_counts_each_unit[2m-120]` покаже `7200` проти очікуваних `120`, а `test_counts_each_unit[1h30m-5400]` — `111600` проти `5400`. Зверніть увагу: очікувані значення пораховано вручну, а не скопійовано з того, що код повернув під час першого запуску.

Ті самі перевірки в `unittest` зі стандартної бібліотеки мали б вигляд підкласу `unittest.TestCase` з методами, назви яких починаються з `test`, і з викликами `assertEqual` та `assertRaises`; `python -m unittest` за замовчуванням шукає файли, що відповідають шаблону `test*.py`. Go вбудовує цю домовленість у сам інструментарій: функцію `func TestXxx(t *testing.T)` у файлі, назва якого закінчується на `_test.go`, запускає `go test`.

Масштаб змінює облік, а не ідею. Генератор бази даних, покритий 981 тестовим випадком — робота, виконана в липні 2026 року, — налічує 383 тестові функції, які з усіма наборами параметрів дають 981 тестовий випадок, у 45 файлах загальним обсягом 7 944 рядки. Збірка вимагає щонайменше 92% покриття гілок, виміряне покриття становить близько 95%, а попередження вважаються помилками. Сторінка чесно каже, де межа цього числа: два дефекти, знайдені в проєкті згодом, містилися в коді, який звіт про покриття показував як покритий.

# Де виникають проблеми

**Тести віддзеркалюють реалізацію.** Тест, який лізе у приватні допоміжні функції або перевіряє точну послідовність внутрішніх викликів, ламається щоразу, коли код перебудовують, хоча для того, хто його викликає, нічого не змінилося. Kuefler називає такий тест крихким і вважає надмірне покладання на фреймворки для моків найпоширенішою причиною того, що тести взаємодій псуються. Тестуйте через публічний інтерфейс, перевіряйте стан, який вийшов у результаті, а не виклики, що до нього привели, і беріть справжні об'єкти там, де вони дешеві.

**Тести залежать один від одного.** Один тест створює запис, наступний його читає, і набір проходить лише в такому порядку. Запустіть один тест окремо або ті самі тести в іншому порядку — і він упаде з причини, якої читач не бачить. Фреймворк Бека створював і викидав тестові дані для кожного тесту окремо, а документація `unittest` вимагає, щоб тестовий випадок був цілком самодостатнім. Дайте кожному тесту власну фікстуру й прибирайте після нього.

**Покриття стає метою.** Покриття показує, що рядок виконався, а не що хтось перевірив, що він зробив. Тест без жодної перевірки піднімає число й нічого не доводить, а *Software Engineering at Google* застерігає, що поріг у 80% зазвичай перетворюється на стелю і зміни зупиняються якраз на мінімумі. Вважайте число підлогою, а перевірки читайте під час рев'ю коду.

**Один тест на метод.** Пара «метод — тестовий метод» виглядає охайно, але тест розростається щоразу, коли метод отримує нову гілку, аж поки ніхто не може сказати, що він перевіряє. Kuefler радить натомість тестувати поведінку: одна поведінка на тест, а назва — за тим, що він доводить. Тоді звіт про падіння читається як речення про те, що саме зламалося.

**Тест чекає, поки мине час.** Виклик `sleep()`, щоб щось устигло завершитися, сповільнює набір на кожному запуску і все одно падає на завантаженій машині. У модульному тесті час — це вхідні дані: передайте кодові годинник або позначку часу, щоб тест сам задав потрібний момент, а не чекав на нього.

# Що почитати далі

- unittest: Unit testing framework — фреймворк зі стандартної бібліотеки Python: фікстури, тестові випадки, набори, запускачі, пошук тестів і різниця між збоєм та помилкою.
- Anatomy of a test — пояснення pytest про arrange, act, assert і cleanup.
- Get Started — вступ від самого pytest: звичайний `assert`, пошук тестів і `pytest.raises`.
- Unit Testing — розділ Erik Kuefler у *Software Engineering at Google* про крихкі тести, публічні API, стан замість взаємодій і тестування поведінки.
- Simple Smalltalk Testing: With Patterns — стаття Кента Бека про фреймворк, про модульні тести по одному на клас і про різницю між збоями та помилками, в архівній копії.
- testing — стандартний пакет Go для тестування: функції `TestXxx`, файли `_test.go` і `go test`.

Словник

Обирайте за фактичними умовами

Опишіть робочий процес, наявні інтерфейси та користувачів, яким потрібна зміна. Ми можемо визначити перевірюваний результат і спосіб передачі, який дасть вашій команді змогу підтримувати його.

**Дивитися, скільки це коштує** Зв'язатися з нами →

# Зробіть термінологію практичною

Запитайте, як «Що таке модульний тест?» стосується вашої організації. Конкретного прикладу з поточної системи достатньо для початку розмови.

**Переглянути наші послуги**

welcome@engineer.company +45 50 10 61 84 Копенгаген, Данія

Візьміть цю сторінку із собою

Візитівка

**Дбайливо зроблено з любов'ю — D.**

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

**Авторське право © 2025 – дотепер · Engineer ApS · Усі права захищено.**

**Надруковано з https://engineer.company/uk/glossary/unit-test/**