Engineer

Розробка через тестування

написання тесту до коду, по одному невеликому тесту за раз

Що це таке

Розробка через тестування — це спосіб програмувати, за якого тест на певну поведінку пишуть раніше за код, що цю поведінку забезпечує, і роблять це по одному невеликому тесту за раз. Кент Бек, який сам називає свою роль у ній повторним відкриттям, описує її як робочий процес програміста. У 2023 році, переконавшись, що розробники по-різному розуміють цей термін, він виклав процес у короткому дописі під назвою Canon TDD. У дописі він майже скрізь пише скорочення TDD, від англійського test-driven development.

Процес складається з п’яти кроків:

- Скласти список тестових сценаріїв, які треба покрити.
- Перетворити рівно один пункт цього списку на конкретний тест, який можна запустити.
- Змінити код так, щоб проходили цей тест і всі попередні, а нові пункти дописувати до списку, щойно вони спадають на думку.
- За бажанням провести рефакторинг, щоб поліпшити дизайн реалізації.
- Повертатися до кроку 2, доки в списку лишаються пункти.

  Список — це аналіз, але аналіз поведінки, а не дизайну: спершу базовий випадок, потім варіації, а наостанок те, чого зміна не повинна зламати в коді, який уже працює. Тест із кроку 2 — справжній автоматизований тест із підготовкою, викликом і твердженнями (assertions), і під час першого запуску він падає, бо поведінку, яку він описує, ще тільки належить написати. Перехід від тесту, що падає, до тесту, що проходить, Бек називає переходом від червоного до зеленого.

Корисно знати й те, чим цей процес не є. Написання всіх тестів до будь-якого коду — часта претензія до TDD, і Бек відповідає прямо: це не TDD, і хто критикує таке, критикує щось інше. В інший бік він так само спокійний.

Команда, яка працює інакше й має добрі результати, може працювати так і далі, а за буквальне дотримання кроків, як він пише, золотої зірочки не дають.

Застосуйте поняття до рішення

Застосуйте «Що таке розробка через тестування?» 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

Бек прямо пише, навіщо потрібен цей процес. Програміст, який змінює поведінку системи, має отримати систему, де все, що працювало раніше, і далі працює, нова поведінка працює як очікувалося, код готовий до наступної зміни, а сам програміст і його колеги впевнені в усіх трьох пунктах. Оскільки кожен тест пишуть, поки зміна мала і ще свіжа в пам’яті, страхувальна сітка росте з тією самою швидкістю, що й код, а не відкладається на потім.

Друга причина — дизайн. Бек розрізняє два види дизайнерських рішень: як певну поведінку викликають і як система її реалізує. Тест, написаний першим, змушує ухвалити рішення щодо інтерфейсу, бо тест мусить щось викликати й сказати, що має повернутися. Рішення щодо реалізації чекають на крок рефакторингу, коли їх уже захищають тести, що проходять. Тест, який незручно писати, — це ранній сигнал, що інтерфейсом буде незручно користуватися.

Третя причина — подвійна перевірка. Очікуване значення в тесті людина виводить із вимоги, а код доходить до нього іншим шляхом. Коли результати збігаються, два незалежні шляхи привели до однієї відповіді, і, за словами Бека, саме подвійна перевірка дає TDD значну частину його цінності як контролю.

Нарешті, процес підказує, коли зупинитися. Бек радить писати тести й код доти, доки страх за поведінку коду не перетвориться на нудьгу. Звичка ця значно старша за допис. Бек відносить своє повторне відкриття до проміжку між 1994 роком, коли він написав SUnit для одного зі своїх консалтингових клієнтів, і жовтнем 1995-го, коли показав TDD Ворду Каннінгему на конференції OOPSLA в Остіні. Його стаття Simple Smalltalk Testing описує простий фреймворк тестування для Smalltalk, і документація unittest у Python досі посилається на неї як на оригінальну статтю Бека про патерн, спільний з unittest.

Як це виглядає на практиці

Уявімо перевірку, яка має знаходити заборонені модні слівця в текстах вебсайту. Робота починається зі списку тестів у вигляді звичайних нотаток:

```
- заборонене слово потрапляє у звіт
- заборонене слово з великої літери потрапляє у звіт
```

Крок 2 перетворює рівно один пункт на тест. Перший пункт стоїть першим не випадково: він передає перевірці відоме заборонене слово й вимагає, щоб вона повідомила про нього. Тест не проходить, доки перевірка не навчиться його знаходити. Тест іде у файл `test_wording.py`, який pytest підхоплює, бо ім'я відповідає шаблону `test_*.py`:

```
from wording import find_banned


def test_reports_a_banned_word():
    assert find_banned("we leverage it", {"leverage"}) == ["leverage"]
```

Запуск `pytest -q` одразу падає, бо імпорт `find_banned` не вдається. Саме в цьому суть: тест бачили червоним, тож коли згодом він пройде, це щось означатиме. Поки що досить найпростішого коду, з яким тест проходить; він лежить у `wording.py`:

```
def find_banned(text, banned):
    return [word for word in text.split() if word in banned]
```

Тест проходить. Наступний пункт стає наступним тестом:

```
def test_reports_a_capitalised_banned_word():
    assert find_banned("Leverage it", {"leverage"}) == ["Leverage"]
```

Тест падає, і оскільки pytest показує проміжні значення у твердженні, у виводі видно порожній список там, де очікувався `["Leverage"]`. Тест проходить, щойно замість `word` порівнювати `word.lower()`. Пишучи цю зміну, програміст помічає, що `text.split()` залишає розділові знаки при слові, тож `leverage,` із комою проскочить. Це йде до списку, а не в код, і стає третім тестом:

```
def test_reports_a_banned_word_before_a_comma():
    assert find_banned("Use it, leverage, win", {"leverage"}) == ["leverage"]
```

Тест падає, і ще один рядок змушує пройти всі три:

```
def find_banned(text, banned):
    words = [word.strip(".,;:!?") for word in text.split()]
    return [word for word in words if word.lower() in banned]
```

Крок 4 необов’язковий, і поліпшувати тут майже нічого: два рядки, кожен зі своїм завданням. Для кожного пункту списку тепер є тест, що проходить, тож роботу завершено. Усі очікувані значення в тестах виписано з пунктів списку, а не скопійовано з прогону.

Порядок цього списку обрано свідомо, і портфоліо сайту показує чому. Проєкт, у якому написали тести для самих перевірок, — роботу виконано в серпні 2026 року — встановив, що власні скрипти перевірки, серед них перевірка орфографії, перевірка стилю викладу й перевірка узгодженості чисел, місяцями давали зелений результат і що скрипт, який бачив лише бездоганні вхідні дані, неможливо відрізнити від скрипту, що взагалі нічого не читає. Рішення полягало в тому, щоб підкласти саме той дефект, який кожна перевірка має знаходити: перевірці орфографії подають навмисно неправильно написане слово, і тест падає, якщо прогін завершується без зауважень; перевірці стилю подають текст від першої особи, і вона мусить його відхилити; перевірці на модні слівця подають заборонене слово. Як

з’ясувалося, два скрипти перевірки читали вужчий набір файлів, ніж зазначено в їхній документації. Сторінка не описує TDD, але її висновок той самий, що його TDD закладає з першого кроку: тест чогось вартий лише тоді, коли його бачили червоним.

Де виникають проблеми

Усі тести пишуть наперед. Перетворити кожен пункт списку на тест ще до того, як бодай один пройде, — саме те хибне прочитання, яке виокремлює Бек. Коли перший тест змушує переглянути рішення, усі спекулятивні тести доводиться переробляти, а програміста, який дійшов до шостого тесту й досі не бачив жодного зеленого, за словами Бека, охоплює зневіра або нудьга. Напишіть один тест, доведіть його до проходження, тоді беріть наступний. Якщо новий тест показує, що вже зроблене не впорається з якимось випадком, треба вирішити, продовжувати чи почати спочатку; Бек радить почати спочатку й обрати інший порядок.

Тест без твердження. Тест, який викликає код і нічого не перевіряє, підвищує показник покриття і нічого не доводить. Кожному тесту потрібні підготовка, виклик і твердження, а Бек радить час від часу писати тест задом наперед, починаючи з тверджень.

Твердження видаляють. Коли тест ніяк не проходить, спокусливо видалити твердження, і тоді тест лише вдає, що проходить. Доведіть його до проходження по-справжньому або поверніть пункт до списку й перегляньте порядок.

Очікувані значення копіюють із виводу. Запустити код, скопіювати те, що він повернув, і вставити це в тест означає перетворити подвійну перевірку на одинарну: тепер тест підтверджує лише, що код робить те, що вже робив. Виводьте очікуване значення з вимоги, за потреби вручну.

Рефакторинг під час проходження. Міняти структуру й поведінку на одному кроці Бек називає носінням двох капелюхів. Спершу хай працює, потім хай буде правильно: дійдіть до зеленого найпростішою зміною, а прибирайте вже тоді, коли тести проходять.

Забагато прибирання або передчасна абстракція. Прибирати приємно, і це може здаватися безпечнішим, ніж братися за наступний тест. Зупиніться, щойно код достатньо добрий для цього сеансу, і не чіпайте двох схожих фрагментів коду, доки шаблон не стане очевидним. Бек формулює це коротко: «Дублювання — це підказка, а не наказ».

Тести знають реалізацію. Тести, які викликають приватні методи або перевіряють, які виклики зробив код, а не що він видав, ламаються щоразу, коли змінюється внутрішня будова. Erik Kuefler у розділі про модульне тестування з книжки *Software Engineering at Google* називає такі тести крихкими: вони падають через сторонню зміну, яка не вносить справжньої помилки. У TDD це обходиться вдвічі дорожче, бо крок рефакторингу спирається на тести, що лишаються зеленими, поки внутрішня будова змінюється. Тестуйте через публічний інтерфейс, перевіряйте стан, у якому код усе залишає, а не взаємодії, і називайте кожен тест за поведінкою, яку він перевіряє.

Що почитати далі

- Canon TDD — власний виклад процесу від Кента Бека, 2023 рік: п’ять кроків, поділ між дизайном інтерфейсу та дизайном реалізації і помилки на кожному кроці.
- Simple Smalltalk Testing: With Patterns — стаття Бека про простий фреймворк тестування для Smalltalk із фікстурами, тестовими випадками й наборами тестів, в архівній копії.
- Unit Testing — розділ, який написав Erik Kuefler для книжки *Software Engineering at Google*, про тести, що переживають рефакторинг, тестування через публічні API і тестування поведінки.
- Get Started — вступ до pytest: встановлення, перший тест, що падає, разом зі звітом, і те, як pytest знаходить файли з тестами.
- unittest: Unit testing framework — фреймворк зі стандартної бібліотеки Python, документація якого називає статтю Бека оригінальним описом патерну, спільного з unittest.

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 розробка через тестування?» 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/test-driven-development/