Engineer

Рефакторинг

🛠 зміна будови коду без зміни того, що він робить

Що це таке

Рефакторинг — це зміна внутрішньої структури коду без зміни того, що код робить для будь-кого, хто ним користується. Змінюються назви, межі функцій і поділ на модулі; вхідні дані, вихідні дані й побічні ефекти лишаються точно такими, як були. Окрему таку зміну теж називають рефакторингом. Англійською це називають refactoring.

Мартін Фаулер дає визначення цього терміна у своїй книжці Refactoring і повторив його на своєму сайті 2004 року. Як іменник рефакторинг — це зміна внутрішньої структури програмного забезпечення, яка робить його легшим для розуміння й дешевшим для змін, не змінюючи його спостережуваної поведінки. Як дієслово (to refactor) воно означає перебудовувати програмне забезпечення, застосовуючи низку таких змін, знову ж таки не змінюючи його спостережуваної поведінки. Важливі обидві половини. Мета — дешевша наступна зміна, а обмеження в тому, що ніхто поза кодом не може помітити, що щось сталося.

Фаулер чітко проводить межу: «Рефакторинг — не просто інша назва прибирання коду», пише він на refactoring.com. Це назва однієї конкретної техніки. Ширшим словом для реорганізації коду будь-якими засобами, зокрема такими, що не зберігають поведінку на кожному кроці, він вважає реструктуризацію (restructuring). Техніку з рефакторингу робить розмір кроків. Кожен крок — невелике перетворення, яке саме по собі робить мало й навряд чи піде не так, а система залишається повністю робочою після кожного з них. Довга послідовність таких кроків усе одно може скластися у велику зміну структури.

Кроки мають назви. Каталог на refactoring.com, що супроводжує друге видання книжки, перелічує їх разом зі старішими назвами, які вони замінили: прийом Extract Function, раніше Extract Method, і обернений до нього Inline Function; Inline Variable; Rename Variable; Move Function; Split Phase і ще десятки. Спільна назва дає розробникові змогу двома словами сказати, якою була зміна, а тому, хто переглядає код, — звірити її з відомим рецептом.

Застосуйте поняття до рішення

Застосуйте «Що таке рефакторинг?» 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

Успішне програмне забезпечення постійно змінюється. Нова функціональність і виправлення нашаровуються одне на одне, і кожна зміна зазвичай робить наступну трохи складнішою, доки нова робота, як описує Фаулер, не починає ледве повзти. Рефакторинг — це противага: він переформовує код так, щоб кожне доповнення не приносило складності, якої йому не треба. Віддачу вимірюють ціною наступної зміни, а не тим, як виглядає код.

Тому Фаулер вважає його частиною звичайного програмування, а не завданням, яке з’являється в плані проєкту. Він називає три моменти. Перш ніж додати нову функціональність, він перевіряє, чи має код таку форму, щоб її було легко додати, і якщо ні, спершу робить рефакторинг, що, як він зазвичай переконується, швидше, ніж втискати функціональність силоміць. Коли функціональність працює, він рефакторингом доводить результат до форми, яку без зайвих роздумів зрозуміє той, хто повернеться до коду за кілька тижнів. А коли йому доводиться читати незнайомий код, щоб його зрозуміти, він робить рефакторинг, щоб наступний читач не спотикався на тому самому місці. Перший із цих моментів має власну назву, підготовчий рефакторинг (preparatory refactoring), і Фаулер приписує Кенту Беку правило, яке за ним стоїть: для кожної зміни спершу перебудуйте код так, щоб зміна стала легкою, а це може бути найважчою частиною, і лише потім зробіть її.

Рефакторинг також тримає ризик низьким так, як цього не може робота над новою функціональністю. Фаулер переказує ще одну метафору Бека — два капелюхи. У капелюсі рефакторингу кожна зміна зберігає поведінку, а тести весь час лишаються зеленими. Коли ж додають функціональність, з'являються нові тести, а наявні ламаються, і це напруженіше й ризикованіше. Розробник, який знає, який капелюх на ньому, знає й те, що означає тест, який не проходить: під час рефакторингу це означає, що останній крок був хибним.

Саме так здебільшого й повертають технічний борг. Стаття Google про прихований борг у системах машинного навчання першим серед способів гасити борг називає рефакторинг коду. Інструменти допомагають: для багатьох мов тепер є редактори, які автоматизують поширені рефакторинги. Фаулер вважає такі інструменти цінними, але не обов'язковими, а там, де їх немає, покладається на малі кроки й часті тести.

Як це виглядає на практиці

Візьмімо функцію, написану для цієї статті, яка підсумовує замовлення й друкує чек в одному місці:

```
function printReceipt(order) {
  let total = 0;
  for (const line of order.lines) {
    total += line.price * line.quantity;
  }

  // вивести рядки чека
  console.log(`Order ${order.id}`);
  console.log(`Total: ${total}`);
}
```

Надходить нова вимога: та сама сума має з'явитися в рахунку. Замість того щоб копіювати цикл, розробник спершу робить рефакторинг. Перший крок — Extract Function: цикл переходить в окрему функцію, названу за тим, що вона обчислює, і запускаються тести. Другий крок так само виокремлює друк, тож назва функції замінює коментар, і тести запускаються знову. Третій крок — Inline Variable: змінна `total` тепер використовується лише раз, тож її місце займає виклик. Після кожного кроку програма робить точно те саме, що й раніше, і кожен крок міг би стати окремим комітом:

```
function printReceipt(order) {
  printLines(order.id, orderTotal(order));
}

function orderTotal(order) {
  let total = 0;
  for (const line of order.lines) {
    total += line.price * line.quantity;
  }
  return total;
}

function printLines(id, total) {
  console.log(`Order ${id}`);
  console.log(`Total: ${total}`);
}
```

Лише тепер суму додають до рахунку, і для цього потрібен один рядок у коді рахунків, який викликає `orderTotal` . Жоден із трьох кроків рефакторингу не потребував нового тесту, бо нічого спостережуваного не змінилося; доводять це саме ті тести, що вже покривали чек.

Рефакторинг роблять і в інфраструктурному коді, і там ставки вищі, бо назви вказують на ресурси, які працюють. Документація Terraform попереджає, що зміна адреси ресурсу в конфігурації за замовчуванням є вказівкою знищити наявний ресурс і створити новий. Тож перейменування `aws_instance.a` на `aws_instance.web` заради читабельності замінило б сервер. Починаючи з Terraform v1.1, блок `moved` натомість фіксує стару адресу, і наступний план оновлює адресу, нічого не знищуючи:

```
resource "aws_instance" "web" {
  # (конфігурація без змін)
}

moved {
  from = aws_instance.a
  to   = aws_instance.web
}
```

Для старіших версій документація радить для такого самого переміщення команду `terraform state mv` . Так чи інакше, рефакторинг безпечний лише тоді, коли план прочитано і він показує переміщення, а не знищення й створення.

Такі зміни легше тримати окремо від роботи над новою функціональністю, коли історія показує, що є що. Стандарт комітів, описаний у записі в портфоліо на цьому сайті про те, як привести чотири репозиторії до одного стандарту історії, вимагає починати кожне повідомлення коміту з конвенційного префікса, що вказує тип зміни, а хук для повідомлень комітів забезпечує дотримання цього правила. Запис, що описує роботу, виконану в липні 2026 року, налічує в історії найбільшого репозиторію 26 комітів рефакторингу поряд із 156 комітами з новими функціями та 133 виправленнями. Він також чесно визнає, що хук перевіряє формат, а не зміст. Коміт, позначений як рефакторинг, усе одно має довести тестами й переглядом коду, що поведінку він не зачепив.

Де виникають проблеми

Зміни поведінки проскакують разом із рефакторингом. Розробник упорядковує функцію і в тому самому коміті виправляє в ній помилку. Коли тест падає, ніхто не може сказати, що з двох це спричинило, і тому, хто переглядає код, доводиться читати кожен рядок як можливу зміну поведінки. Носіть один капелюх за раз: рефакторинг в одному коміті, зміна поведінки в наступному, а якщо тест падає під час рефакторингу, скасуйте останній крок, замість того щоб латати його далі.

Велику реструктуризацію називають рефакторингом. Гілка тижнями живе зі зламаною збіркою, поки код нібито «проходить рефакторинг». Це реструктуризація в термінах Фаулера, і вона несе саме той ризик, якого рефакторинг покликаний уникати. Розбийте роботу на кроки, кожен з яких залишає систему робочою і може бути злитий окремо; де це неможливо, назвіть це переписуванням і плануйте відповідно.

Немає тестів, які показали б, що нічого не змінилося. Збережена поведінка — це твердження, а без тестів — здогадка. Перш ніж торкатися коду без тестів, напишіть тести, які фіксують, що він робить сьогодні, навіть там, де це виглядає неправильно, а дивацтва виправте окремими змінами згодом. Тут діє емпіричне правило Фаулера з його прикладу підготовчого рефакторингу: щоразу, коли виникає бажання запустити код і перевірити результат на око, напишіть натомість тест.

Перейменування в інфраструктурному коді замінює ресурс. Охайна нова назва в Terraform під час перегляду коду виглядає як рефакторинг, а план тихо пропонує знищити базу даних і створити порожню. Фіксуйте кожне

перейменування і кожне переміщення в модуль чи з модуля блоком `moved`, а будь-яке знищення в плані зміни, задуманої як рефакторинг, вважайте невдалим кроком.

Старий блок `moved` видаляють. Блоки moved накопичуються, і їх видалення виглядає як звичайне прибирання. Документація Terraform називає це зміною, що порушує сумісність: будь-яка конфігурація, яка досі має стару адресу, тоді спланує видалення наявного об'єкта замість його переміщення. Зберігайте історичні блоки, вибудовуйте їх у ланцюжок, коли об'єкт перейменовують двічі, і видаляйте їх лише з приватного модуля, коли всі, хто ним користується, виконали apply з версією, що їх містить.

Рефакторинг чекає на окремий етап. Команда, що відкладає рефакторинг на окремий період десь згодом, і далі додає функціональність до коду, який їй опирається, а період, запланований на потім, легко викреслити, коли зсуваються терміни. Фаулер радить протилежне: робіть рефакторинг як частину кожної зміни, спершу щоб зробити зміну легкою, а потім щоб залишити результат зрозумілим, тож ця робота ніколи не потребує окремого місця в плані.

Що почитати далі

- Definition Of Refactoring — визначення Мартіна Фаулера для іменника й дієслова, як він подає їх у своїй книжці.
- Refactoring — сайт Фаулера до книжки: чому рефакторинг є частиною щоденного програмування, і повний каталог.
- Extract Function — запис каталогу про один із найпоширеніших рефакторингів, з оберненим до нього прийомом і назвою з першого видання.
- An example of preparatory refactoring — Фаулер малими кроками перебудовує власний код, перш ніж додати нову функціональність, і пояснює два капелюхи.
- Refactor modules — документація Terraform про блоки `moved`, перейменування, поділ модулів і про те, чому старі блоки варто залишати.

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 рефакторинг?» 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/refactoring/