Що таке рефакторинг і як не займатися дурницями

Рефакторинг — це процес удосконалення внутрішньої структури програмного коду без зміни його зовнішньої поведінки. Основна мета — підвищення читабельності, зрозумілості, підтримуваності та ефективності коду, зменшення його складності й усунення технічної заборгованості.

Рефакторинг — це не дебагінг і не реінжиніринг.

Коли не слід займатися рефакторингом

Якщо ви працюєте з кодом, що містить багато помилок або не покритий тестами, займатися його рефакторингом небезпечно. За визначенням, рефакторинг не повинен змінювати поведінку програми, тому всі баги, які були до рефакторингу, мають залишитися. Це означає, що виправлення помилок і рефакторинг — це різні процеси.

Якщо в коді багато невідомих помилок, спочатку потрібно забезпечити його стабільність і покриття тестами, і лише потім — рефакторити.

Саме тому Мартін Фаулер наголошує: рефакторити слід лише той код, який покрито автоматичними тестами. Це дозволяє гарантувати, що після змін функціональність не порушиться.

Кент Бек, автор методології екстремального програмування та TDD (розробка через тестування), підкреслює:

Спочатку реалізуйте робочий функціонал, який проходить всі тести та відповідає вимогам, а вже потім проводьте рефакторинг.

Різниця між рефакторингом і реінжинірингом

Мартін Фаулер також пропонує практичне правило:

Якщо під час “рефакторингу” частина функціоналу не працює тривалий час, а його відновлення вимагає значних зусиль, — це вже не рефакторинг, а реінжиніринг (тобто створення заново).

Аналогія:

Рефакторинг — це прибирання кімнати: перестановка меблів, зміна освітлення, організація простору.

Реінжиніринг — це повна перебудова будинку: нове планування, інші матеріали, зміна фундаменту.

Реінжиніринг застосовується тоді, коли система застаріла, не масштабується або не відповідає поточним вимогам. Він вимагає значних ресурсів і супроводжується ризиками — зокрема, ефектом другої системи (second-system effect), коли нова версія системи стає занадто складною, надмірною і непрактичною.

Основні методи рефакторингу

Найпоширеніші техніки:

- Перейменування змінних для покращення зрозумілості.
- Винесення логіки у функції, класи або окремі модулі.
- Видалення неактуального або неактивного коду (dead code).
- Прибирання застарілих або надмірних коментарів.
- Об’єднання дублікатів (усунення повторюваного коду).
- Спрощення складних умов або вкладеностей.

Загалом, імена функцій мають бути дієсловами (наприклад, getUserData()), а змінних — іменниками (user, sessionData). Для форматування імен часто використовують стилі camelCase, snake_case або угорську нотаці (Hungarian notation) — залежно від стандартів проєкту.

Code smells vs антипатерни

Слід розрізняти "code smells" (ознаки проблемного коду) та антипатерни.

Code smell — це симптом, що вказує на можливу проблему у структурі коду, але сам по собі не є помилкою.
Антипатерн (антишаблон) — це погане архітектурне рішення, яке призводить до складнощів у підтримці, масштабуванні або тестуванні системи. Це те, як не треба робити.

Приклади "code smells":

- Повторення одного й того ж коду.
- Неконсистентне іменування змінних (одночасно різні стилі найменування).
- Застосування суперечливих підходів до вирішення однієї задачі.
- Dead code — непотрібний або недосяжний код, який не використовується.

Під час рефакторингу:

- Виправляються як code smells, так і антипатерни (за наявності достатнього розуміння і контексту).
- Пріоритет — підтримка робочого функціоналу та покращення якості коду без порушення його логіки.

Висновки

Рефакторинг — це важливий етап розробки, який покращує якість коду, але не повинен змінювати функціональність.
Не варто рефакторити нестабільний або нетестований код.
Завжди переконайтеся, що у вас є тести, які гарантують збереження поведінки системи.
Не плутайте рефакторинг з реінжинірингом — це різні підходи з різними цілями та масштабами.
Пам’ятайте: робочий код із технічним боргом краще, ніж ідеальний, але неробочий.

Поганий приклад:

Ви приєднуєтесь до проєкту, але команда вам не довіряє. Ваше перше завдання — це рефакторинг, але немає задокументованих тестових випадків. Ви в пастці — без тестів ви не можете бути впевненими, що ваші зміни не зламають існуючу функціональність. Ретроспективне тестування завжди повинно бути проведене після рефакторингу. Далі вам призначають велику і складну задачу. Знову ж, у вас проблеми. Завдання надто велике, щоб його ефективно виконати. Правильна розробка починається з поділу завдання на чітко визначені підзадачі з ясними назвами та короткими описами. Тільки після цього команда може точно оцінити завдання.

Декомпозиція завдань та оцінка повинні бути колективною відповідальністю, а не залишатися на плечах одного розробника. Хтось має представити завдання, і команда повинна зробити попередню грубу оцінку. Потім має бути проведена правильна оцінка—можливо, за допомогою планування у вигляді покеру з відносними одиницями, такими як бали історій (сторі пойнти). Якщо завдання важко оцінити, команда повинна провести дослідження, спайки (spike) або додаткові уточнення перед тим, як дати оцінку.

Уявіть, що ви намагаєтесь оцінити велике завдання наодинці, без допомоги. Згодом, коли виникають проблеми, скептична команда звинувачує вас за неправильно оцінену складність. Це не просто індивідуальна помилка, а проблема процесу.

Щоб уникнути цього, тестові випадки повинні бути доступні до рефакторингу. Завдання (історії користувачів) повинні бути маленькими, тестованими та незалежними. Оцінка має бути командною роботою, оскільки розробка програмного забезпечення — це не індивідуальний вид спорту. Ми вимірюємо успіх за результатами команди, а не тільки однієї людини. Завдання включають кілька аспектів — тестування, комунікацію,

проєктування, інтеграцію — що робить їх спільною відповідальністю. Як команда може довіряти своїм членам, якщо вона не має структурованого способу оцінки їх роботи?

Джерела:

1. Martin Fowler — Refactoring: Improving the Design of Existing Code,
2. Robert C. Martin — Clean Code,
3. Kent Beck — Test-Driven Development: By Example,
4. Kent Beck — Extreme Programming Explained,
5. Andrew Hunt, David Thomas — The Pragmatic Programmer.