**Лекція 11. Поняття та методи рефакторингу.**

**Рефакторинг** (англ. *refactoring*) — перетворення програмного коду, зміна внутрішньої структури програмного забезпечення для полегшення розуміння коду і легшого внесення подальших змін без зміни зовнішньої поведінки самої системи. Слово «рефакторинг» пішло від терміну «факторинг» в структурному програмуванні, який означав декомпозицію програми на максимально автономні та елементарні частини.

**Причини рефакторингу**

Існує міф про те, що правильно організований процес розробки продукту методично дотримується поставлених вимог, визначає однозначний, стабільний список обов’язків програми, і при цьому програмний код може бути написаний майже лінійно — від початку до закінчення, кожна ділянка — один раз написана, відтестована і забута. Згідно з цим міфом, єдиний випадок, коли наявний код може змінюватись, — це в процесі підтримки і адміністрування програми, коли початкова версія продукту уже здана замовнику.

Однак реальність є дещо інакшою. Насправді код еволюціонує в процесі розробки продукту. Як правило, кодування, відлагодження та модульне тестування займають в середньому 30–65% зусиль від загального часу існування проекту (залежно від величини проекту). Навіть на добре організованих проектах вимоги змінюються в середньому на 1–4% за місяць, що неминуче спричиняє зміни до програмного коду — як дрібні, так і досить серйозні.

Також, на відміну від старіших методик розробки програмних продуктів, де основний акцент ставився на мінімізації змін до коду, сучасна методика вбачає великий потенціал у внесенні змін. Вона є більш сфокусованою на коді (code-centered) і під час розробки можна очікувати, що код буде вдосконалюватися більше, ніж зазвичай.

**Підстави для проведення рефакторингу**

1. **Код дублюється.**

*Це нерідко призводить до необхідності вносити паралельні зміни до кількох скопійованих ділянок коду одночасно (що не відповідає принципам «DRY» (Don't Repeat Yourself) та «Copy And Paste is a Design Error»).*

**Дублювання коду** — термін відомий із програмування, під яким розуміється, що деякі ділянки початкового коду можуть зустрічатися більш, ніж один раз, як всередині однієї, так і в всередині декількох програм. Дублювання є ознакою так званого «поганого коду» або «коду з запахом» (англ. *Code smell*), тому що саме через дублювання збільшується довжина коду. Послідовності дублікатів іноді називають *клонами*.

**Причини за яких дві частини коду можуть вважатися дублікатами**

* Посимвольний збіг;
* Посимвольний збіг, з ігноруванням пробільних символів і коментарів;
* Збіг за лексемами;
* Частковий збіг за лексемами;
* Функціональний збіг;

**Звідки з'являються дублікати коду**

Причини виникнення дублікатів коду:

* Програмування копіюванням-вставленням (копіпаст), при якому ділянки коду копіюються через те, що «це працює». У більшості випадків така операція вимагає невеликих змін перенесеного коду, наприклад перейменування змінних або додавання/видалення окремих ділянок.
* Бажана функціональність дуже схожа на вже наявну в іншій частині програми, і програміст створює код дуже близький до того, який вже існує.
* Плагіат, коли код просто копіюється без дотримання прав або будь-яких умов.

**Проблеми до яких призводять дублікати коду**

Дублювання коду є ознакою низького стилю програмування. Гарний стиль програмування звичайно заснований на повторному використанні коду. Може здаватися, що використання дублікатів дозволить дещо прискорити процес створення програми, так як програмісту не потрібно буде думати над тим, як код використовується і як він може використовуватися надалі. Однак проблема полягає в тому, що написання коду це лише невелика частина життєвого циклу продукту, і подальший супровід коду з дублікатами буде занадто ускладненим. Ось кілька проблем до яких призводить дублювання коду:

* **Велика кількість коду ускладнює його розуміння**: дублювання коду часто призводить до створення довгих, повторюваних послідовностей коду які відрізняються лише кількома рядками або символами.
* **Приховане значення**: важко вловити різницю в повторюваних ділянках коду, і тому стає важче розуміти для чого саме призначена та чи інша частина коду. Найчастіше, єдина різниця полягає в параметрах. У цій ситуації найкраще використовувати процедури і функції.
* **Аномалії оновлення**: дублювання коду суперечить основному принципу теорії баз даних: *«Уникайте надмірності»*. Невиконання цього принципу призводить до аномалій оновлення, які сильно збільшують витрати на обслуговування коду. У цьому випадку одну і ту ж зміну потрібно ввести в усі дублікати. І в кращому випадку, час витрачений на внесення змін і тестування коду збільшується пропорційно кількості дублікатів. А в гіршому — деякі місця в коді можуть бути пропущені, і виправлення всіх помилок може зайняти місяці або навіть роки. Намагайтеся використовувати бібліотеки коду у такій ситуації.
* **Розмір файлу**: без застосування будь-якого стиснення, файл початкового коду займатиме більше місця на твердому диску.

**Пошук дублікату коду**

Існує певна кількість алгоритмів які дозволяють відшукати дублікати коду. Наприклад:

* Алгоритм Бейкера
* Алгоритм Рабіна — Карпа.
* Використання абстрактних синтаксичних дерев.
* Візуальне визначення дублікатів.
* Зчитування матриці клонування.

1. **Підпрограма занадто довга.**

*Хоча питання, яку максимальну довжину може мати підпрограма, є досить суперечливим, однак загальноприйнятим неофіційним стандартом є написання підпрограм довжиною не більше, ніж один екран коду.*

1. **Цикл занадто довгий, або рівень вкладеності тіла циклу занадто великий.**
2. **Клас має багато обов'язків, слабо пов'язаних між собою.**

*В такому випадку краще розділити клас на кілька атомарних класів.*

1. **Інтерфейс класу не забезпечує достатній рівень абстракції.**
2. **Функція має занадто багато параметрів.**
3. **Потрібно одночасно змінювати кілька паралельних ієрархій класів.**

*Для вирішення цієї проблеми можна, наприклад, скористатися шаблоном «Міст»* (англ. *Bridge*) — шаблон проектування, призначений для того, щоб відділити [абстракцію](https://uk.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%86%D1%96%D1%8F_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85) від її конкретної [імплементації](https://uk.wikipedia.org/wiki/%D0%86%D0%BC%D0%BF%D0%BB%D0%B5%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D1%96%D1%8F) таким чином, щоб вони могли бути змінені незалежно один від одного. Належить до класу структурних шаблонів.

1. **Споріднені дані, які використовуються разом, не організовані в клас.**
2. **Клас не виконує ніяку роботу самостійно, а тільки передоручає обов'язки іншим класам.**
3. **Назва класу чи методу має ім'я, яке недостатньо точно відповідає змісту.**
4. **Клас має занадто багато відкритих (public) членів.**
5. **Нестатичний клас складається тільки з даних або тільки з методів.**
6. **В ланцюжку викликів методів передається багато зайвих даних.**
7. **Занадто поширене використання глобальних змінних.**

**Прийоми рефакторингу**

1. Прийоми, що дозволяють розбити код на дрібніші, зрозуміліші частини.
   * Відокремлення методу (Extract Method).

**Відокре́млення ме́тоду** ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *Extract Method*) - прийом [рефакторингу](https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3), що дозволяє розбити код на дрібніші та зрозуміліші частини, прибрати [дублювання коду](https://uk.wikipedia.org/wiki/%D0%94%D1%83%D0%B1%D0%BB%D1%8E%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%BA%D0%BE%D0%B4%D1%83) і полегшити подальшу роботу з ним. Чим більше коду в [методі](https://uk.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4), тим складніше розібратися в тому, що він робить. Це основна проблема, яку вирішує даний [рефакторинг](https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3).

**Переваги**

1. Покращує читабельність коду.
2. Прибирає [дублювання коду](https://uk.wikipedia.org/wiki/%D0%94%D1%83%D0%B1%D0%BB%D1%8E%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%BA%D0%BE%D0%B4%D1%83). Іноді код, винесений в метод, можна знайти і в інших місцях програми. У такому разі є сенс замінити знайдені ділянки коду викликом нового методу.
3. Ізолює незалежні частини коду, зменшуючи вірогідність помилок. (наприклад, з вини перепризначення не тієї змінної).

**Порядок рефакторингу**

1. Створити новий [метод](https://uk.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4). Підібрати назву яка відображає суть того, що робитиме цей метод.
2. Скопіювати фрагмент коду, що цікавить, в новий метод. Видалити цей фрагмент із старого місця і замінити викликом нового методу.
3. Знайти усі змінні, які використовувалися в цьому фрагменті коду. Якщо вони були оголошені всередині цього фрагменту і не використовуються поза ним, залишити їх без змін — вони стануть локальними змінними нового методу.
4. Якщо змінні оголошені перед ділянкою коду, то їх слід передати в параметри нового методу, щоб використати значення, які в них знаходилися раніше. Іноді від таких змінних простіше позбавитися за допомогою заміни змінних викликом методу.
5. Якщо локальна змінна якось змінюється у ділянці коду, це може означати, що її змінене значення знадобиться далі в основному методі, якщо це дійсно так, то значення цієї змінної слід повернути в основний метод.

**Анти-рефакторинг**

* Вбудовування методу

**Схожі рефакторинги**

* Переміщення методу

**Допомагає іншим рефакторингам**

* Заміна параметрів об'єктом
* Створення шаблонного методу
* Параметризація методу
  + Відокремлення базового класу (Extract Superclass).

**Відокрéмлення клáсу** ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *Extract class*) — прийом рефакторингу, що полягає в розділенні класу та виділенні полів і методів, що підтримують окремий функціонал, у новостворений клас з метою полегшення роботи з даними.

**Причини рефакторингу**

У ході написання програми класи можуть отримати масу додаткових обов'язків.

**Переваги здійснення відокремлення класу**

1. Функція такого рефакторингу — сприяння дотримання принципу єдиного обов'язку класу. В результаті код класів стає більш чистим і зрозумілим.
2. Класи з єдиним обов'язком більш стійкі до змін. Наприклад, якщо є клас, який відповідає за 10 різних речей, і до нього потрібно внести певні зміни, то змінюючи один елемент є ризик пошкодити інші.

**Недоліки**

Якщо часто проводити такий рефакторинг, потрібно буде вдаватися до антирефакторингу — вбудовування класу.

**Порядок рефакторингу**

1. Створити новий клас, який міститиме виділену функціональність.
2. Створити зв'язок між старим і новим класом. Найкраще, якщо цей зв'язок буде одностороннім; при цьому другий клас можна буде без проблем використати повторно. З іншого боку, за необхідністю, завжди можна створити двосторонній зв'язок.
3. Використати переміщення поля і переміщення методу для кожного поля і методу, які потрібно перенести в новий клас. Для методів слід розпочинати з приватних, таким чином знижуючи вірогідність допустити масу помилок. Задля полегшення процесу виправлення помилок потрібно проводити тестування після кожного переміщення, аби не отримати багато помилок в кінці роботи.
4. Після переміщення потрібно подивитись на отримані класи. Можливо, класи потрібно буде перейменувати, зважаючи на їх нові обов'язки. Також варто перевірити, чи можна позбавитися від двостороннього зв'язку між класами, якщо він з'явився.
5. Важливим моментом є доступність класу ззовні: можна повністю сховати клас, зробивши приватним, і в той же час управляти його полями із старого класу, або зробити публічним, надавши клієнтові можливість безпосередньо міняти значення. Рішення залежить від того, наскільки безпечні для поведінки старого класу будуть несподівані прямі зміни значень в новому класі.
6. Прийоми, що дозволяють забезпечити додаткову абстракцію.
   * *Інкапсуляція поля* (Encapsulate Field) — замінює прямий доступ до поля на доступ через методи-аксесори (або властивості в C#).
   * *Узагальнення типу* (Generalize Type) — заміна типів, з якими працює клас, на більш узагальнені.
   * *Заміна блоків* перевірки типів на шаблони «Стан» (State) або «Стратегія» (Strategy).

**Стан** (англ. *state*) — шаблон проектування (належить до [шаблонів поведінки](https://uk.wikipedia.org/wiki/%D0%A8%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D0%B8_%D0%BF%D0%BE%D0%B2%D0%B5%D0%B4%D1%96%D0%BD%D0%BA%D0%B8)), що реалізує скінченний автомат в обʼєктно-орієнтованому програмуванні.

Він реалізується шляхом створення для кожного стану скінченного автомата класу-спадкоємця інтерфейсу (або абстрактного класу) та дозволяє об'єктові варіювати свою поведінку залежно від внутрішнього стану.

**Стратегія** ([англ.](https://uk.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D1%96%D0%B9%D1%81%D1%8C%D0%BA%D0%B0_%D0%BC%D0%BE%D0%B2%D0%B0) *Strategy* ) — шаблон проектування, належить до класу шаблонів поведінки. Відомий ще під іншою назвою — «Policy». Його суть полягає у тому, щоб створити декілька схем поведінки для одного об'єкту та винести в окремий клас. Шаблон Стратегія (Strategy) дозволяє міняти вибраний алгоритм незалежно від об'єктів-клієнтів, які його використовують.

* + Заміна умовних операторів поліморфізмом.
  + Створення поля або локальної змінної (Introduce Field/Introduce Local Variable).
  + Дублювання видимих даних

**Дублювання видимих даних** - (англ. duplicate observed data) це один з прийомів [рефакторингу](https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3), який застосовується у випадках, коли потрібно розділити дані наприклад [GUI](https://uk.wikipedia.org/wiki/Graphical_user_interface) та предметної області.

**Проблема**

Дані предметної області програми зберігаються в класах, що відповідають за призначений для користувача інтерфейс (GUI).

**Рішення**

Має сенс виділити дані предметної області в окремі класи і, таким чином, забезпечити зв'язок і синхронізацію між класом предметної області і GUI.

**Причини рефакторингу**

Ви хочете мати декілька видів інтерфейсу для однакових даних (наприклад, у вас є додаток не лише для [десктопа](https://uk.wikipedia.org/wiki/%D0%94%D0%B5%D1%81%D0%BA%D1%82%D0%BE%D0%BF), але також для телефонів і планшетів). В цьому випадку вам буде дуже складно уникнути великої кількості помилок і дублювання коду, якщо ви не розділите GUI і предметну область.

**Переваги**

1. Ви розділяєте відповідальність між класами бізнес-логіки і представлення (принцип єдиного обов'язку), що спрощує читабельність і розуміння програми в цілому.
2. Якщо потрібно буде додати новий вид інтерфейсу, вам треба буде створити нові класи представлення, при цьому код бізнес-логіки чіпати немає ніякої нужди (принцип відкритості/закритості).
3. Над бізнес-логікою і призначеними для користувача інтерфейсами тепер можуть працювати різні люди.

**Коли його не можна застосувати**

Цей рефакторинг, який в класичному виконанні здійснюється з введенням шаблону Спостерігач, практично не застосовується для веб-додатків, де всі класи перестворюються при кожному запиті до веб-сервера.

Проте, загальний принцип відокремлення бізнес-логіки в окремі класи має сенс, у тому числі, і для веб-додатків. Але реалізується він за допомогою інших рефакторингів, які обираються з урахуванням дизайну вашої системи.

**Порядок рефакторингу**

1. Необхідно приховати прямий доступ до даних предметної області в класі GUI, для чого найкраще використати самоінкапсуляцію поля. Таким чином, ви створите геттери і сеттери до цих даних.
2. У обробниках подій *класу GUI використайте сеттери для установки нових значень полів. Це дасть можливість передавати нові значення в пов'язаний об'єкт предметної області*.
3. Створіть клас предметної області і скопіюйте в нього необхідні поля з класу GUI. Для всіх цих полів створіть геттери і сеттери.
4. Застосуйте патерн Спостерігач до цих двох класів:
   * В *класі предметної області* створіть масив для зберігання об'єктів спостерігачів (об'єктів GUI), а також методи їх реєстрації, видалення і сповіщення.
   * В *класі GUI створіть поле для зберігання посилання на об'єкт предметної області*, а також метод update(), який реагуватиме на зміни в цьому об'єкті та буде оновлювати значення полів в класі GUI. Зверніть увагу, в методі оновлення значення повинні встановлюватися безпосередньо, щоб уникнути рекурсії.
   * У конструкторі *класу GUI створіть екземпляр класу предметної області* і збережіть його в створеному полі. Зареєструйте *об'єкт GUI* як спостерігач в *об'єкті предметної області*.
   * У сеттерах полів *класу предметної області* викликайте метод сповіщення спостерігача (тобто метод оновлення в *класі GUI*), щоб передати нові значення в призначений для користувача інтерфейс.
   * Змініть сеттери полів *класу GUI так, щоб вони тепер встановлювали нові значення в об'єкті предметної області, причому безпосередньо. Будьте уважні, якщо значення встановлюватимуться через сеттер класу предметної області*, це приведе до нескінченної рекурсії.
5. Прийоми, що змінюють назви членів та їх розташування.
   * Переміщення методу (Move Method) або переміщення поля в інші класи або файли коду.

**Перемі́щення ме́тоду** (англ. *Move Method*) - це прийом [рефакторингу](https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3), який базується на принципі перенесення методу з одного [класу](https://uk.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%29) в інший, який більше його використовує.

**Проблема**

Певний створений метод після його створення, почав використовуватися іншим класом більше, ніж у власним.

**Рішення**

Потрібно створити новий метод в класі, який використовує його більше за інших, в який перенесться код із старого методу. При цьому код оригінального методу потрібно перетворити на звернення до нового методу в іншому класі або прибрати його взагалі.

**Причини рефакторингу**

1. Рішення перемістити метод до класу, де знаходяться дані, з якими, в основному, і працює цей метод. Це призводить до *підвищення пов'язаності всередині класів*.
2. Рішення перемістити метод, щоб прибрати або зменшити залежність класу, що викликає цей метод, від класу, в якому він знаходився. Це може бути корисно у випадку, коли клас, що викликає, вже має залежність від того класу, куди планується його перенести. Таким чином, *зменшується зв'язаність між класами*.

**Порядок проведення**

1. Перевіряємо властивості, які використовував метод до початку рефакторингу. Якщо вони використовуються тільки самим методом, то потрібно перемістити і їх. Якщо ця властивість використовується ще якимось методами, то бажано перенести і ці методи. Адже налаштовувати зв’язок між цими методами у різних класах може бути проблематично.
2. Перевірити наявність інших оголошень вихідного методу у базових і похідних класах. Якщо вони є, то потрібно або відмовитися від ідеї перенесення, або реалізувати в цільовому класі (клас, до якого переносимо метод) подібність поліморфізму для забезпечення різної функціональності методу.
3. Оголосити метод, який переміщуємо в цільовому класі. За потреби змінити ім’я методу для кращої відповідності цільовиму класу.
4. Копіюємо код вихідного методу в цільовий, після чого пристосовуємо скопійований код під нове місце.
5. Якщо метод містить обробники виключних ситуацій, то слід визначити, в якому з класів - цільовому або вихідному - логічніше буде обробляти виключення.
6. Визначити як буде виконуватися звернення до переміщеного методу.
   1. За наявності поля чи методу, які повертають об’єкт, не вносимо додаткових змін. За відсутності – потрібно написати новий метод або поле, в якому б зберігався об’єкт класу з переміщеним методом.
   2. Можливо зробити з вихідного методу делегуючий, який не буде містити код, а просто буде викликати цільовий метод. Можливо залишити метод як делегуючий у вихідному класі або видалити, попередньо замінивши всі звернення до нього зверненнями до створеного в іншому класі методу.
   * Перейменування члена (Rename) — зміна імені, з автоматичною заміною всіх посилань на старе ім’я в коді.
   * Переміщення члену до базового/дочірнього класу (Pull Up/Push Down).
   * Розщеплення змінної

**Розще́плення змі́нної** (англ. *Split Temporary Variable*) — прийом [рефакторингу](https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D1%84%D0%B0%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%BD%D0%B3), що дозволяє зменшити складність всередині методу, зробити метод простим для розуміння і полегшити подальшу роботу з ним.

**Проблема**

Є локальна змінна, яка використовується для зберігання різноманітних значень всередині методу (не рахуючи змінних циклів).

Приклад C#:

double temp = 2 \* (height + width);

Console.WriteLine(temp);

temp = height \* width;

Console.WriteLine(temp);

**Рішення**

Використати різні змінні для різних значень. Кожна змінна повинна відповідати тільки за одну певну річ.

Приклад C#:

readonly double perimeter = 2 \* (height + width);

Console.WriteLine(perimeter);

readonly double area = height \* width;

Console.WriteLine(area);

**Причини рефакторингу**

Якщо в процесі написання коду було «зекономлено» змінні всередині функції і повторно використано їх для різноманітних непов'язаних між собою цілей, то обов'язково почнуться проблеми в той момент, коли потрібно буде внести якісь оновлення в код, що містить ці змінні. Доведеться перевірити декілька разів усі випадки використання змінної, щоб впевнитись у відсутності помилки в коді.

**Переваги**

1. Кожен елемент програми повинен відповідати тільки за одну річ. Це значно спрощує підтримку коду в майбутньому, оскільки можна спокійно змінити цей елемент, не побоюючись побічних ефектів.
2. Покращується читабельність коду. Якщо змінна створювалася дуже давно, та ще і в поспіху, вона могла дістати елементарну назву, яка не пояснює суті значення, що зберігається, наприклад, k, n, value і так далі. Є шанс виправити ситуацію, призначивши новим змінним хороші назви, що відображають суть значень, що зберігаються. Наприклад, customerTaxValue, cityUnemploymentRate, clientSalutationString і так далі.
3. Цей рефакторинг допомагає надалі виділити ділянки коду, що повторюються, в окремі методи.

**Порядок рефакторингу**

1. Потрібно знайти місце в коді, де змінна вперше заповнюється якимось значенням. У цьому місці перейменуйте цю змінну, причому нова назва повинна бути зрозумілою і відповідати її значенню.
2. Підставити її нову назву замість старої в місцях, де використовувалося це значення змінної.
3. Повторити операцію для інших випадків, де змінній присвоюється нове значення.

**Автоматизований рефакторинг**

Багато інтегрованих середовищ розробки містять вбудовані механізми рефакторингу коду. Крім інтегрованої функціональності, існує також багато продуктів сторонніх виробників, які, як правило, реалізовані у вигляді додатків (plugins) до відповідного IDE. Приклади таких пакетів:

|  |  |  |
| --- | --- | --- |
| **Пакет** | **Мова** | **Середовище** |
| [Microsoft Visual Studio](https://uk.wikipedia.org/wiki/Microsoft_Visual_Studio) | [C#](https://uk.wikipedia.org/wiki/C_Sharp) | Microsoft Visual Studio (вбудований) |
| [Java](https://uk.wikipedia.org/wiki/Java) Development Tooklit | [Java](https://uk.wikipedia.org/wiki/Java) | [Eclipse](https://uk.wikipedia.org/wiki/Eclipse) (вбудований) |
| [IntelliJ IDEA](https://uk.wikipedia.org/wiki/IntelliJ_IDEA) | Java | IntelliJ IDEA (вбудований) |
| [NetBeans](https://uk.wikipedia.org/wiki/NetBeans) | Java | NetBeans (вбудований) |
| [Visual Assist](https://uk.wikipedia.org/w/index.php?title=Visual_Assist&action=edit&redlink=1) | C#, [C++](https://uk.wikipedia.org/wiki/C%2B%2B), [VB](https://uk.wikipedia.org/wiki/Visual_Basic), [VB.NET](https://uk.wikipedia.org/wiki/Visual_Basic_.NET) | Microsoft Visual Studio |
| [Photran](https://uk.wikipedia.org/w/index.php?title=Photran&action=edit&redlink=1) | [Fortran](https://uk.wikipedia.org/wiki/Fortran) | Eclipse |

Рефакторинг – ефективний і потужний інструмент програмування, але як і всі потужні інструменти при неправильному використанні він може нанести шкоду. Тому при рефакторингу потрібно користуватись наступними прийомами:

o Збереження початкового коду.

o Обмеження об’єму окремих видів рефакторингу.

o Виконання окремих видів рефакторингу по одному за раз.

o Складання списку дій, які програміст збирається виконати.

o Складання і підтримка списку видів рефакторингу, які потрібно виконати пізніше.

o Часте створення контрольних точок.

o Використання попереджень компілятора.

o Виконання регресивного тестування.

o Створення додаткових тестів.

o Виконання оглядів змін.

o Зміна підходу в залежності від ризикованості рефакторингу.

**Стратегії рефакторингу**

Число видів рефакторингу, вигідних для конкретної програми, майже нескінчене. Рефакторинг підлягає тому ж закону зниження вигоди, що й інші процеси програмування і до нього теж можна застосувати правило 80/20. Тому доцільно витрачати час на 20% видів рефакторингу, які забезпечать 80% вигоди. При визначенні найважливіших видів рефакторингу варто:

o Виконувати рефакторинг при створенні нових методів.

o Виконувати рефакторинг при створенні нових класів.

o Виконувати рефакторинг при виправленні дефектів.

o Виконувати рефакторинг модулів, в яких велика ймовірність виникнення помилок.

o Виконувати рефакторинг складних модулів.

o При супроводженні програми покращувати фрагменти, які доводиться виправляти.

o Визначити інтерфейс між акуратним і поганим кодом та перенести поганий код на інший бік цього інтерфейсу.

# Рефакторинги рівня даних

o Замінити магічне число іменованою константою.

o Замінити ім’я змінної на більш зрозуміле та інформативне.

o Відмовитись від проміжної змінної, записавши вираз в рядку використання його значення.

o Замінити вираз викликом підпрограми.

o Ввести проміжну змінну.

o Замінити змінну, яка використовується багаторазово для різних цілей, групою змінних, кожна з яких використовується з однією метою.

o Використовувати локальні змінні для локальних цілей, а не параметри.

o Перетворити змінну базового типу в об’єкт.

o Перетворити набір значень в клас.

o Перетворити набір значень в клас з підкласами.

o Перетворити масив на об’єкт.

o Інкапсулювати колекцію.

o Замінити запис (структуру) на клас.

# Рефакторинги рівня операторів

o Провести декомпозицію логічного виразу.

o Перенести складний логічний вираз в добре іменовану булеву функцію .

o Об’єднати оператори, що знаходяться в різних частинах умовного оператора.

o Використовувати break чи return замість змінних, що служать для виходу з циклу.

o Виконувати return як тільки обчислено результат функції замість присвоєння значення, що повертається, спеціальній змінній.

o Заміняти умови поліморфізмом (особливо case-оператори).

o Створювати і використовувати null об’єкти замість перевірки на значення null.

**Рефакторинги рівня методів**

o Виокремити метод.

o Перенести код методу у місце виклику.

o Перетворити довгий метод у клас.

o Замінити простий алгоритм більш ефективним складнішим алгоритмом.

o Додати параметр.

o Вилучити параметр.

o Відділити операції читання від операцій запису.

o Об’єднати схожі методи, використавши параметри.

o Розділити методи, чия поведінка залежить від переданого параметра.

o Передавати як параметр цілий об’єкт, а не окремі поля.

o Передавати як параметр окремі поля, а не цілий об’єкт.

o Інкапсулювати приведення типу вниз.

**Рефакторинги рівня реалізації класу**

o Уникати багаторазового створення об’єктів шляхом використання вказівників на об’єкт, що розділяється декількома частинами програми (для великих, громіздких об’єктів).

o Уникати використання багатьох вказівників шляхом створення багатьох об’єктів (для малих об’єктів).

o Змінювати положення методів або даних в ієрархії.

o Виокремити спеціалізований клас в підклас.

o Скомбінувати схожий код в клас вищого рівня ієрархії.

**Рефакторинги рівня інтерфейсу класу**

o Перенести метод в інший клас.

o Розбити клас на два.

o Прибрати клас.

o Вилучити проміжний метод.

o Вилучити set()-методи для полів, які не можуть змінюватись.

o Приховувати методи, які не використовуються поза межами класу.

o Об’єднати клас-нащадок та клас-предок, якщо їх реалізація дуже подібна.

**Рефакторинги рівня системи**

o Робити копії даних, якими програміст не керує.

o Замінити односторонній зв’язок класів двостороннім.

o Замінити двосторонній зв’язок класів одностороннім.

o Створити фабрику об’єктів замість простого конструктора.

o Замінити коди помилок на виключні ситуації або навпаки.

**Рефакторинг. Виміри у конструюванні.**

Значна частка результатів, як і, власне, діяльності по конструюванню програмного забезпечення, може бути виміряною, в тому числі й кількісно. Це й вихідний код, що було розроблено, й модифікований об’єм коду, і степінь повторного використання, і багато інших характеристик. Ці виміри, які ще прийнято називати результатами аудиту коду та метрики коду, несуть велику користь як для оцінки ризиків та якості (зниження ризиків, підвищення якості), так і для управління конструюванням та програмними проектами в цілому. Про планування не може бути й мови, якщо ми не здатні передбачити ні тривалості робіт, ні вартості окремих задач, ні ймовірності виникнення дефектів проти заданих параметрів прийнятної якості.

Код являється одним із найбільш чітко детермінованих активів проекту (поступово такими стають і моделі, які будуються на основі структур метаданих, і тісно пов’язані з кодом – наприклад, UML). Також код є носієм тієї функціональності, яка вимагається. Відповідно, застосування вимірів у відношенні до коду стає тим самим інструментом, який впливає і на сам код. Застосування вимірів, зокрема, **метрик**, дозволяє визначити необхідність проведення **рефакторингу** коду, як методам його реструктуризації, що мають на меті без зміни змісту (тобто функціональності та функціональної цілісності) забезпечити рішення задач мінімізації складності, готовності до змін (гнучкості), прозорості документування та інших актуальних аспектів конструювання. Термін «рефакторинг» виник із слова «факторинг», яке у контексті структурного програмування означає максимально можливу декомпозицію програми на складові.

Для виконання рефакторингу існують певні попереджувальні знаки, які Мартін Фаулер ще називає «запахами» (smells):

1. **Код повторюється**, що говорить про неповну факторизацію системи на етапі проектування;

2. **Метод занадто великий**. Одним із способів покращення системи є підвищення її модульності – збільшення кількості добре визначених та іменованих методів, що успішно вирішують тільки одну задачу. Якщо здається, що після розділення одного методу на кілька код стає зрозумілішим, створюйте додаткові методи;

3. **Цикл занадто великий або глибоко вкладений в інші цикли**. Часто знизити складність циклу та краще факторизувати код допоможе перетворення тіла циклу в метод;

4. **Клас має погану зв’язність**, тобто має багато аспектів відповідальності, які ніяк не пов’язані між собою. Тоді краще розбити клас на декілька класів так, щоб кожний із них отримав зв’язний набір аспектів;

5. **Метод приймає надто багато параметрів.** Добре факторизовано програми включають багато невеликих, добре визначених методів, що не потребують велику кількість параметрів;

6. **Окремі частини класу змінюються незалежно від інших частин.** Якщо клас має різні області відповідальності й лише деякі зміни охоплюють його частини, краще розділити клас на кілька класів відповідно до окремих областей відповідальності;

7. **При зміні програми потрібно паралельно змінювати кілька класів.** Якщо при написанні програми вже не вперше змінюється один і той же набір класів, краще подумати, чи можна реорганізувати код цих класів, щоб зміни охоплювати лише один клас;

8. **Метод використовує більше елементів іншого класу, ніж своїх власних.** Метод потрібно перемістити в інший клас і викликати із старого;

9. **Клас має надто обмежену функціональність**, тобто не відповідає своєму найменуванню, подумайте, чи не можна видалити його, розділивши всі його аспекти відповідальності між іншими класами;

10. **По ланцюгу методів передаються бродячі дані (tramp data).** Тобто дані передаються в метод для того, щоб він передав їх іншому методу. Пошукайте спосіб, що дозволить покращити узгодженість інтерфейсу кожного методу;

11. **Об’єкт-посередник нічого не робить**, просто перенаправляє виклики методів до інших класів. Подумайте, чи можна такий метод усунути;

12. **Один клас надто багато знає про інший.** Намагайтесь зробити інкапсуляцію більш строгою;

13. **Метод має невдале ім’я;**

14. **Складний код пояснюється за допомогою коментарів.** Як зауважують Б. Керніган та Б. Плагер: не документуйте поганий код– перепишіть його;

15. **У коді присутні глобальні змінні.** Перегляньте код на можливість їх усунення;

16. **У програмі міститься код, який може ще знадобитися.** Думка експертів однозначна: не пишіть гіпотетично потрібний код, а підвищену увагу приділіть ясності та зрозумілості коду, що потрібен прямо зараз.

Значення слова «рефакторинг» досить розмите: так називають виправлення дефектів, реалізацію нової функціональності, модифікацію проекту – по суті, будь-яку зміну коду. Проте самі по собі, без ефективної стратегії зміни коду переваг не мають.

**Рефакторинг на рівні даних**:

· Якщо Ви використовуєте числовий або рядковий літерал, наприклад, 3.14, замініть його іменованою константою, такою як PI.

· Якщо ім’я змінної неінформативне, присвойте їй краще ім’я.

· Замінюйте проміжну змінну, якій присвоюється результат обчислення виразу, на сам вираз. Якщо ж така змінна все ж вводиться, назвіть її так, щоб ім’я резюмувало суть виразу.

· Заміна виразу на виклик методу зазвичай слугує усуненню з коду виразів, які повторюються.

· Якщо змінна використовується більш, ніж з однією ціллю (часто такими є i, j, temp, x), створіть під кожну ціль свою змінну та оберіть для неї більш визначене ім’я.

· Якщо вхідний параметр методу служить в якості локальної змінної, краще створювати для цього справжню локальну змінну.

· Якщо елементарний тип даних необхідно розширити додатковими формами поведінки (наприклад, більш строгим контролем типу) або додатковими даними, перетворіть його у клас і реалізуйте потрібну поведінку. Це відноситься й до простих числових типів на зразок Money чи Temperature, і до перечислень, таких як Color, Shape чи Country.

· Якщо різні елементи, що асоційовані з різними типами, можуть по-різному себе поводити, подумайте про створення базового класу для типу і похідних класів для кодів кожного типу.

· Якщо елементами масиву є різні типи, створюйте клас, що включатиме поля для кожного елемента масиву.

**Рефакторинг на рівні окремих операторів**:

· Спрощуйте логічні вирази, вводячи грамотно названі проміжні змінні.

· Якщо вираз досить складний, винесіть його в грамотно названу булеву функцію.

· Якщо в кінці блоків else та if є однаковий фрагмент коду, винесіть його за межі оператора if-then-else.

· Використовуйте замість управляючої змінної виходу із циклу break або return.

**Рефакторинг на рівні окремих методів**:

· Іноді фрагмент методу краще представити окремим методом.

· Якщо метод простий та зрозумілий, можна вбудувати його код відразу в програму.

· Якщо метод дуже об’ємний, спробуйте перетворити його у клас і розбити на кілька методів.

· Заміна складного алгоритму на простий.

· Якщо метод потребує додаткової інформації, передавайте її у вигляді додаткового параметра, і навпаки, якщо метод параметра не потребує – видаліть його.

· Розділення операцій запиту даних від операцій по їх зміні.

· Об’єднання схожих методів шляхом їх параметризації.

· Якщо метод у залежності від параметра виконує різний код, краще розділити його на окремі методи, щоб не передавати параметр.

· Якщо Ви передаєте у метод кілька значень одного об’єкта, подумайте, чи не краще передати відразу весь об’єкт. І навпаки, якщо об’єкт створюється лише для передачі його в метод, подумайте про таку зміну методу, щоб він приймав окремі поля.

**Рефакторинг реалізації класів**:

· Якщо ви створюєте і підтримуєте багато копій крупних або складних об’єктів, змініть підхід так, щоб існував лише один оригінал об’єкта, а в решті коду використовувалися об’єкти-посилання. І навпаки, якщо доводиться докладати зусиль для обробки посилань на невеликі чи прості об’єкти, зробіть всі об’єкти посилання об’єктами-значеннями.

· Якщо кілька підкласів містять схожий код, об’єднайте цей код і перемістіть його в суперклас.

**Рефакторинг інтерфейсів класів**:

· Якщо клас майже нічого не робить, перемістіть його код в інші, більш зв’язні класи, а його видаліть.

· Якщо клас має кілька областей відповідальності, розділіть його на кілька класів, що матимуть чітко визначені області відповідальності.

· Для переміщення методу з одного класу в інший можна створити в цільовому класі новий метод і перемістити в нього тіло методу зі старого класу. Після цього можна викликати новий метод зі старого.

· Якщо клас не можна змінювати, проте потрібно включити додатковий метод, Ви можете створити цей метод у клієнтському класі.

· Якщо дані-члени відкриті, краще зробити їх закритими та реалізовувати доступ до них за допомогою методів.

· Якщо значення поля встановлюватиметься під час створення об’єкту і не змінюватиметься надалі, ініціалізуйте поле в конструкторі об’єкта.

· Якщо степінь спеціалізації підкласу невисока, об’єднайте його із суперкласом.

**Невірному застосуванню рефакторингу можуть запобігати**:

· Збереження первісного коду.

· Виконання окремих видів рефакторингу по одному за раз.

· Складання списку дій, що Ви збираєтесь виконувати та списку видів рефакторингу, що будете виконувати пізніше.

· Часте створення контрольних точок.

· Завдання компілятору найбільш строгого рівня діагностики.

· Створення регресивних та інших додаткових тестів.

· Зміна підходу в залежності від ризикованості рефакторингу (деякі види рефакторингу пов’язані з більш високим рівнем ризику, ніж інші).

Рефакторинг – зміна **працездатного** коду, яка не впливає на поведінку програми. Тому не потрібно його застосовувати для приведення програми до працездатного стану. Також іноді код краще переписати з нуля, ніж покращувати невеликими змінами.

Приклади рефакторингу:

1. До: x = y % 32;

Після: x = y & 31;

2. До: x = y \* 8;

Після: x = y << 3;

3. До: x = y / w + z / w;

Після: x = (y + z) / w;

1. До: if( a==b && c==d && e==f ) {...}

Після: if( ((a-b)|(c-d)|(e-f))==0 ) {...}

1. До: if( (x & 1) || (x & 4) ) {...}

Після: if( x & 5 ) {...}

1. До: if( x >= 0 && x < 8 && y >= 0 && y < 8 ) {...}

Після: if( ((unsigned)(x|y))<8 ) {...}

1. До: if( (x==1) || (x==2) || (x==4) || (x==8) || ... )

Після: if( x&(x-1)==0 &&x!=0 )

1. До: if( (x==2) || (x==3) || (x==5) || (x==7) || (x==11) || (x==13) || (x==17) || (x==19) ) {...}

Після: if( (1<<x) & ((1<<2)|(1<<3)|(1<<5)|(1<<7)|(1<<11)|(1<<13)|(1<<17)|(1<<19)) ) {...}

1. До: #define abs(x) (((x)>0)?(x):-(x))

Після: static i32 abs( i32 x )

{  
i32 y = x >> 31;

return ( x ^ y ) - y;

}

1. До: int a[3][3][3]; int b[3][3][3];

for(i=0;i<3;i++)

for(j=0;j<3;j++)

for(k=0;k<3;k++)  
b[i][j][k] = a[i][j][k];

Після: struct 3DArrayType

{  
int elem[3][3][3];

};  
3DArrayType a,b;

b = a;

1. До: for(i=0;i<3;i++)

for(j=0;j<3;j++)  
for(k=0;k<3;k++)  
a[i][j][k] = 0;

Після: memset( a, 0, sizeof( a ) );

***Контрольні запитання*.**

1. Що таке рефакторинг?
2. Які підстави проведення рефакторингу?
3. Яким чином можна оптимізувати код?
4. Яким чином засоби рефакторингу впливають на ризики програмного забезпечення?

***Контрольні запитання для письмових відповідей***

Для прикладів №№1-11 надайте пояснення в чому полягав рефакторинг і в чому полягало покращання. Перед поясненням вкажіть номер прикладу та текст прикладу.

Кожна відповідь 0,25 балів.

**Надіслати до наступного заняття.**

***Література.***

1. Лавріщева К.М. Програмна інженерія. Електронний підручник. URL: <http://csc.knu.ua/uk/library/books/lavrishcheva-6.pdf>.
2. МакКоннелл С. Совершенный код. Практическое руководство по разработке программного обеспечения / С.МакКоннелл. –Спб. : Питер, 2007. –896 с.
3. Соммервил И. Инженерия программного обеспечения / И.Соммервил.–М. : Издательский дом «Вильямс», 2002. –623 с.

***Для самостійного вивчення***: Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.