**Лабораторна робота №10. Застосування методів рефакторінгу при розробці програмного забезпечення**

**Мета роботи**. Навчитися застосовувати на практиці прийоми рефакторингу та оптимізації коду.

**Завдання на лабораторну роботу**

1. Ознайомитись з теоретичними відомостями, необхідними для виконання роботи.

2. Використовуючи власні тексти програм виконати їх рефакторинг та оптимізацію коду.

3. Оформити звіт.

4. Відповісти на контрольні запитання.

**Зміст звіту**

1. Прізвище, назва групи, назва роботи.

2. Текст розробленого програмного забезпечення до проведення рефакторингу з коментарями щодо фрагментів програмного коду, які підлягають рефакторингу, з визначенням підстав для проведення рефакторингу та напрямок проведення рефакторингу.

3. Текст розробленого програмного забезпечення після проведення рефакторингу .

4. Висновки щодо цілей, які досягаються через проведення рефакторингу .

**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?
5. **Строк виконання цієї роботи ІПЗ-41 18.01.2021**

ТЕОРЕТИЧНІ ВІДОМОСТІ

**Рефакторинг** – підхід до розроблення програмного забезпечення з удосконаленням коду шляхом послідовності семантично коректних перетворень, зміна внутрішньої структури програмного забезпечення для полегшення розуміння коду і здешевлення модифікації.

**Підстави для проведення рефакторингу**:

– код дублюється;

– метод занадто довгий;

– цикл занадто довгий або рівень вкладеності тіла циклу занадто великий;

– клас має погану зв’язність;

– інтерфейс класу не формує узгоджену абстракцію;

– метод має занадто багато параметрів;

– окремі частини класу змінюються незалежно від інших частин;

– під час зміни програми потрібно паралельно змінювати декілька класів;

– доводиться паралельно змінювати декілька ієрархій наслідування;

– споріднені дані, які використовуються разом, не організовані в клас;

– назва класу чи методу має ім'я, яке недостатньо точно відповідає змісту;

– занадто поширене використання глобальних змінних;

– складний код пояснюється за допомогою коментарів;

– підклас використовує тільки деяку частину методів батьківського класу;

– об’єкт-посередник нічого не виконує;

– дані–члени класу відкриті.

**Напрямки проведення рефакторингу** можуть включати наступні:

– рефакторинг даних;

– рефакторинг на рівні окремих операторів;

– рефакторинг на рівні окремих методів;

– рефакторинг реалізації класів;

– рефакторинг інтерфейсів класів;

– рефакторинг на рівні системи.



З рефакторингом пов’язаний певний ризик, який виникає внаслідок внесення змін в діючий код, що може привести до появи трудно знаходжуваних помилок в програмі. Неправильно здійснюючи рефакторинг, можна втратити дні та навіть тижні. Ще більшим ризиком буде рефакторинг, здійснюваний без формальностей або епізодично.

**Існує декілька методів рефакторингу.** Кожен метод описує мотивацію та техніку випробуваного на практиці перетворення коду. Деякі види рефакторингу, такі як «Виділення методу» або «Переміщення поля», можуть здаватися очевидними, але це може бути оманою. Для організованого здійснення рефакторингу важливим є розуміння техніки методів рефакторингу. За допомогою методів рефакторингу можна поетапно модифікувати код, вносячи кожний раз невеликі зміни, що знижує ризик, пов’язаний з розвитком проекту.

**Рефакторинг** - це процес такої зміни програмної системи, при якому не змінюється зовнішня поведінка коду, але покращується його внутрішня структура. Це спосіб систематичного приведення коду в порядок, при якому шанси появи нових помилок мінімальні. По суті при проведенні рефакторинга коді ви покращуєте його дизайн вже після того, як він написаний.

В сьогоднішньому розумінні розробки програмного забезпечення ми спочатку створюємо дизайн системи, а потім пишемо код. Спочатку створюється хороший дизайн, а потім виконується кодування. З часом код модифікується, і цілісність системи, відповідність її структури початково створеному дизайну поступово погіршується.

**Рефакторинг є протилежною практикою.** З її допомогою можна взяти поганий проект, навіть хаотичний, та перетворити його в добре спроектований код. Кожен крок цього процесу є простим. Переміщується поле з одного класу в інший, вилучається частина коду з методу і поміщується в окремий метод, якийсь код переміщується в ієрархії в тому або іншому напряму. Підсумковий ефект таких невеликих змін може радикально покращити проект. Це прямо протилежне звичайному явищу поступового розпаду програми.

**При проведенні рефакторингу співвідношення різних етапів робіт змінюється.** Проектування безперервно здійснюється весь час при розроблюванні, а не виконується цілком завчасно. При реалізації системи стає ясно, як можна покращувати її проект. Внаслідок чого створюється програма, якість проекту якої залишається високим протягом продовження розробки.

**Правила рефакторингу**

* Якщо в програму потрібно додати нову функціональність, але код програми не структурований зручним для додаванням цієї функціональності, спочатку потрібно провести рефакторинг програми, щоб спростити внесення необхідних змін, а тільки потім додавайте функцію.
* Перед початком рефакторингу потрібно впевнитися, що мається надійний комплект тестів. Ці тести повинні бути самоперевіряющимися.
* При застосування рефакторингу програма модифікується невеликими кроками. помилку неважко виявити.
* Написати код, зрозумілий комп’ютеру, може кожен, але тільки хороші програмісти пишуть код, зрозумілий людям.

Ритм рефакторингу: тестування, малі зміни, тестування, малі зміни, тестування, малі зміни ... Саме такий ритм робить рефакторинг швидким і надійним.

**Принципи рефакторингу**

**Рефакторинг (Refactoring):** зміни у внутрішній структурі ПЗ, що має на меті полегшення його роботи та спрощує модифікацію, не зачиплюючі спостерігаємої поведінки.

**Виконувати рефакторинг (Refactor):** змінювати структуру ПЗ, застосовуючи ряд рефакторингів, не зачиплюючі його поведінки.

Рефакторинг не змінює видимої поведінки ПЗ. Воно продовжує виконувати існуючи функції. Ніхто — ні кінцевий користувач, ні програміст — не може сказати по зовнішньому виду, що щось змінилося.

**Потреба у рефакторингу**

* Рефакторинг покращує композицію ПЗ
* Рефакторинг полегшує розуміння ПЗ
* Рефакторинг допомагає знайти помилки
* Рефакторинг дозволяє швидше писати програми

**Когда следует проводить рефакторинг?**

Рефакторингом потрібно займатися постійно потрошку.

* **Правило трех ударов** Дона Роберте (Don Roberts). Роблячи щось вперше, ви це просто робите. Роблячи щось аналогічне вдруге, ви зморщуєтесь від необхідності повторення, але все-таки повторюєте те ж саме. Роблячи щось схоже в третій раз, ви починаєте рефакторинг.
* Застосовуйте рефакторинг при додаванні нової функції
* Застосовуйте рефакторинг, якщо потрібно виправити помилку.
* Застосовуйте рефакторинг при розборі коду

**Причини, за яких важко працювати програмами**

1. Програми, важкі для читання, їх важко модифікувати.
2. Програми, в логіці яких є дублювання, важко модифікувати.
3. Програми, яким потрібні додаткові функції, що вимагає змін в працюючому коді, важко модифікувати.
4. Програми, які реалізують складну логіку умовних операторів, важко модифікувати.

Таким чином, нам потрібні програми, які легко читати, вся логіка яких задана в одному і тільки одному місці, модифікація яких не ставить під загрозу існуючі функції і які дозволяють виражати умовну логіку можливо у простіший спосіб.

Рефакторинг є процесом покращання працюючої програми не шляхом зміни її функцій, а шляхом підсилення в ній вказаних якостей, що дозволяють продовжити розробку високою швидкістю.

**Коли рефакторинг не потрібен**

**В деяких випадках рефакторинг взагалі непотрібен,** наприклад, коли потрібно переписати програму з нуля. Часто це випадок, коли наявний код настільки заплутаний, що замість його ре факторингу простіше все розпочати з самого початку.

Основна ознака необхідності переписати код — його непрацездатність. Це виявляється тільки при його тестуванні, коли з’ясовується, що помилок так багато, що зробити код стійким не вдається. Головне правило - перед початком рефакторингу код повинен виконуватися в основному коректно.

Потрібно утримуватися від рефакторингу, коли близька дата завершення проекту. зростання продуктивності, досягеуте завдяки рефакторингу, проявить себе занадто пізно — після завершення строку. Уорд Каннингем (Ward Cunningham) порівнює незавершений рефакторинг з влізанням в борги. Більшості компаній для нормальної роботи потрібні кредити. Але разом з боргами з’являються відсотки, тобто додаткова вартість обслуговування і розширення, обумовлена занадто високою складністю коду. Виплату будь-яких відсотків можна перенести, якщо платежі не настільки великі, що приведуть до розорення. Важно керувати своїми боргами, сплачуючи їх частину через рефакторинг.

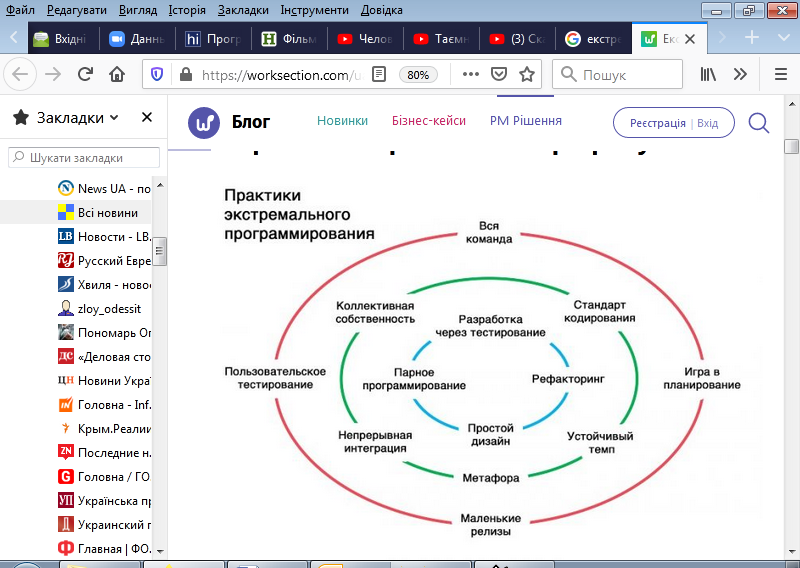
Але наближення строку завершення робот — єдиний випадок, коли можна відкласти рефакторинг, посилаючись на нестачу часу. Оскільки проведення рефакторингу приводить до росту продуктивності праці. Нехватка часу зазвичай сигналізує про необхідність рефакторингу.

**Рефакторинг та проектування**

Рефакторинг є певним доповненням до проектування. Продумана архітектура програми убезпечує від подальшої високовартісної переробки. Програма цілком пов’язана з обміркуванням. Елістер Кокберн (Alistair Cockburn): «При наявності готового дизайну я думаю дуже швидко, але в моєму мисленні повно пробілів».

Рефакторинг не може бути альтернативою попередньому проектуванню. Може фактично діяти такий підхід: перше рішення, що прийшло на думку, реалізується в коді, доводиться до робочого стану, а потім йому за допомогою рефакторингу надається потрібна форма. Пропагандистами такого підходу є прихильники «екстремального програмування». Такий підхід не є самим ефективним. Навіть «екстремальні» програмісти спочатку розробляють деяку архітектуру майбутньої системи. Вони апробовують різні ідеї за допомогою карток або подібних підходів, доки не отримають первісного рішення, яке заслуговує на довіру, після чого приступають до кодування, а потім до рефакторингу. Таким чином, при використанні рефакторингу змінюється роль попереднього проектування. Але при такому підході велика ймовірність, що можливі зміни проекту в майбутньому можуть бути занадто високовартісними. Саме для унеможливлення наких ситуацій вкладається багато зусиль в попереднє проектування. При застосуванні рефакторингу шукається не єдине правильне рішення (як при попередньому проектуванні), а прийнятне рішення. Це рішення в процесі реалізації при заглибленні в предметну область і поглибленому розумінні задачі може бути покращене за допомогою рефакторингу, тому модифікація не буде вартісною.

Рефакторинг надає інший підхід до ризиків модифікації. Потрібно передбачати можливі зміни, розглядати гнучкі рішення, але якщо за допомогою ре факторингу звичайне рішення буде нескладно перетворити на гнучке, то доцільно реалізовувати звичайне рішення. Рефакторинг дозволяє створювати більш прості проекти, не жертвуючи гнучкістю, що робить процес проектування легшим та менш напруженим. Головне - навчитися в цілому розпізнавати те, що легко піддається рефакторингу, після чого о гнучкості рішень навіть перестаєш замислюватися, оскільки є впевненість в можливості застосування в разі потреби рефакторингу. Створюються простіші рішення, які можуть працювати, а гнучкі і складні рішення можуть бути не затребуваними.



**Рефакторинг та продуктисність**

З рефакторингом зазвичай пов’язане питання о його впливі на продуктивність програми. Для полегшення розуміння роботи програми часто здійснюється модифікація, що приводить до зменшення швидкості виконання програми. Але при цьому рефакторинг робить її більш зручною і для внесення змін, і для настроювання продуктивності. Зазвичай швидкі програми (якщо тільки вони не призначені для роботи в жорсткому режимі реального часу) спочатку пишуться як такі, що можна настроювати, а потім настроюються до потрібної швидкості. Але модифікації, що підвищують продуктивність зазвичай утруднюють роботу з програмою, що впливає на швидкість розробки програми.

При аналізі більшості програм з’ясовується, що велика частина часу витрачається невеликою частиною коду. Саме для ефективної оптимізації потрібен зрозумілий код, щоб легко можна було локалізувати потрібні місця в програмі. Це передбачає створення програми, розкладену на компоненти без погляду на продуктивність. Налагодження продуктивності зазвичай виконується на пізніх етапах розробки. Завдяки компонентній структурі вузькі місця, де неефективно використовується час і пам’ять легко локалізуються. І саме на цих місцях зосереджується увага. Як і при проведенні рефакторингу, зміни потрібно вносити невеликими порціями, кожен раз компілюючи, тестуючи і контролюючі отримані показники продуктивності. Якщо продуктивність не зросла зміни відміняються. Процес пошуку і ліквідації вузьких місць продовжується до досягнення продуктивності, яка задовольняє користувачів.

**Розробка тестів**

При проведенні рефакторингу важливою попередньою умовою є наявність надійних тестів.

**Правила розробки тестів**

* Робіть все тести повністю автоматичними, щоб вони перевіряли власні результати.
* Комплект тестів є потужним детектором помилок, який різко скорочує час їх пошуку.
* Частіше запускайте тести. Запускайте тести при кожній компіляції — кожний тест хоч раз на день.
* Отримавши повідомлення про помилку, починайте зі створення тесту модуля, який надав цю помилку.
* Краще написати і виконати неповні тести, ніж не виконати повні тести.
* Пам’ятайте про граничні умови, які можуть бути неправильно оброблені, та зосередьте на них свої тести.
* Не забувайте перевіряти, щоб у випадку виникнення в проблем генерувались виключні ситуації.
* Те, що тестування не виявить всі помилки, не повинно заважати написанню тестів, які виявлять більшість помилок.

**Проблеми рефакторингу**

* Потреба вносити зміни в існуючий код
* Необхідність строго притримуватися поставленої задачі
* Покривати код перевірочними тестами

**Ознаки потреби в рефакторингу**

* Програмний продукт працює, але внесення нової функціональності інколи затягується на тижні;
* В окремих місцях код працює зовсім не так, як очікується;
* Часті помилки в строках реалізації поставленої задачі;
* Потреба вносити однотипні зміни в різних місцяхх.

**Методи рефакторинга**

* Інкапсуляція поля (Encapsulate Field);
* Виділення класу (Extract Class);
* Виділення інтерфейсу (Extract Interface);
* Виділення локальної змінної (Extract Local Variable);
* Виділення методу (Extract Method);
* Генералізація типу (Generalize Type);
* Вбудовування (Inline);
* Введення фабрики (Introduce Factory);
* Введення параметра (Introduce Parameter);
* Піднімання поля/методу (Pull Up);
* Спуск поля/методу (Push Down);
* Заміна умовного оператору поліморфізмом (Replace Conditional with Polymorphism);
* тощо