МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Я.В. Хіцко, Л.А. Люшенко, Ю.В. Бухтіяров

ТЕХНОЛОГІЯ ПРОЕКТУВАННЯ ПРОГРАМНИХ СИСТЕМ

ЛАБОРАТОРНИЙ ПРАКТИКУМ

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського як навчальний посібник для студентів, які навчаються за спеціальністю 121 «Інженерія програмного забезпечення» (освітня програма «Програмне забезпечення комп’ютерних та інформаційно-пошукових систем»)

Київ

КПІ ім. Ігоря Сікорського

2019

Рецензенти: Боярінова Юлія Євгенівна, кад. техн. наук, доц.
Голуб Белла Львівна, канд. техн. наук, доц.

Відповідальний редактор Легеза Віктор Петрович, д-р. техн. наук, проф.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 6 від 31.01.2020 р.)
за поданням Вченої ради факультету прикладної математики (протокол № 8 від 27.01.2020 р.)

Електронне мережне навчальне видання

Хіцко Яна Володимирівна, канд. техн. наук,
Люшенко Леся Анатоліївна, канд. техн. наук,
Бухтіяров Юрій Вікторович

ТЕХНОЛОГІЯ ПРОЕКТУВАННЯ ПРОГРАМНИХ СИСТЕМ

ЛАБОРАТОРНИЙ ПРАКТИКУМ

Технологія проектування програмних систем: лабораторний практикум [Електронний ресурс] : навч. посіб. для студ. спеціальності 121 «Інженерія програмного забезпечення» (освітня програма «Програмне забезпечення комп’ютерних та інформаційно-пошукових систем») / КПІ ім. Ігоря Сікорського ; уклад.: Я.В. Хіцко. – Електронні текстові дані (1 файл: 1,26 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2019. – 44 с.

Навчальний посібник розроблено для ознайомлення студентів з методологіями проектування, планування та забезпечення якості програмних систем, а також рефакторингу. Навчальне видання призначене для студентів, які навчаються за спеціальністю 121 Інженерія програмного забезпечення (освітня програма «Програмне забезпечення комп’ютерних та інформаційно-пошукових систем») факультету прикладної математики КПІ ім. Ігоря Сікорського.

© Я.В. Хіцко, 2020
© Л.А. Люшенко, 2020
© Ю.В. Бухтіяров, 2020
© КПІ ім. Ігоря Сікорського, 2020

ЗМІСТ

ВСТУП 5
ЛАБОРАТОРНА РОБОТА № 1. ІНСПЕКЦІЇ ПРОГРАМНОГО КОДУ 6
Завдання для лабораторної роботи 6
Теоретичні відомості 6
Вказівки до виконання лабораторної роботи 9
Вимоги до оформлення звіту 14
Питання для самоперевірки 14
Рекомендована література 15
ЛАБОРАТОРНА РОБОТА № 2. МОДУЛЬНІ ТЕСТИ 15
Завдання для лабораторної роботи 15
Теоретичні відомості 15
Вказівки до виконання лабораторної роботи 23
Вимоги до оформлення звіту 26
Питання для самоперевірки 26
Рекомендована література 27
ЛАБОРАТОРНА РОБОТА № 3. КІЛЬКІСНІ МЕТРИКИ КОДУ 27
Завдання для лабораторної роботи 27
Теоретичні відомості 27
Вказівки до виконання лабораторної роботи 29
Вимоги до оформлення звіту 32
Питання для самоперевірки 33
Рекомендована література 33
ЛАБОРАТОРНА РОБОТА № 4. ОБ’ЄКТНО_ОРІЄНТОВАНІ МЕТРИКИ КОДУ 33
Завдання для лабораторної роботи 34
Вказівки до виконання лабораторної роботи 42
Вимоги до оформлення звіту 43

Питання для самоперевірки 43
Рекомендована література 43

ВСТУП

Навчальна дисципліна "Технологія проектування програмних систем" є нормативною і входить до складу циклу професійно-орієнтованих дисциплін навчального плану підготовки магістрів, що навчаються за спеціальністю 121 «Інженерія програмного забезпечення» освітньої програми «Програмне забезпечення комп'ютерних та інформаційно-пошукових систем».

Предмет дисципліни – теоретичні і практичні основи організації процесів проектування та розробки програмних систем, процесів вимірювання та забезпечення якості програмного забезпечення та рефакторингу існуючого вихідного коду.

Метою навчальної дисципліни є формування у студентів здатностей аналізувати вимоги до програмних системи та умов їх проектування, обирати методологію розробки програмних систем відповідно до визначених вимог та середовища проектування та конструювання програмного забезпечення, визначати та аналізувати метрики якості програмного забезпечення, забезпечувати якісну інспекцію артефактів розробки програмного забезпечення,
забезпечувати модульне та інтеграційне тестування програмного забезпечення,
визначати та аналізувати метрики якості програмного забезпечення, а також забезпечувати якісний рефакторинг існуючого програмного коду.

У кожному розділі надаються теоретичні відомості з певної теми, завдання на лабораторну роботу цієї теми, вказівки до виконання завдання, а також наводяться вимоги до оформлення звіту з виконаної лабораторної роботи, контрольні питання для самоперевірки та список використаної та рекомендованої літератури.

ЛАБОРАТОРНА РОБОТА № 1. ІНСПЕКЦІЇ ПРОГРАМНОГО КОДУ

Мета роботи: навчитися робити огляд коду та фіксувати зауваження, навчитися надсилати код на огляд та виправляти зауваження до свого програмного коду.

Завдання для лабораторної роботи

У ході роботи потрібно:

1. Закодувати завдання і надіслати посилання на виконане завдання іншому студенту, після інспекції коду виправити зауваження.
2. Виконати інспекцію коду іншого студента, зафіксувати зауваження та переглянути код після виправлення зауважень.

Теоретичні відомості

Огляд коду (code review) - систематичний перегляд вихідного коду ПЗ. Він призначений для виявлення помилок, виявлених на початковому етапі розробки, і, як результат, підвищення загальної якості програмного забезпечення.

Огляди коду роблять в різних формах, таких як парне програмування, неформальні наскрізні перевірки і формальні інспекції.

В оглядах коду часто можна знайти і усунути поширені вразливості, такі як вразливості форматування рядків, умовні розгалуження, витоку пам'яті і переповнення буфера, тим самим підвищуючи безпеку програмного забезпечення.

Online репозиторії програмного забезпечення на основі Subversion (з Redmine або Trac), Mercurial, Git або інші дозволяють групам людей спільно переглядати код. Крім того, спеціальні інструменти для спільного перегляду коду можуть полегшити процес перевірки коду.

Швидкість огляду коду повинна становити від 200 до 400 рядків коду на годину. Перевірка і аналіз більше декількох сотень рядків коду на годину для

критично важливого програмного забезпечення (наприклад, критично важливого програмно-апаратних засобів) може виявитися занадто швидкими для того, щоб знайти помилки.

Дефекти, виявлені в оглядах коду, типізуються і досліджуються. Емпіричні дослідження свідчать про те, що до 75% дефектів оглядів коду впливають на еволюцію програмного забезпечення, а не на функціональність, що робить огляди коду відмінним інструментом для компаній-розробників програмного забезпечення з великим життєвим циклом продукту або системи.

Для інспекторів початківців рекомендується складати контрольний список того, на що необхідно звернути увагу під час переглядів коду / архітектури.

Контрольний список огляду коду

1. Загальні питання

- Чи є посилання на неініціалізовану змінну?
- Чи є цілими індекси масиву та рядка і чи завжди вони знаходяться в межах розміру масиву чи рядка?
- Чи є якісь потенційні помилки "від однієї" помилки в операціях індексації або посилання підпису на масиви?
- Чи застосовується змінна, коли константа фактично працює краще, наприклад, під час перевірки межі масиву?
- Чи змінній колись присвоюється значення типу, яке відрізняється від змінної? Наприклад, чи присвоюється число з плаваючою комою цілій змінній?
- Чи виділяється пам'ять для вказівників?
- Якщо на структуру даних посилаються декілька функцій або підпрограм, чи визначена структура однаково в кожній з них?

2. Помилки декларації даних

- Якщо змінна ініціалізується одночасно декларацією, чи правильно вона ініціалізована та відповідає її типу?

- Чи є змінні з подібними іменами? Це не обов'язково помилка, але це може бути ознакою того, що імена плутають із тими, що є десь у програмі.
- Чи оголошуються будь-які змінні, на які ніколи не посилаються або посилаються лише один раз?
- Чи всі змінні явно задекларовані в їх конкретному модулі? Якщо ні, чи розуміється, що змінна поділяється з наступним вищим модулем?

3. Помилки обчислення

- Чи є у коді будь-які обчислення, що використовують змінні різних типів даних, наприклад додавання цілого числа до числа з плаваючою комою?
- Чи можливий переповнення в процесі числового обчислення?
- Чи є можливим випадок, що дільник дорівнювати нулю?
- У випадках цілочисельної арифметики, чи обробляє код ситуації (наприклад, операція ділення), що призведять до втрати точності?
- Чи може значення змінної вийти за межі її значущого діапазону? Наприклад, чи може результат імовірності бути меншим за 0% або більше 100%?
- Чи є плутанина щодо виразів, що містять кілька операторів, щодо порядку оцінювання і чи правильний пріоритет оператора? Чи потрібні дужки для уточнення?
- Чи правильні порівняння?
- Чи є порівняння між значеннями дробу або плаваючої точки? Якщо так, чи вплинуть якісь проблеми з точністю на їх порівняння? Чи 1.00000001 досить близький до 1.00000002, щоб бути рівним?
- Чи операнди булевого оператора булеві? Наприклад, чи використовується ціла змінна для обчислення булевої величини?

4. Помилки управління потоком програми

- Чи припиняться програма, модуль, підпрограма чи цикл? Якщо цього не буде, це прийнятно?

- Чи існує можливість передчасного виходу з циклу?
- Чи можливо, що цикл ніколи не виконується? Чи прийнятно це, якщо цього немає?
- Якщо програма містить багатосторонню гілку, таку як switch ... case, чи може змінна індексу колись перевищувати кількість можливостей гілки? Якщо це так, чи правильно обробляється ця справа?

5. ***Помилки параметрів підпрограми***

- Чи відповідають типи та розміри параметрів, отриманих підпрограмою, тим, що надсилаються кодом, що її викликає? Чи правильний порядок?
- Якщо константи коли-небудь передаються як аргументи, чи не змінюються вони випадково у підпрограмі?
- Чи змінює підпрограма параметр, призначений лише як вхідне значення?
- Якщо існують глобальні змінні, чи мають вони подібні визначення та атрибути у всіх посилаються на підпрограми?

6. ***Помилки вводу/виводу***

- Чи програмне забезпечення суворо дотримується заданого формату даних, які читає чи записує зовнішній пристрій?
- Якщо файл або база дани відсутні, чи обробляється ця помилка?
- Чи програмне забезпечення вирішує ситуацію з відключенням зовнішнього пристрою під час читання чи запису?
- Чи перевірені всі повідомлення про помилки на правильність, відповідність, граматику та написання?

Вказівки до виконання лабораторної роботи

1. Необхідно сформувати команди по 2 студенти.

2. Обидва студента роблять завдання Euro Diffusion на будь-якій мові програмування та завантажують готові матеріали на github.com, повідомляють про це викладача та іншого студента з команди по імейл.

Euro Diffusion

Повинна бути написана програму для імітації розповсюдження євро монет по всій Європі, використовуючи дуже спрощену модель. Європейські міста представлені точками у прямокутній сітці. У кожному місті може бути до 4 сусідів (по одному на північ, схід, південь та захід). Кожне місто належить країні, а країна - прямокутна частина площини. На рис. 1 нижче показана карта з 3 країнами та 28 містами. Графи країн пов'язані, але країни можуть мати прикордонні пустоти, які представляють моря. Спочатку в кожному місті є мільйон (1000000) монет мотиву своєї країни. Щодня репрезентативна частина монет, що вираховується з балансу міста на початку дня, перевозиться до кожного сусіднього міста. Репрезентативна частина визначається як одна монета на кожні 1000 монет одного мотиву.

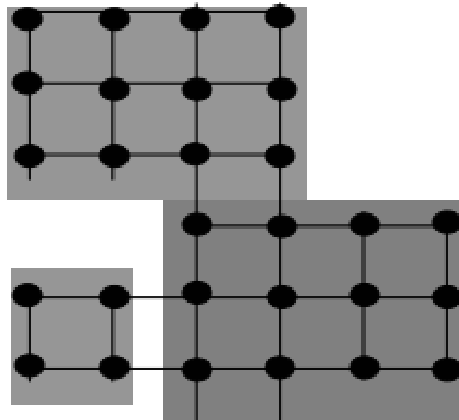

Рис. 1. Карта країн

Вхідні дані

Вхідні дані складаються з декількох тестових сценаріїв. Перший рядок кожного тестового сценарію - це кількість країн ($1 \leq c \leq 20$). Наступні c рядки описують кожну країну. Опис країни має формат: <назва xl yl xh yh>, де «назва» - це одне слово, що містить не більше 25 символів; xl, yl - нижні ліві координати

міста цієї країни і xh, yh - верхні праві координати міста цієї країни. $1 \leq xl \leq xh \leq 10$ та $1 \leq yl \leq yh \leq 10$. Останній випадок на вході супроводжується одиничним нулем.

Вихідні дані

Для кожного тестового випадку надрукуйте рядок із зазначенням номера сценарію, а потім рядок для кожної країни з назвою країни та кількістю днів для завершення обігу монет в цій країні. Розсортувати по кількості днів для завершення обігу. Якщо дві країни мають однакову кількість днів, їх потрібно сортувати за назвою в алфавітному порядку. Використовувати вихідний формат, наведений у Таблиці 1.

Таблиця 1. Приклад вхідних і вихідних даних для задачs Euro Diffusion.

Приклад вхідних даних	**Вихідні дані**
3 France 1 4 4 6 Spain 3 1 6 3 Portugal 1 1 2 2 1 Luxembourg 1 1 1 1 2 Netherlands 1 3 2 4 Belgium 1 1 2 2 0	Case Number 1 Spain 382 Portugal 416 France 1325 Case Number 2 Luxembourg 0 Case Number 3 Belgium 2 Netherlands 2

3. Код виконаної роботи завантажити на github.com та повідомити про це іншому студенту з команди та викладачу, а також надати посилання на репозиторій на github для інспекції.

4. Інспектор підписується на проект автора (рис. 2).

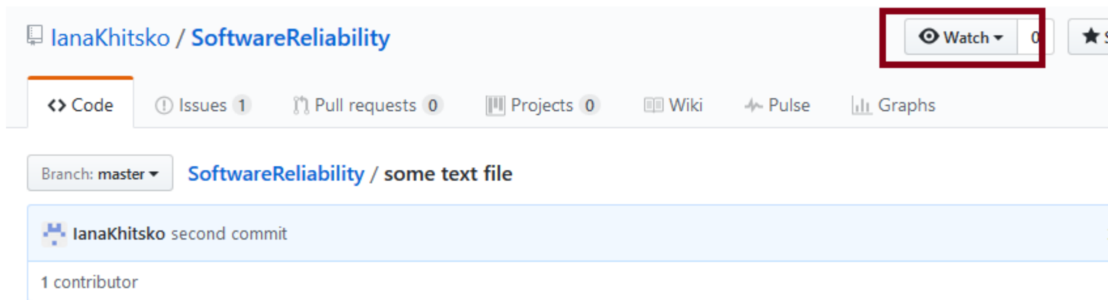

Рис. 2. Інспектор підписується на проект автора.

5. Інспектор додає зауваження по коду (рис. 3-4).

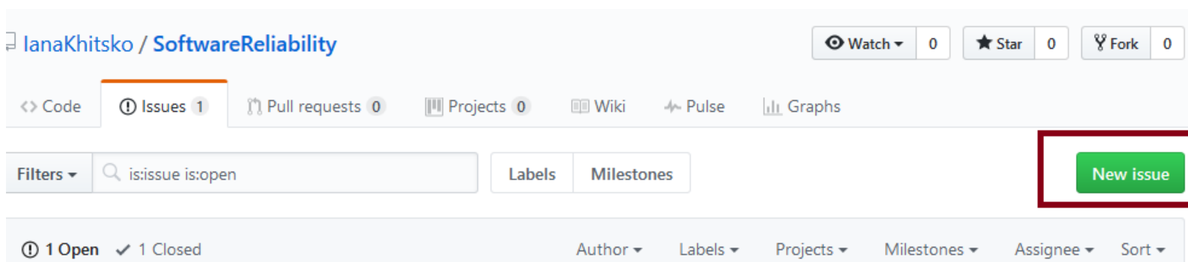

Рис. 3. Додавання зауваження

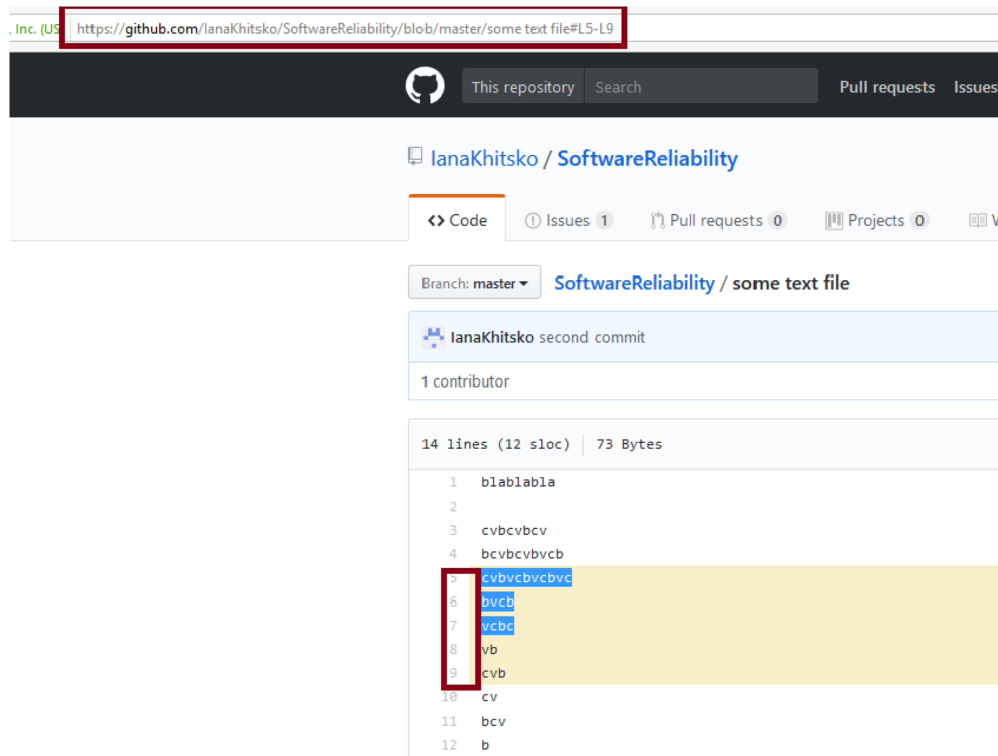

Рис. 4. Копіювання посилання на рядки файлу

Зауваження можна додавати до кожного файлу, бажано попередньо копіювати URL конкретної рядки (рядків), докладніше тут - http://stackoverflow.com/questions/23821235/how-to-link-to-specific-line-number-on-github .

6. Автор переглядає зауваження та сортує їх (рис. 5). Автор усуває функціональні та нефункціональні дефекти, пише коментарі на питання і закриває кожне зауваження. Если автор не согласен или не понял вопрос, он пишет ответный комментарий. Якщо автор не згоден із зауваженням, він пише відповідний коментарій (рис. 6).

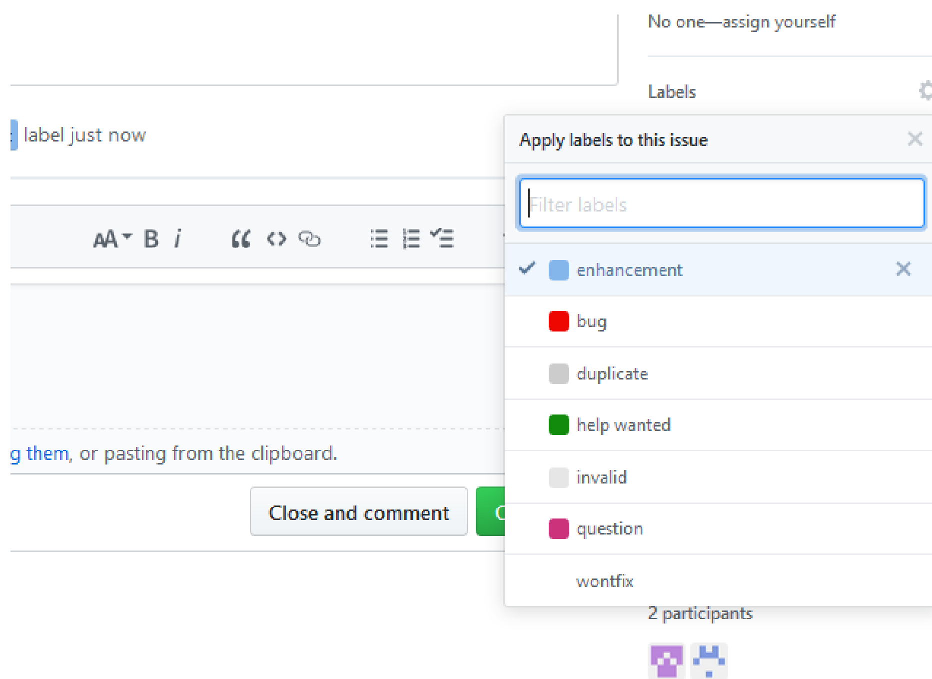

Рис. 5. Сортування зауважень

Рис. 6. Дискусія за зауваженням.

7. Після того, як всі зауваження опрацьовані, автор поновлює файли репозиторію і повідомляє про це інспекторів. Якщо нових зауважень у інспекторів немає, інспекція вважається завершеною.

Вимоги до оформлення звіту

Звіт має включати:

1. Титульний аркуш.
2. Завдання на лабораторну роботу.
3. Хід роботи. Цей розділ складається з послідовного опису виконуваних кроків згідно інструкцій до лабораторної роботи.
4. Висновки.

Питання для самоперевірки

1. Які є форми огляду коду?
2. Як враховуються метрики огляду коду?

3. Про що свідчить низьке число зауважень на тисячу рядків коду? (меньше 20 KLOC)?
4. Яким чином можуть позначатися критичні зауваження, що виявляють дефекти ПЗ?

Рекомендована література

1. Williams, L. & Kessler, R. Pair Programming Illuminated [Text] / Williams, L. & Kessler, R. - Boston: Addison-Wesley Professional, 2003. Kemerer,, C.F.; Paulk, M.C. (2009-04-17). The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *IEEE Transactions on Software Engineering* **35** (4): 534–550. doi:10.1109/TSE.2009.27.
2. Code Review Metrics. *Open Web Application Security Project*. Open Web Application Security Project. - 2015-10-09, режим доступу - https://www.owasp.org/index.php/Code_Review_Metrics#Inspection_Rate.

ЛАБОРАТОРНА РОБОТА № 2. МОДУЛЬНІ ТЕСТИ

Мета роботи: навчитися будувати на запускати модульні тести для обраного модуля та працювати згідно методології Test Driven Development (TDD).

Завдання для лабораторної роботи

У ході роботи потрібно закодувати завдання і написати модульні тести до виконаного рішення.

Теоретичні відомості

Модульне тестування, або юніт-тестування (unit testing) - процес в програмуванні, що дозволяє перевірити на коректність окремі модулі вихідного коду програми.

Процес полягає в тому, щоб писати тести для кожної нетривіальною функції або методу. Це дозволяє досить швидко перевірити, чи не призвела чергова зміна коду до регресії, тобто до появи помилок у вже відтестованих місцях програми, а також полегшує виявлення і усунення таких помилок.

Мета модульного тестування - ізолювати окремі частини програми і показати, що окремо ці частини працездатні.

Цей тип тестування зазвичай виконується програмістами.

Переваги модульного тестування:

- ***Заохочення змін***

Модульне тестування пізніше дозволяє програмістам проводити рефакторинг і бути впевненими, що модуль як і раніше працює коректно (регресійні тестування). Це заохочує програмістів до змін коду, оскільки досить легко перевірити, що код працює і після змін.

- ***Спрощення інтеграції***

Модульне тестування допомагає усунути сумніви з приводу окремих модулів і може бути використано для підходу до тестування «знизу вгору»: спочатку тестуючи окремі частини програми, а потім програму в цілому.

- ***Документування коду***

Модульні тести можна розглядати як «живий документ» для тестованого класу. Клієнти, які не знають, як використовувати даний клас, можуть використовувати юніт-тести в якості прикладу.

- ***Відділення інтерфейсу від реалізації***

Оскільки деякі класи можуть використовувати інші класи, тестування окремого класу часто поширюється на пов'язані з ним. Наприклад, клас користується базою даних; в ході написання тесту програміст виявляє, що тесту доводиться взаємодіяти з базою. Це помилка, оскільки тест не повинен виходити за кордон класу. В результаті розробник абстрагується від з'єднання з базою даних

і реалізує цей інтерфейс, використовуючи свій власний mock-об'єкт. Це призводить до менш пов'язаного коду, мінімізуючи залежності в системі.

Коли модульне тестування не працює:

- ***Складний код***

Тестування програмного забезпечення - комбінаторна задача. Наприклад, кожне можливе значення булевою змінної потребує двох тестів: один на варіант TRUE, інший - на варіант FALSE. В результаті на кожен рядок вихідного коду потрібно 3-5 рядків тестового коду.

Як і будь-яка технологія тестування, модульне тестування не дозволяє відловити всі помилки програми. Справді, це випливає з практичної неможливості трасування всіх можливих шляхів виконання програми, за винятком найпростіших випадків.

- ***Результат відомий лише приблизно***

Наприклад, у математичному моделюванні. Бізнес-додатки часто працюють з кінцевими і зліченними множинами, наукові - з континуальними. Тому складно підібрати тести для кожної з гілок програми, складно сказати, чи вірний результат, витримується чи точність, і т. д.

- ***Помилки інтеграції та интеграции и продуктивності***

При виконанні юніт-тестів відбувається тестування кожного з модулів окремо. Це означає, що помилки інтеграції, системного рівня, функцій, виконуваних в декількох модулях, не будуть визначені. Крім того, дана технологія марна для проведення тестів на продуктивність.

- ***При загальній низькій культурі програмування***

Для отримання вигоди від модульного тестування потрібно строго слідувати технології тестування протягом усього процесу розробки програмного забезпечення. Потрібно зберігати не тільки записи про всі проведені тести, але і про всі зміни вихідного коду у всіх модулях. З цією метою слід використовувати

систему контролю версій ПЗ. Таким чином, якщо пізніша версія ПЗ не проходить тест, який був успішно пройдений раніше, буде нескладно звірити варіанти вихідного коду і усунути помилку. Також необхідно переконатися в незмінному відстеженні та аналізу невдалих тестів, інакше розробка призведе до лавиноподібного збільшення невдалих тестових результатів.

- ***Проблеми з об'єктами-заглушками (stubs)***

За виключенням самих простих випадків, об'єкт, що тестується, повинен взаємодіяти з іншими об'єктами. Ці «товариши з взаємодії» - об'єкти-заглушки - роблять гранично простими: або вкрай спрощеними (пам'ять замість БД), або розрахованими на конкретний тест і що механічно повторюють сесію обміну. Питання починаються, коли протокол обміну змінюється; треба відшукувати ці заглушки у всіх тестах і переводити під новий протокол.

Техніка модульного тестування

1. Складність написання модульних тестів залежить від самої організації коду. Сильне зачеплення (coupling) або велика зона відповідальності окремих сутностей (класи для об'єктно-орієнтованих мов) можуть ускладнити тестування.
2. Для об'єктів, що здійснюють зв'язок із зовнішнім світом (мережева взаємодія, файлове введення-виведенняк і т.д.), слід створювати заглушки. У термінології виділяють більш «просунуті» заглушки - Mock-об'єкти, які несуть в собі логіку. Також спростити тестування може виділення якомога більшої частини логіки в чисті функції. Вони ніяк не взаємодіють із зовнішнім світом і їх результат залежить тільки від вхідних параметрів.
3. Код тестів прийнято виділяти в окремі каталоги. Бажано, щоб додавання нових тестів в проекті не було складним завданням і була можливість запускати всі тести. Деякі системи контролю версій (git), підтримують хукі, за допомогою яких можна налаштувати запуск всіх тестів перед фіксуванням змін. При

помилку хоча б в одному з тестів, зміни зафіксовано не буде. Також можна застосовувати системи безперервної інтеграції.

Видимість коду

1. Набір тестів повинен мати доступ до коду, що тестується. З іншого боку, принципи інкапсуляції і приховування даних не повинні порушуватися. Тому модульні тести зазвичай пишуться в тому ж модулі або проекті, що і тестований код.
2. З коду тесту може не бути доступу до приватних полів і методів. Тому при модульному тестуванні може знадобитися додаткова робота. У Java і C #, розробник може використовувати відображення (*reflection*), щоб звертатися до полів, поміченими як приватні. Модульні тести можна реалізувати у внутрішніх класах, щоб вони мали доступ до членів зовнішнього класу. В .NET Framework можуть застосовуватися колективні класи (*partial classes*) для доступу з тесту до приватних полів і методів.
3. Важливо, щоб фрагменти коду, призначені виключно для тестування, не залишалися в *production* коді. В C ++ для цього можуть бути використані директиви умовної компіляції. Однак це буде означати, що production код не повністю збігається з протестованиv. Систематичний запуск інтеграційних тестів на збірці, що випускається, допоможе упевнитися, що не залишилося коду, що неявно залежить від різних аспектів модульних тестів.

Коли розробляється код, що використовує бази даних, веб-сервіси або інші зовнішні процеси, має сенс виділити частину, що покривається тестуванням. Це робиться в два етапи:

- Скрізь, де потрібен доступ до зовнішніх ресурсів, повинен бути оголошений інтерфейс, через який цей доступ буде здійснюватися.
- Інтерфейс повинен мати дві реалізації. Перша, власне надає доступ до ресурсу, і друга, що є fake- або mock-об'єктом.

Fake-об'єкти додають повідомлення виду «Об'єкт person збережений» в лог, щоб потім перевірити правильність поведінки.

Mock-об'єкт являє собою конкретну фіктивну реалізацію інтерфейсу, призначену виключно для тестування взаємодії і щодо якого висловлюється твердження.

Mock-об'єкти відрізняються від *fake-* тим, що самі містять твердження (assertions), перевіряючі поведінку тестованого коду. Методи *fake- і mock-об'єктів*, які повертають дані, можна налаштувати так, щоб вони повертали при тестуванні одні і ті ж правдоподібні дані. Вони можуть емулювати помилки так, щоб код обробки помилок міг бути ретельно протестований. Іншими прикладами fake-служб, корисними при розробці модульних тестів, можуть бути: служба кодування, що не кодує дані, генератор випадкових чисел, який завжди видає одиницю.

Розробка через тестування (*TDD*) — техніка розробки програмного забезпечення, яка ґрунтується на повторенні дуже коротких циклів розробки: спочатку пишеться тест, що покриває бажану зміну, потім пишеться код, який дозволить пройти тест, і під кінець проводиться рефакторинг нового коду до відповідних стандартів.

Розробка через тестування вимагає від розробника створення автоматизованих модульних тестів, що визначають вимоги до коду безпосередньо перед написанням самого коду.

Тест містить перевірки умов, які можуть або виконуватися, або ні. Коли вони виконуються, кажуть, що тест пройдено. Проходження тесту підтверджує поведінку, передбачуване програмістом.

Розробники часто користуються бібліотеками для тестування (*testing frameworks*) для створення і автоматизації запуску наборів тестів. На практиці модульні тести покривають критичні і нетривіальні ділянки коду. Це може бути

код, який схильний до частих змін, код, від роботи якого залежить працездатність великої кількості іншого коду, або код з великою кількістю залежностей.

Середовище розробки повинно швидко реагувати на невеликі модифікації коду. Архітектура програми повинна базуватися на використанні безлічі сильно пов'язаних компонентів, які слабо зчеплені один з одним, завдяки чому тестування коду спрощується.

TDD не тільки передбачає перевірку коректності, але і впливає на дизайн програми. Спираючись на тести, розробники можуть швидше представити, яка функціональність необхідна користувачеві. Таким чином, деталі інтерфейсу з'являються задовго до остаточної реалізації рішення.

Зрозуміло, до тестів застосовуються ті ж вимоги стандартів кодування, що і до основного коду.

Схема робочого процесу зображена на рис. 7:

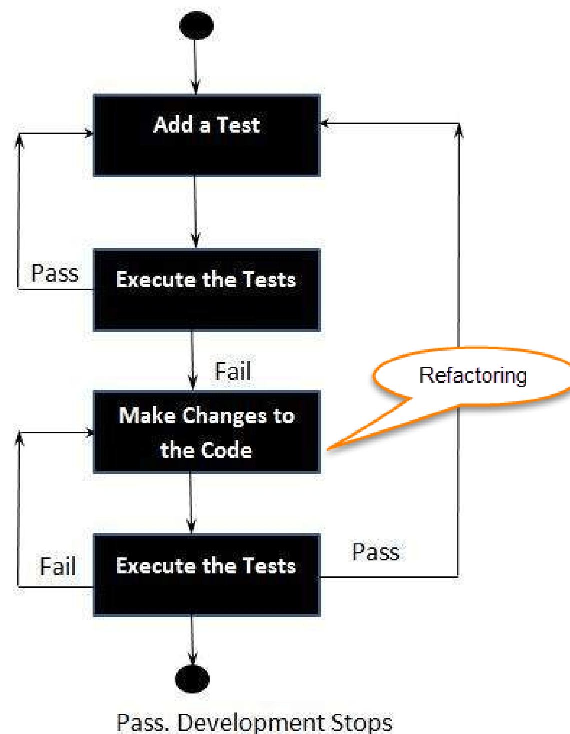

Рис. 7. Цикл розробки через тестування

1. **Додавання нового тесту**

При розробці через тестування, додавання кожної нової функціональності (feature) в програму починається з написання тесту.

Неминуче цей тест не буде проходити, оскільки відповідний код ще не написаний. (Якщо ж написаний тест пройшов, це означає, що або запропонована «нова» функціональність вже існує, або тест має недоліки.)

Щоб написати тест, розробник повинен чітко розуміти які пред'являються до нової можливості вимоги. Для цього розглядаються можливі сценарії використання і призначені для користувача історії. Нові вимоги можуть також спричинити зміну існуючих тестів. Це відрізняє розробку через тестування від технік, коли тести пишуться після того, як код вже написаний: вона змушує розробника ***сфокусуватися на вимогах до написання коду*** - тонка, але важлива відмінність.

2. **Запуск усіх тестів: переконатися, що нові тести не проходять**

На цьому етапі перевіряють, що тільки що написані тести не проходять. Цей етап також перевіряє самі тести: написаний тест може проходити завжди і відповідно бути даремним. Нові тести повинні не проходити із зрозумілих причин. Це збільшить впевненість (хоча не гарантуватиме повністю), що тест справді тестує те, для чого він був розроблений.

3. **Написання коду**

На цьому етапі пишеться новий код так, що тест буде проходити. Цей код не обов'язково повинен бути ідеальний. Припустимо, щоб він проходив тест якимось неелегантним способом. Це прийнятно, оскільки наступні етапи поліпшать і відполірують його.

Важливо писати код, призначений саме для проходження тесту. Не слід додавати зайвої функціональності.

4. **Запуск усіх тестів: переконатися, що всі тести проходять**

Якщо всі тести проходять, програміст може бути впевнений, що код задовольняє всім тестованим вимогам. Після цього можна приступити до завершального етапу циклу.

5. **Рефакторинг**

Рефакторинг - процес зміни внутрішньої структури програми, що не зачіпає її зовнішньої поведінки і має на меті полегшити розуміння її роботи, усунути дублювання коду, полегшити внесення змін в найближчому майбутньому.

6. **Повторити цикл**

Описаний цикл повторюється, реалізуючи все нову і нову функціональність. Кроки слід робити невеликими, від 1 до 10 змін між запусками тестів.
Якщо новий код не задовольняє новим тестам або старі тести перестають проходити, програміст повинен повернутися до налагодження.

При використанні сторонніх бібліотек не слід робити настільки невеликі зміни, які буквально тестують саму сторонню бібліотеку, а не код, що її використовує, якщо тільки немає підозр, що бібліотека містить помилки.

Вказівки до виконання лабораторної роботи

1. Потрібно створити середовище, готове для додавання тестів і реалізації задачі, за допомогою інструментів та бібліотек для модульного тестування:

Для C++ :

CxxTest, CPPUnit, Boost Test, Google C++ Testing Framework, API Sanity Autotest, Qt Test framework

Для C# :

Nunit, XUnit.net, MbUnit.

Для Java і Groovy :

Junit, JUnit.org, TestNG, testNG.org, JavaTESK, UniTESK.ru, Spock.

Для JavaScript:

Mocha (тестовий фреймворк), Chai, Sinon.JS (бібліотека для створення mock-об'єктів, заглушок, використовується з тестовим framework'ом), Karma runner, QUnit, Jasmine.

Для Python :

PyUnit, PyTest, Nose.

2. За допомогою TDD реалізувати задачу Coffee Central:

Постійно зростаюча кількість кав'ярень, які відкриваються у вашому рідному місті, безумовно, стала досить привабливою. Мабуть, люди стали настільки захопленими кавою, що квартири, які знаходяться поруч із багатьма кав'ярнями, фактично отримають більш високу оренду.

На це звернула увагу місцева компанія з нерухомості. Вони зацікавлені у визначенні найцінніших локацій міста з точки зору їхньої близькості до великої кількості кав'ярень. Вони дали вам карту міста, на якій розміщені місця розташування кав'ярень. Припускаючи, що пересічна людина готова пройтися лише фіксованою кількістю кварталів для купівлі їх ранкової кави, ви повинні знайти місце, з якого можна дістатися до найбільшої кількості кав'ярень. Як ви, напевно, знаєте, ваше рідне місто побудовано на квадратній сітці, з кварталами, вирівняними на осях північ-південь та схід-захід. Оскільки вам доводиться йти вулицями, відстань між перехрестями (a; b) та (c; d) становить | a - c | + | b - d |.

Вхідні дані

Вхідні дані містять кілька тестових сценаріїв. Кожен тестовий сценарій описує місто. Перший рядок кожного тестового сценарію містить чотири цілі числа dx, dy, n та q. Це розміри міської сітки dx : dy ($1 \leq dx; dy \leq 1000$), кількість кав'ярень n ($0 \leq n \leq 5105$) та кількість запитів q ($1 \leq q \leq 20$). Кожен з наступних n рядків містить два цілих числа xi та yi ($1 \leq xi \leq dx$, $1 \leq yi \leq dy$); вони вказують місце розташування i-ї кав'ярні. На перехресті буде щонайбільше одна кав'ярня. Кожен з наступних

рядків q містить одне ціле число m (0 ≤ m ≤ 106), максимальну відстань, яку людина бажає пройти за чашкою кави.

Останній тестовий випадок супроводжується рядком, що містить чотири нулі.

Вихідні дані

Для кожного тестового сценарію потрібно відобразити його номер. Потім відобразіть по одному рядку за запитом у тестовому сценарії. У кожному рядку відображається максимальна кількість кав'ярень, доступних за вказану відстань запиту m з подальшим оптимальним розташуванням. Наприклад, у наведеній нижче Табл. 2. показано, що 3 кав'ярні знаходяться на відстані 1 від оптимальної локації (3; 4), 4 магазини знаходяться в межах відстані 2 від оптимальної локації (2; 2), а 5 магазинів знаходяться на відстані 4 від оптимального місця розташування (3; 1). Якщо існує декілька оптимальних місць розташування, виберіть місце, яке знаходиться на самому півдні (мінімальна позитивна ціла y-координата). Якщо таким чином утворюється «петля», тоді виберіть місце, розташоване на самому заході (мінімальна позитивна ціла x-координата).

3. Побудувати систему таким чином, щоб була можливість поступового додавання одного тесту, який би не проходив, з наступною реалізацією такої частини задачі, щоб щойно написаний тест проходив.
4. При написанні тестів врахувати, що:
   - Модульні тести тестують кожен модуль окремо.
   - Тести, які використовуються при розробці через тестування, не повинні перетинати кордони процесу, використовувати мережеві з'єднання. В іншому випадку проходження тестів буде займати великий час, і розробники будуть рідше запускати набір тестів цілком.

Таблиця 2. Приклад вхідних і вихідних даних для задачі Coffee Central.

Вхідні дані	Вихідні дані
4 4 5 3 1 1 1 2 3 3 4 4 2 4 1 2 4 0 0 0 0	Case 1: 3 (3,4) 4 (2,2) 5 (3,1)

Вимоги до оформлення звіту

Звіт має включати:

1. Титульний аркуш.
2. Завдання на лабораторну роботу.
3. Хід роботи. Цей розділ складається з послідовного опису виконуваних кроків згідно інструкцій до лабораторної роботи.
4. Висновки.

Питання для самоперевірки

1. Чи можна виходити за межі модуля при написанні модульних тестів?
2. Чи може щойно написаний тест проходити при застосуванні іітехніки TDD?
3. Що таке mock-об’єкт?

4. Про що свідчать тести, які не проходять при додаванні нової функціональності?

Рекомендована література

1. Макконел С. Совершенный код. [Текст] / С. Макконел — СПб.: Питер, MSPress, 2005. — 893с. — ISBN 5-7502-0064-7.
2. Williams, L. & Kessler, R. Pair Programming Illuminated [Text] / Williams, L. & Kessler, R. - Boston: Addison-Wesley Professional, 2003.

ЛАБОРАТОРНА РОБОТА № 3. КІЛЬКІСНІ МЕТРИКИ КОДУ

Мета роботи: навчитися відрізняти і підраховувати фізичні, логічні рядки коду, а також рівень коментованості програм.

Завдання для лабораторної роботи

Для довільної бібліотеки / модуля на C # / Java / C ++ / Python необхідно створити утиліту, яка дозволяє розрахувати:

- Кількість рядків коду
- Кількість порожніх рядків
- Кількість фізичних і логічних рядків
- Показати # с коментарями і оцінку рівня коментування

Обрана бібліотека / модуль повинна бути обсягом не менше 50 KLOCs.

Результатом практичного завдання є виконувана утиліта і лістинг її коду.

Теоретичні відомості

Метрика програмного забезпечення - чисельна міра, що дозволяє оцінити певні властивості конкретної ділянки програмного коду.

Метрики потрібні для:

- розуміння і управління змінами;
- планування майбутніх проектів;
- порівняння одного продукту, процесу або організації з іншими;
- визначення відповідності стандартам;
- забезпечення основи для контролю.

Найбільшого поширення в практиці створення програмного забезпечення отримали розмірно-орієнтовані метрики. В організаціях, зайнятих розробкою програмної продукції для кожного проекту прийнято реєструвати наступні показники:

- загальні трудовитрати (в людино-місяцях, людино-годинах);
- обсяг програми (в тисячах рядках вихідного коду - KLOC);
- вартість розробки;
- обсяг документації;
- помилки, виявлені протягом року експлуатації;
- кількість людей, які працювали над продуктом;
- календарні терміни розробки.

На основі цих даних зазвичай підраховуються прості метрики для оцінки продуктивності праці (KLOC / людино-місяць) і якості продукту.

Ці метрики не універсальні та спірні, особливо це відноситься до такого показника як LOC, який істотно залежить від використовуваної мови програмування.

Кількість рядків коду (Source lines of code or lines of code / SLOC or LOC) - це метрика програмного забезпечення, що використовується для вимірювання його обсягу за допомогою підрахунку кількості рядків в тексті вихідного коду.

Як правило, цей показник використовується для прогнозу трудовитрат на розробку конкретної програми на конкретній мові програмування, або для оцінки продуктивності праці вже після того, як програма написана.

Кількість рядків вихідного коду є найбільш простим і поширеним способом оцінки обсягу робіт за проектом.

Залежно від того, яким чином враховується вихідний код, виділяють два основних показника SLOC:

- кількість «фізичних» рядків коду
- кількість «логічних» рядків коду

Кількість фізичних рядків коду -- Physical SLOC (використовувані абревіатури LOC, SLOC, KLOC, KSLOC, DSLOC) - визначається як загальне число рядків вихідного коду, включаючи коментарі і порожні рядки (при вимірюванні показника на кількість порожніх рядків, як правило, вводиться обмеження - при підрахунку враховується число порожніх рядків , яке не перевищує 25% загального числа рядків в вимірюваному блоці коду).

Кількість «логічних» рядків коду -– Logical SLOC (використовувані абревіатури LSI, DSI, KDSI, де «SI» - source instructions) - визначається як кількість команд і залежить від використовуваної мови програмування. У тому випадку, якщо мова не допускає розміщення кількох команд на одному рядку, то кількість «логічних» SLOC буде відповідати числу «фізичних», за винятком числа порожніх рядків і рядків коментарів. У тому випадку, якщо мова програмування підтримує розміщення кількох команд на одному рядку, то одна фізична рядок повинна бути врахована як кілька логічних, якщо вона містить більше однієї команди мови.

Вказівки до виконання лабораторної роботи

1. Кількість фізичних рядків коду потрібно визначити як загальне число рядків вихідного коду, включаючи коментарі і порожні рядки, але кількість порожніх рядків повинно бути обмежено 25% від загального числа рядків. В іншому випадку порожні рядки, кілість яких перевищує 25% від загального числа рядків, не враховуються.

2. Кількість логічних рядків коду потрібно вимірювати згідно з правилами, представленими у Таблиці 3.

Таблиця 3. Правила підрахунку логічних рядків коду

Структура	**Порядок врахування**	**Логічні рядки**
SELECTION STATEMENTS:	1	Рахувати як один логічний рядок. Вкладені висловлювання рахуються аналогічно.
if, else if, else, “?” operator, try , catch, switch		
ITERATION STATEMENTS:	2	Рахувати як один логічний рядок.
For, while, do..while		Ініціалізація, умови та інкремент в конструкції “for” не враховуються. Тобто for (i= 0; i < 5; i++)… Крім того, будь-які необов'язкові вирази в конструкції "for" також не враховуються, наприклад, for (i = 0, j = 5; i < 5, j > 0; i++, j--)… Дужки {...}, що додаються до ітераційних операцій та крапки з комою, що слідує "while" у структурі "do ... while", не враховуються.
JUMP STATEMENTS:	3	Рахувати як один логічний рядок.

Return, break, goto, exit, continue, throw		Мітки, що використовуються з операторами “goto”, не враховуються.
EXPRESSION STATEMENTS:	4	Рахувати як один логічний рядок.
Виклик функції, призначення, порожній оператор		Порожні висловлювання не впливають на логіку програми, і зазвичай служать заповнювачами місць (placeholder) або споживають CPU для встановлення термінів.
STATEMENTS IN GENERAL:	5	Рахувати як один логічний рядок.
Вирази, що закінчуються крапкою з комою		Крапка з комою в межах виразу «for» або як зазначено в розділі коментарів для заяви «do..while» не враховуються.
BLOCK DELIMITERS, BRACES	6	Враховувати як один логічний рядок пару фігурних дужок {}, крім випадку, коли за закриваючою дужкою слідує крапка з комою, тобто «};». Фігурні дужки, які використовуються для операторів вибору та ітерації, не враховуються. Визначення функції підраховується один раз, оскільки за ним слідує набір дужок.

COMPILER DIRECTIVE	7	Рахувати як один логічний рядок.
DATA DECLARATION	8	Рахувати як один логічний рядок. Включає прототипи функцій, оголошення змінної, оператори "typedef". Ключові слова, такі як "структура", "клас", не враховуються.

3. Метрика коментованості програм визначається за формулою:

$$F = \frac{\mathrm{Nc}}{\mathrm{N}_1},$$

де N_c - кількість коментарів в програмі; N_l - кількість рядків або операторів початкового тексту.

Таким чином, метрика *F* відображає насиченість програми коментарями. Виходячи з практичного досвіду прийнято вважати, що $F >= 0.1$, тобто на кожні десять рядків програми має припадати мінімум один коментар.

Перевірку можна здійснити за допомогою LocMetrics (www.locmetrics.com) або Unified Code Count (UCC, http://csse.usc.edu/ucc_wp/).

Вимоги до оформлення звіту

Звіт має включати:

1. Титульний аркуш.
2. Завдання на лабораторну роботу.
3. Хід роботи. Цей розділ складається з послідовного опису виконуваних кроків згідно інструкцій до лабораторної роботи.
4. Висновки.

Питання для самоперевірки

1. Чи змінюється кількість логічних рядків, якщо написати цикл for в однин рядок?
2. Яким чином буде рахуватись кількість рядків із коментарями у такій конструкції:

   *for (int i=0; /*comment*/ i<5; i++) {}*

3. Скільки логічних рядків має декларація порожнього класу?
4. Як враховуються логічні рядки у контрукціях switch ?

Рекомендована література

1. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ [Текст] / Г. Буч; 2-е изд.; пер. с англ. — М.: Издательства "Бином", СПб: "Невский диалект", 1998. — 560 с. — ISBN 0-8053-5340-2.
2. Орлов, С. А. Технологии разработки программного обеспечения [Текст] / С.А. Орлов. — СПб.: Питер, 2002. — 464 с. — ISBN 5-94723-145-X.
3. Graham, I. Object-Oriented Methods. Principles & Practice [Text] / I. Graham. 3rd Edition. — Addison-Wesley, 2000. — 864 pp. — ISBN 978-0201619133.

ЛАБОРАТОРНА РОБОТА № 4. ОБ'ЄКТНО_ОРІЄНТОВАНІ МЕТРИКИ КОДУ

Мета роботи: навчитися підраховувати та аналізувати метрики механізму об'єктно-орієнтованої парадигми: інкапсуляції, спадкування, поліморфізму і отриманих повідомлень.

Завдання для лабораторної роботи

Для довільної бібліотеки / модуля на C # / Java / C ++ / Python необхідно створити утиліту, яка дозволяє розрахувати об'єктно-орієнтовані метрики для окремих класів та для бібліотеки в цілому:

- Глибину дерева наслідування.
- Кількість нащадків класу.
- Метрики MOOD.

Обрана бібліотека / модуль повинна бути обсягом не менше 50 KLOCs.

Результатом практичного завдання є виконувана утиліта і лістинг її коду.

Теоретичні відомості

Об'єктно-орієнтовані метрики вводяться з метою:

- Поліпшити розуміння якості продукту.
- Оцінити ефективність розробки.
- Поліпшити якість роботи на етапі проектування.

Одними з найпоширеніших наборів об’єктно-орієнтованих метрик є метрики Чидамбера-Кемерера та метрики Фернанда Андреу (MOOD).

Метрики Чидамбера-Кемерера орієнтовані на классы, які є фундаментальним елементом об'єктно-орієнтованої системи. Тому вимірювання та метрики для окремого класу, ієрархії класів та співробітництва класів є безцінними для програмного інженера, який повинен оцінити якість проекту. Розглянемо кожну з метрик набору.

Метрика 1: Зважені методи на клас WMC (Weighted Methods Per Class) - відносна міра складності класу

Припустимо, в класі C визначені n методів зі складністю $c_1...,c_i,..., c_n$

Кількість методів і їх складність є індикатором витрат на реалізацію і тестування класів. Крім того, чим більше методів, тим складніше дерево

спадкування (всі підкласи успадковують методи їх батьків). З ростом кількості методів в класі його застосування стає все більш специфічним, тим самим обмежується можливість багаторазового використання. З цих причин метрика WMC повинна мати розумно низьке значення.

$$WMC = \sum_{i=1}^{n} c_i$$

Дуже часто застосовують спрощену версію метрики. При цьому припускають, що $C_i = 1$, і тоді WMC — кількість методів в класі.

Варіанти обліку методів в класі:

- Підраховуються тільки методи поточного класу. Успадковані методи ігноруються. Обгрунтування - успадковані методи вже підраховані в тих класах, де вони визначалися. Таким чином, інкрементність класу - кращий показник його функціональних можливостей, який відображає його право на існування.
- Підраховуються методи, визначені в поточному класі, і всі успадковані методи. Цей підхід підкреслює важливість простору станів в розумінні класу (а не інкрементного класу).

Існує ряд проміжних варіантів. Наприклад, підраховуються поточні методи і методи, прямо успадковані від батьків. Аргумент на користь даного підходу - на поведінку дочірнього класу найбільш сильно впливає спеціалізація батьківських класів.

Метрика 2: Висота дерева наслідування DIT (Depth of Inheritance Tree) - визначається як максимальна довжина шляху від листа до кореня дерева успадкування класів.

Для окремого класу DIT - це довжина максимального шляху від даного класу до кореневого класу в ієрархії класів.

- Висока ієрархія класів (велике значення DIT) призводить до більшої складності проекту, оскільки означає залучення більшої кількості методів і класів.
- Разом з тим, велике значення DIT має на увазі, що багато методів можуть використовуватися багаторазово.

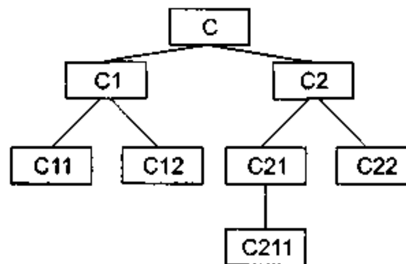

Рис. 8. Глибина дерева наслідування класів

Метрика 3: Кількість дітей NOC (Number of children) - кількість безпосередніх спадкоємців класу в ієрархії класів.

Зі збільшенням NOC зростає багаторазовість використання, однак послаблюється абстракція батьківського класу.

Крім того, кількість дітей характеризує потенційний вплив класу на проект. У міру зростання NOC зростає кількість тестів, необхідних для перевірки кожної дитини.

Метрики DIT і NOC - кількісні характеристики форми і розміру структури класів. Добре структурована об'єктно-орієнтована система частіше буває організована як ліс класів, ніж як надвисока дерево. На думку Г. Буча, слід будувати збалансовані по висоті і ширині структури успадкування: зазвичай не вище, ніж 7 ± 2 рівня, і не ширше, ніж 7 + 2 гілки.

Метрика 4: Зчеплення між класами об'єктів CBO (Coupling between object classes) — це кількість співробітництв, передбачених для класу, тобто кількість класів, з якими він з'єднаний. З'єднання означає, що методи даного класу використовують методи або екземплярні змінні іншого класу.

Інше визначення метрики має наступний вигляд: CBO дорівнює кількості зчеплень класу; зчеплення утворює виклик методу або властивості в іншому класі.

- З ростом CBO багаторазовість використання класу, ймовірно, зменшується. Очевидно, що чим більша незалежність класу, тим легше його повторно використовувати в інших програмах.
- Високе значення CBO ускладнює модифікацію і тестування, яке слід за виконанням модифікації. Зрозуміло, що, чим більше кількість зчеплень, тим вище чутливість всього проекту до змін в окремих його частинах. Мінімізація межоб'ектних зчеплень покращує модульність і сприяє інкапсуляції проекту.

CBO для кожного класу повинно мати розумно низьке значення. Це узгоджується з рекомендаціями щодо зменшення зчеплення стандартного програмного забезпечення.

Метрика 5: Відгук для класу RFC (Response For a Class)

Безліч відгуку класу **RS** - це безліч методів, які можуть виконуватися у відповідь на прибуття повідомлень в об'єкт цього класу :

$$RS = \{M\} U_{all_i} \{R_i\}$$

де $\{R_i\}$ — множина методів, що визиваються методом і, $\{M\}$ — множина всіх методів в класі.

Метрика RFC дорівнює кількості методів у множині відгуку, тобто дорівнює потужності цієї множини:

$$RFC = |RS|.$$

Інше визначення метрики: RFC — це кількість методів класу плюс кількість методів інших класів, що викликаються з даного класу.

Метрика RFC є мірою потенційної взаємодії даного класу з іншими класами, дозволяє судити про динаміку поведінки відповідного об'єкта в системі. Дана метрика характеризує динамічну складову зовнішніх зв'язків класів.

- Якщо у відповідь на повідомлення може бути викликано велику кількість методів, то ускладнюються тестування і налагодження класу, оскільки від розробника тестів потрібен більший рівень розуміння класу, зростає довжина тестової послідовності.
- З ростом RFC збільшується складність класу. Найгірша величина відгуку може використовуватися при визначенні часу тестування.

Метрика 6: Недолік зв'язності в методах LCOM (Lack of Cohesion in Methods)

Кожен метод всередині класу звертається до одного або декількох властивостей (екземплярним змінним). Метрика LCOM показує, наскільки методи не пов'язані один з одним через властивості (змінні). Якщо все методи звертаються до однакових властивостей, то LCOM = 0.

Вводяться позначення:

- NC - кількість пар методів без загальних екземплярність змінних;
- C — кількість пар методів із загальними екземплярними змінними.
- I_j — набір екземплярних змінних, що використовуються методом M_j
- $NC = ||\{I_{ij} | I_i \cap I_j = 0\}||$
- $C = ||\{I_{ij} | I_i \cap I_j \neq 0\}||$

$$LCOM = \max(0, NC - C)$$

LCOM — це кількість пар методів, не пов'язаних з властивостями класу, мінус кількість пар методів, що мають такий зв'язок.

- Зв'язність методів всередині класу повинна бути високою, оскільки це сприяє інкапсуляції. Якщо LCOM має високе значення, то методи слабо пов'язані один з одним через властивості. Це збільшує складність, в зв'язку з чим зростає ймовірність помилок при проектуванні.
- Високі значення LCOM означають, що клас, ймовірно, треба спроектувати краще (розбиттям на два або більше окремих класів). Будь-яке обчислення LCOM допомагає визначити недоліки в проектуванні класів, оскільки ця метрика характеризує якість упаковки даних і методів в оболонку класу.

Отже, зв'язність в класі бажано зберігати високу, тобто слід домагатися найнижчого значення LCOM.

MOOD - Metrics for Object Oriented Design

Основними цілями MOOD-набору є:

- покриття базових механізмів об'єктно-орієнтованої парадигми, таких як інкапсуляція, успадкування, поліморфізм, посилка повідомлень;
- формальне визначення метрик, що дозволяє уникнути суб'єктивності вимірювання;
- незалежність від розміру оцінюваного програмного продукту;
- незалежність від мови програмування, на якому написаний оцінюваний продукт.

1. Фактор закритості методу (Method Hiding Factor, MHF):

$$MHF = \sum_{i=1}^{TC} Mh_i / \sum_{i=1}^{TC} (Mv_i + Mh_i)$$

де Mv_i – кількість видимих методів класу,

Mh_i – кількість прихованих методів класу,

TC - кількість класів.

Зі збільшенням MHF зменшуються щільність дефектів у системі і витрати на їх усунення. Зазвичай розробка класу представляє собою покроковий процес, при якому до класу додається все більше і більше деталей (прихованих методів). Така схема розробки сприяє зростанню як значення MHF, так і якості класу.

2. Фактор закритості атрибута (Attribute Hiding Factor, AHF)

Нехай $A_h(C_i)$- кількість прихованих атрибутів класу C_i (інтерфейс класу), $A_d(C_i)$ – загальна кількість атрибутів, визначених у класі C_i (без врахування успадкованих), TC – загальна кількість класів.

Тоді фактор закритості атрибута AHF буде обчислюватися за такою формулою:

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

В ідеальному випадку всі атрибути повинні бути відкриті і доступні тільки для методів відповідного класу (AHF = 100%).

3. Фактор наслідування методу (Method Inheritance Factor, MIF)

Нехай $M_i(C_i)$-кількість внаслідуваних та неперевизначених методів класу C_i, $M_a(C_i)$-кількість всіх методів, доступних в класі C_i. Тоді:

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

Значення MIF = 0 вказує, що в системі відсутнє ефективне спадкування, наприклад, все успадковані методи перевизначені.

Зі збільшенням MIF зменшується щільність дефектів і витрати на виправлення помилок. Дуже великі значення MIF (70-80%) призводять до зворотного ефекту, але цей факт потребує додаткової експериментальної перевірки.

4. Фактор наслідування властивості (Attribute Inheritance Factor, AIF)

Нехай $A_i(C_i)$- кількість наслідуваних та неперевизначених атрибутів класу C_i, $A_a(C_i)$ – загальна кількість атрибутів, визначених в класі C_i, TC – загальна кількість класів.

Тоді фактор наслідування властивості AIF буде дорівнювати:

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

5. Фактор поліморфізму (Polymorphism Object Factor, POF)

Нехай $M_o(C_i)$-кількість внаслідуваних та перевизначених методів класу C_i, $M_n(C_i)$- кількість нових методів, доступних в класі C_i, DC - кількість нащадків класу C_i.

Тоді фактор поліморфізму POF буде дорівнювати:

$$POF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC} M_n(C_i) * DC(C_i)}$$

Помірне використання поліморфізму зменшує як щільність дефектів, так і витрати на доопрацювання. Однак при POF> 10% можливий зворотний ефект.

6. Фактор зчеплення (Coupling Factor, COF)

Зчеплення фіксує наявність між класами відносини клієнт-постачальник (client-supplier), тобто клас-клієнт містить щонайменше одне не успадкуване

посилання на атрибут або метод класу-постачальника. Наявність відношення визначається за формулою :

$$is_client(C_c, C_s)\begin{cases}1 & if \quad Cc \Rightarrow Cs \cap Cc \neq Cs \\ 0 & else\end{cases}$$

Тоді, якщо позначити загальну кількість класів - TC, то

$$COF = \frac{\sum_{i=1}^{rc}\left[\sum_{j=1}^{rc} is_client(C_i, C_j)\right]}{TC^2 - TC}$$

Зчеплення негативно впливають на якість ПЗ - збільшують складність, зменшує інкапсуляцію і можливості повторного використання.

Кожна з цих метрик відноситься до основного механізму об'єктно-орієнтованої парадигми: інкапсуляції (MHF і AHF), спадкування (MIF і AIF), поліморфізму (POF) і отриманих повідомлень (COF). У визначеннях MOOD не використовуються специфічні конструкції мов програмування.

Вказівки до виконання лабораторної роботи

1. Обрати мову програмування
2. Для бібліотеки, написаної обраною мовою програмування, необхідно проаналізувати
3. Для кожного класу проаналізувати глибину дерева наслідування та відобразити на екрані.
4. Для кожного класу проаналізувати кількість нащадків для кожного класу.
5. Для всіх класів підсумувати кількість закритих атрибутів та методів та вивести фактори закритості атрибутів та методів на екран.
6. Для всіх класів бібліотеки підсумувати кількість внаслідуваних та перевизначених атрибутів та методів та вивести фактори наслідування атрибутів та методів на екран.

7. Для всіх класів бібліотеки підсумувати кількість внаслідуваних та перевизначених атрибутів та методів та вивести фактори наслідування атрибутів та методів на екран.
8. Для всіх класів бібліотеки підсумувати кількість внаслідуваних та перевизначених методів класу та кількість нови методів класів та вивести фактор поліморфізму на екран.
9. Підрахувати фактор зчеплення та вивести на екран.

Вимоги до оформлення звіту

Звіт має включати:

1. Титульний аркуш.
2. Завдання на лабораторну роботу.
3. Хід роботи. Цей розділ складається з послідовного опису виконуваних кроків згідно інструкцій до лабораторної роботи.
4. Висновки.

Питання для самоперевірки

1. Чи можна перевизначити атрибут класу і в якій мові програмування?
2. Яким чином рахувати фактор поліморфізму, якщо кількість нащадків дорівнює нулю?
3. Про що свідчить високе число DIT?
4. Про що свідчить низьке число метрики MIF?
5. Про що свідчить низьке значення фактору закритості атрибуту і як його підвищити?

Рекомендована література

1. Chidamber, S. R. A Metrics Suite for Object Oriented Design [Text] /

S.R. Chidamber, C. F. Kemerer // IEEE Transactions on Software Engineering. — June 1994. — vol. 20, No. 6. — pp. 476-493.

2. Graham, I. Object-Oriented Methods. Principles & Practice [Text] / I. Graham. 3rd Edition. — Addison-Wesley, 2000. — 864 pp. — ISBN 978-0201619133.
3. Hitz, M. Measuring Coupling in Object-Oriented Systems. [Text] / M. Hitz, B. Montazeri // Object Currents. — Apr 1996. — vol. 2, No. 4. — 17 pp.
4. Lorenz, M. Object-Oriented Software Metrics [Text] / M. Lorenz, J. Kidd. — Prentice Hall, 1994. — 146 p. — ISBN 978-0131792920.