

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ, КОМП'ЮТЕРНОЇ ТА ПРОГРАМНОЇ
ІНЖЕНЕРІЇ**

Кафедра _____ комп'ютеризованих систем управління _____

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____ Литвиненко О.Є.

«____» _____ 2020 р.

ДИПЛОМНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

**ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ
"МАГІСТР"**

Тема: _____ Система оцінки якості програмного коду веб-проектів _____

Виконавець: _____ Ситник Л.Ю. _____

Керівник: _____ Нечипорук В.В. _____

Нормоконтролер: _____ Тупота Є.В. _____

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютеризованих систем управління
Освітнього ступеня магістр
Спеціальність 123 "Комп'ютерна інженерія"
(шифр, найменування)
Спеціалізація 123.02 "Системне програмування"
(шифр, найменування)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Литвиненко О. Є.
« » 2020 р.

ЗАВДАННЯ на виконання дипломної роботи (проекту)

Ситника Леоніда Юрійовича
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. **Тема роботи:** "Система оцінки якості програмного коду веб-проектів"
затверджена наказом ректора від " 27 " серпня 2020 року № 1203 /ст.
2. **Термін виконання роботи:** з 05.10.2020 до 31.12.2020
3. **Вихідні дані до роботи:** аналізатор програмного коду веб-проектів
4. **Зміст пояснювальної записки (перелік питань, що підлягають розробці):**
 - 1) принципи оцінювання якості кода в програмній інженерії;
 - 2) методи оцінки якості програмного коду;
 - 3) розробка програмного забезпечення для оцінки якості коду.
5. **Перелік обов'язкового графічного матеріалу:**
 - 1) основний фрейм реалізованої програми;
 - 2) інформація, показана для метрики LOC;
 - 3) інформація, показана для метрики NOC;
 - 4) звіт про файл JAVA для метрик LOC та NOC у фізичних рядках коду;
 - 5) схема алгоритму визначення метрик рядків коду.

6. Календарний план

| № п/п | Етапи виконання дипломної роботи | Термін виконання етапів | Примітка |
|----------|---|----------------------------|----------|
| 1 | Ознайомитись з постановкою задачі дипломного проектування | 05.10.2020 – 06.10.2020 | |
| 2 | Вивчити спеціальну літературу і технічну документацію щодо аналізу якості коду <i>web</i> -проектів | 07.10.2020 – 10.11.2020 | |
| 3 | Провести аналіз методів аналізу якості коду <i>web</i> -проектів | 11.11.2020 – 14.11.2020 | |
| 4 | Написати і налагодити програму | 15.11.2020 – 19.11.2020 | |
| 5 | Написати пояснювальну записку | 29.11.2020 – 13.12.2020 | |
| 6 | Підготувати графічний і демонстраційний матеріали | 21.12.2020 – 25.12.2020 | |

7. Дата видачі завдання _____ 05.10.2020 _____

Керівник _____ Нечипорук В.В.
(підпис)

Завдання прийняв до виконання _____ Ситник Л.Ю.
(підпис студента)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи “Система оцінки якості програмного коду веб-проектів”: 93 с., 6 рис., 25 літературних джерел, 1 додаток.

ЯКІСТЬ ПРОГРАМНОГО КОДУ, ВЕБ-ПРОЕКТ, ОЦІНКА ЯКОСТІ,
ПРОГРАМНИЙ ПРОЕКТ, ОБ'ЄКТНО-ОРІЄНТОВАНІ МЕТРИКИ, КОД
ДЖЕРЕЛА

Мета дослідження – розробити систему оцінки якості коду веб-проектів.

Об'єкт дослідження – програмний код веб-проектів.

Предмет дослідження – система оцінки якості програмного коду веб-проектів.

Метод дослідження – аналіз, синтез, моделювання і програмна реалізація системи оцінки якості програмного коду веб-проектів.

Встановлено, що запропонований метод аналізу коду проектів, який базується на багатопараметричній обробці рядків коду, є новим підходом до аналізу якості програмного коду.

Результати дипломної роботи рекомендується використовувати при розробці програмних модулів веб-проектів для оцінки їх якості. Розвиток проекту передбачається за рахунок розширення групи параметрів оцінки якості коду.

Публікація: Ситник Л.Ю. Система оцінки якості програмного коду веб-проектів// Тези доповідей наук.-практ. конф. “Сучасні тенденції розвитку системного програмування” (25-26 листопада 2020 р.). – К.: НАУ, 2020. – С. 28.

ЗМІСТ

| | |
|--|----|
| ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ | 7 |
| ВСТУП | 8 |
| РОЗДІЛ 1 ПРИНЦИПИ ОЦІНЮВАННЯ ЯКОСТІ КОДА В ПРОГРАМНІЙ ІНЖЕНЕРІЇ..... | 11 |
| 1.1. Оцінювання якості кода через відповідність правилам | 11 |
| 1.2. Зв'язок якості і системи оцінки точок зору. | 14 |
| 1.3. Стандарти в області якості | 15 |
| 1.4. Вимоги до розробника ПЗ..... | 20 |
| 1.5. Критерії якості програмного забезпечення..... | 21 |
| 1.6. Висновки до розділу..... | 22 |
| РОЗДІЛ 2 МЕТОДИ ОЦІНКИ ЯКОСТІ ПРОГРАМНОГО КОДУ | 24 |
| 2.1. Зовнішні чинники якості..... | 24 |
| 2.2. Внутрішні чинники якості | 25 |
| 2.3. Стратегії тестування..... | 44 |
| 2.4. Методи тестування програмного забезпечення..... | 47 |
| 2.5. Методи стратегії «білого ящика». | 49 |
| 2.6. Тестування методом чорного ящика | 54 |
| 2.7. Рефакторинг і блокові тести | 56 |
| 2.8. Висновки до розділу..... | 65 |
| РОЗДІЛ 3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ОЦІНКИ ЯКОСТІ КОДУ | 67 |
| 3.1. Розробка метричних лічильників | 67 |

| | |
|---|----|
| 3.2. Розробка системи автоматизованих вимірювань метрик..... | 69 |
| 3.3. Дослідження та порівняльна характеристика даного додатка з відповідними професійними інструментами..... | 85 |
| 3.4. Висновки до розділу..... | 87 |
| ВИСНОВКИ | 88 |
| СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ..... | 91 |
| ДОДАТОК А..... | 94 |

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

| | |
|-------------|---|
| <i>OOAD</i> | – <i>Object-oriented analysis and design.</i> |
| <i>OOD</i> | – <i>Object-oriented design.</i> |
| <i>OOM</i> | – <i>Object-oriented metrics.</i> |
| <i>WMC</i> | – <i>Weighted Methods per Class.</i> |
| <i>DIT</i> | – <i>Depth of Inheritance Tree.</i> |
| <i>NOC</i> | – <i>Number of children.</i> |
| <i>CBO</i> | – <i>Coupling between object classes.</i> |
| <i>RFC</i> | – <i>Response for a Class.</i> |
| <i>LCOM</i> | – <i>Lack of Cohesion in Methods.</i> |
| <i>MOOD</i> | – <i>Metrics for OOD.</i> |
| <i>MHF</i> | – <i>Method Hiding Factor.</i> |
| <i>AHF</i> | – <i>Attribute Hiding Factor.</i> |
| <i>MIF</i> | – <i>Method Inheritance Factor.</i> |
| <i>AIF</i> | – <i>Attribute Inheritance Factor.</i> |
| <i>PF</i> | – <i>Polymorphism Factor.</i> |
| <i>CF</i> | – <i>Coupling Factor.</i> |
| <i>CC</i> | – <i>Cyclomatic Complexity.</i> |
| <i>CP</i> | – <i>Comment Percentage</i> |
| <i>IC</i> | – <i>Intelligence Content.</i> |

ВСТУП

На сьогоднішній день в Україні та в усьому світі програмування є основною технологією. Без програмування неможливо розвивати науку та техніку.

Однією з важливих частин програмування є зменшення зайвого коду та його оптимізація для створення ефективних та високопродуктивних програм. Якість вихідного коду залежить від багатьох факторів, наприклад, мови програмування, знань та досвіду програміста та багатьох інших. Аналіз вихідного коду показує, що основна частина його якості залежить від дотримання програмних показників.

Компілятори вихідного коду призначені для пошуку лише помилок, а частина, що відповідає метрикам, залежить лише від програміста.

Практика показує, що необхідне термінове вивчення метрик програмування, щоб пришвидшити процес підготовки висококваліфікованих фахівців. Ось чому проблема створення програмного забезпечення, що відповідає метрикам, є актуальною.

В цілому вже давно загальновизнано, що поняття "поганий / хороший код" пов'язано навіть не стільки з ефективністю використання обчислювальних ресурсів (швидкодія програми, обсяг займаної оперативної пам'яті та ін.), Скільки з завданнями налагодження і модифікації ПЗ (на етапах як власне розробки, так і супроводу). У цій ситуації якість коду визначається такими показниками, як правильне розбиття програми на модулі, обмеження у використанні потенційно ризикованих мовних конструкцій, наочне оформлення вихідного коду, тощо.

Однак навіть визначивши склад ключових показників (метрик) якості коду, практично неможливо створити універсальні критерії, що дозволяють вважати програму "поганий" або "хорошою". У будь-якому випадку така оцінка носить занадто суб'єктивний характер, і навіть для одного програміста вона буде варіюватися в залежності від типу проекту. Тому існуючі інструменти

визначення метрик коду (*code metrics, CM*) в основному обмежуються обчисленням відповідних значень, інтерпретація яких повністю покладається на людину.

Раніше для виконання подібних завдань в середовищі *Microsoft Visual Studio* потрібно було використовувати додаткові кошти третіх фірм. Але тепер CM-функції з'явилися в бета-версії *Visual Studio 2008 Team Edition for the Software Developer*, яка повинна вийти на ринок в остаточному вигляді на початку наступного року. Ось за якими метриками коду можна стежити вже зараз:

- індекс експлуатаційної надійності (*Maintainability Index, MI*)- комплексний показник якості коду (від 0 до 100 – чим вище, тим краще); методика його визначення розроблена фахівцями *Carnegie Mellon Software Engineering Institute*;

- циклічна складність (*Cyclomatic Complexity, CC*)- показник, що характеризує число гілок в програмному коді і обчислюється шляхом підрахунку операторів циклу, умовного переходу і перемикачів;

- глибина успадкування (*Depth of Inheritance*)- характеризує довжину ланцюжків спадкування в програмному коді;

- зчеплення класів (*Class Coupling*)- відображає ступінь залежності класів між собою (в тому числі наявність загальних даних, об'єктів і ін.);

- число рядків коду- тут все очевидно: чим більше рядків, тим складніше програма.

Звичайно, для більш детального аналізу якості коду бажано використовувати більше показників, які можна визначати за допомогою засобів третіх фірм (є чимало і безкоштовних продуктів). Але, як правило, більш широкий спектр характеристик потрібно тільки в навчальних цілях і не застосовується розробниками на практиці.

Перш за все, слід розглянути кількісні характеристики вихідного коду програм (на увазі їх простоти). Найелементарнішої метрикою є кількість рядків коду (*SLOC*). Дана метрика була спочатку розроблена для оцінки трудовитрат за проектом. Однак через те, що одна і та ж функціональність може бути розбита на

кілька рядків або записана в один рядок, метрика стала практично непридатною з появою мов, в яких в один рядок може бути записано більше однієї команди. Тому розрізняють логічні і фізичні рядки коду. Логічні рядки коду - це кількість команд програми. Даний варіант опису так само має свої недоліки, так як сильно залежить від використовуваної мови програмування і стилю програмування

З огляду на вищесказане визначено мету, об'єкт та предмет дослідження:

Мета дослідження – розробити систему оцінки якості коду веб-проектів.

Об'єкт дослідження – програмний код веб-проектів.

Предмет дослідження – система оцінки якості програмного коду веб-проектів.

РОЗДІЛ 1

ПРИНЦИПИ ОЦІНЮВАННЯ ЯКОСТІ КОДА В ПРОГРАМНІЙ ІНЖЕНЕРІЇ

При оцінюванні якості коду в першу чергу треба з'ясувати за якими параметрами визначають якість коду та для чого це необхідно. У програмуванні в більшості випадків, для визначення параметрів досить визначити наступні характеристики:

- відповідність правилам;
- складність коду;
- дублювати;
- коментування;
- покриття тестами.

1.1. Оцінювання якості кода через відповідність правилам

Під цей пункт підпадають ситуації коли код компілюється і виконує команду правильно. Це необхідна характеристика і залежить від правила написання коду. В *Java Code Conventions*, *GCC Coding Conventions*, *Zends Coding Standard* є готові рішення, але за необхідності можна доповнити їх своїми, більш підходящими для специфіки завдання.

Існує кілька типів правил:

- синтаксичні правила- до них відносять стиль іменування змінних (*camelCase*, через підкреслення), констант (*uppercase*), методів, стиль написання фігурних дужок і чи потрібні вони якщо в блоці тільки один рядок коду. При написанні коду його легко читати, бо він знає свій власний стиль. Але варто дати код, де використовується інша нотація і дужки з нового рядка, доведеться додаватково виділяти ресурси на сприйняття нового стилю.

- правила підтримки коду- правила, які повинні сигналізувати що код занадто складний і його буде важко реалізовувати. Наприклад, індекс складності методу або класу занадто великий або занадто багато рядків коду в методі,

наявність дублікатів в коді або "*magic numders*". Це все вказує на складні місця, які складно буде супроводжувати. Необхідно пам'ятати, що при створенні коду маємо враховувати індекс складності

– очищення і оптимізація коду. Сюди відносять зайві обсяги імпорту, змінні і методи які вже не використовуються, але через необхідність їх залишили в коді.

1.1.1. Цикломатична складність коду.

Характеристика, від якої безпосередньо залежить складність підтримки коду. В данному випадку виділити метрику складніше ніж у попередній характеристиці. Вона залежить від кількості вкладених операторів розгалуження і циклів. Чим індекс нижче тим краще, і тим простіше міняти структуру коду. Варто вимірювати складність методу, класу, файлу. Значення цієї метрики треба обмежити певним граничним числом. Наприклад цикломатична складність методу не повинна перевищувати 10, інакше потрібно спростити або розбити його.

1.1.2. Дублікати.

Важлива характеристика, яка відображає наскільки легко можна буде вносити зміни в код. Метрику можна означити у відсотках, як співвідношення рядків дублікатів до всіх рядків коду. Чим менше дублікатів, тим легше корегувати код.

1.1.3. Коментування

Це індивідуальна характеристика, яка виходить зі специфіки завдання. Цілей та вихідного продукту. Для невеликих проектів коментування не є необхідним параметром. В коментуванні виділяють наступні характеристики:

– ставлення коментарів до всього коду- з цієї метрики можна зробити висновки наскільки детальні коментарі та наскільки вони можуть бути корисними. Звичайно, з цієї метрики можна сказати чи є коментарі "цикл" перед циклом, але це потрібно виправляти коли проводиться ревію.

– коментування публічних методів- ставлення коментованих публічних методів до загальної їх кількості. Так як публічні методи використовуються поза межами класу або пакета, то краще прокоментувати що цей метод повинен робити і на що може вплинути. Кількість публічних методів без коментаря має прагнути до нуля.

1.1.4. Покриття тестами

Рівень покриття зчитується як відношення кількості покритих тестами елементів коду до кількості всіх існуючих. Залежно від того, що означити як елемент коду, часто виділяють наступні типи покриття:

- покриття файлів- найчастіше файл покритий, якщо тест потрапив в файл і виконав хоча б один рядок коду з файлу. Така метрика використовується рідко.
- покриття класів- характеристика аналогічна з покриттям файлів, тільки покриття класів.
- покриття методів- це спосіб обчислення метрики. За допомогою цієї метрики можна швидко знайти код, що не відповідає правилам.
- покриття рядків- одна з найбільш використовуваних метрик по покриттю. Той же спосіб обчислення, тільки за об'єкт береться рядок.
- покриття розгалужень- за елемент береться розгалуження. З цієї метрики можна дізнатися про комплементарність коду до покриття тестами.
- сумарне покриття- метрика покриття при якій в розрахунках береться до уваги не один елемент, а декілька. Найбільш часто використовують сумарне покриття рядків і розгалужень.

1. Поняття якості програмного забезпечення.

Якість програмного забезпечення – залежить від того чим є програмне забезпечення: продуктом або послугою. Існують дві точки зору:

1. Системний адміністратор самостійно реалізує програмне забезпечення і виводить його на ринок. В такому випадку ПЗ – це продукт.
2. Системний адміністратор отримує завдання від Замовника – надання послуги з розробки програмного забезпечення.

Для вирішення цієї проблеми необхідно визначити, хто головний у відносинах Замовник (Споживач) / Розробник. Безумовно, будь-яка програмна система призначена для кінцевого користувача. Тоді розробники виробляють послуги, а не товар, причому саме виробляють, а не роблять, з усіма додатковими складнощами виробництва, такими як документування.

Показовим є система оцінок якості програмного забезпечення на міжнародних і національних конкурсах. Наприклад, при аналізі виробів, висунутих на премію *Malcolm Bridge*, враховуються 20 критеріїв із загальною сумою балів 1000, з них лише 250 балів припадають на технічні аспекти якості.

Розробник має реалізує весь цикл програмного забезпечення. Але часто занадто зосереджується на безпосередньої реалізації програми, не звертаючи уваги на складний життєвий цикл.

1.2. Зв'язок якості і системи оцінки точок зору.

Незважаючи на те, що розробка програмного забезпечення – це саме послуга, в сучасному світі програмне забезпечення – це продукт..

Головний критерій якості, з точки зору комерційного споживача програмного забезпечення обчислювальної системи – її здатність підвищити комерційну ефективність всієї фірми. Це критерій якості з точки зору керівника і власника фірми. З точки зору розробника / інженера, головне – це підвищити загальну ефективність фірми.

З точки зору програмування, програмне забезпечення повинно бути ефективним на всіх напрямках діяльності. Крім того, сьогоdnішня організація неможлива без обчислювальних систем з різним програмним забезпеченням, яке дозволяє аналізувати діяльність підприємства в цілому і по відділах, оптимально планувати виробництво і ресурси, враховувати робочий час і управляти роботою людей.

У найбільш загальному випадку ефективність підприємства – це система якості цього підприємства. На сьогоdnішній день системи якості тісно переплетені з програмним забезпеченням. З одного боку, програмне забезпечення

підпорядковується законам системи якості, з іншого боку, забезпечує її функціонування.

Оскільки велика кількість проектів в галузі впровадження програмного забезпечення можна визнати невдалими, таке ж питання і відповідь з'являються стосовно систем менеджменту якості програмного забезпечення та оцінки якості програмних продуктів взагалі. Програмні проекти зазнають невдачі, як правило, на стадії впровадження, оскільки зустрічають негативний відгук з боку кінцевих користувачів системи.

Найголовніший погляд на якість – це точка зору кінцевого користувача. До нього можна віднести і якість офісної роботи, повноту інформації, точність даних, стійкість програм і даних по відношенню до призначених для користувача помилок і збоїв в обчислювальних системах.

1.3. Стандарти в області якості

В цілому система якості підприємства регламентується цілим сімейством стандартів ISO-9000. Відносини в області якості програмного забезпечення обчислювальних систем регламентуються стандартом ISO-9126.

ISO 9126 пропонує розглядати дві точки зору:

1. замовника;
2. розробника.

У першому випадку обов'язково оформляти повне технічне завдання, аналізувати предметну область і погоджувати з кінцевим користувачем, рекомендується аналізувати також і точку зору розробника.

У другому випадку все розпливчасто. Фактично декларується тільки відповідальність розробників за вироблене програмне забезпечення обчислювальних систем, яке повинно задовольняти вимогам якості. Розробник повинен бути зацікавлений в якості проміжних результатів розробки програмного забезпечення, також як і в якості результату. З точки зору розробника, найбільш широко застосовуються такі критерії: технічне якість роботи (швидкодія, надійність), придатність до супроводу і розвитку, стійкість.

Стандарт дозволяє пред'являти і інші вимоги та критерії якості, що не суперечать головним вимогам стандарту. Приклад таких вимог і критеріїв:

1. функціональність;
2. відповідність призначенню;
3. точність;
4. здатність взаємодіяти з середовищем;
5. відповідність нормам;
6. інформаційна безпека;
7. надійність;
8. зрілість;
9. відмовостійкість;
10. здатність відновлюватися після збоїв;
11. придатність до використання;
12. зручність і простота в роботі;
13. ефективність;
14. швидкодію і час відгуку;
15. споживання ресурсів;
16. супроводжуваність;
17. аналізованість (діагностика причин помилок і зіставлення з вихідним кодом);
18. придатність до змін;
19. стабільність;
20. тестованість;
21. переносимість;
22. адаптованість;
23. легкість інсталяції;
24. відповідність нормам по переносимості та інсталяції;
25. замінність.

Незважаючи на те, що стандарт пред'являє вимоги до якості всіх етапів розробки програмного забезпечення, що пред'являються характеристики зачіпають лише якість програмного забезпечення як продукту.

Кілька більш просунутий підхід до якості взагалі і програмного забезпечення зокрема застосовується в системі стандартів Японії. Для опису процесу створення якості використовується «спіраль пізнання» (рис. 3).

- через усупільнення, шляхом опису інтуїтивних відчуттів по аналогії з іншими знаннями;
- узагальненням різноманітного досвіду і пошуком закономірностей;
- виробленням мов, систем і теорій для точного і явного вираження знання.



Рис. 1.1. Спіраль створення та накопичення знань

Дана концепція звичайно застосовується і до системи якості та критеріям, використовуваним в процесі створення програмного забезпечення обчислювальної системи. Пропонується спіраль розвитку технологій розробки та оцінки якості програмного забезпечення обчислювальної системи в цілому, показана на рис. 1.2.

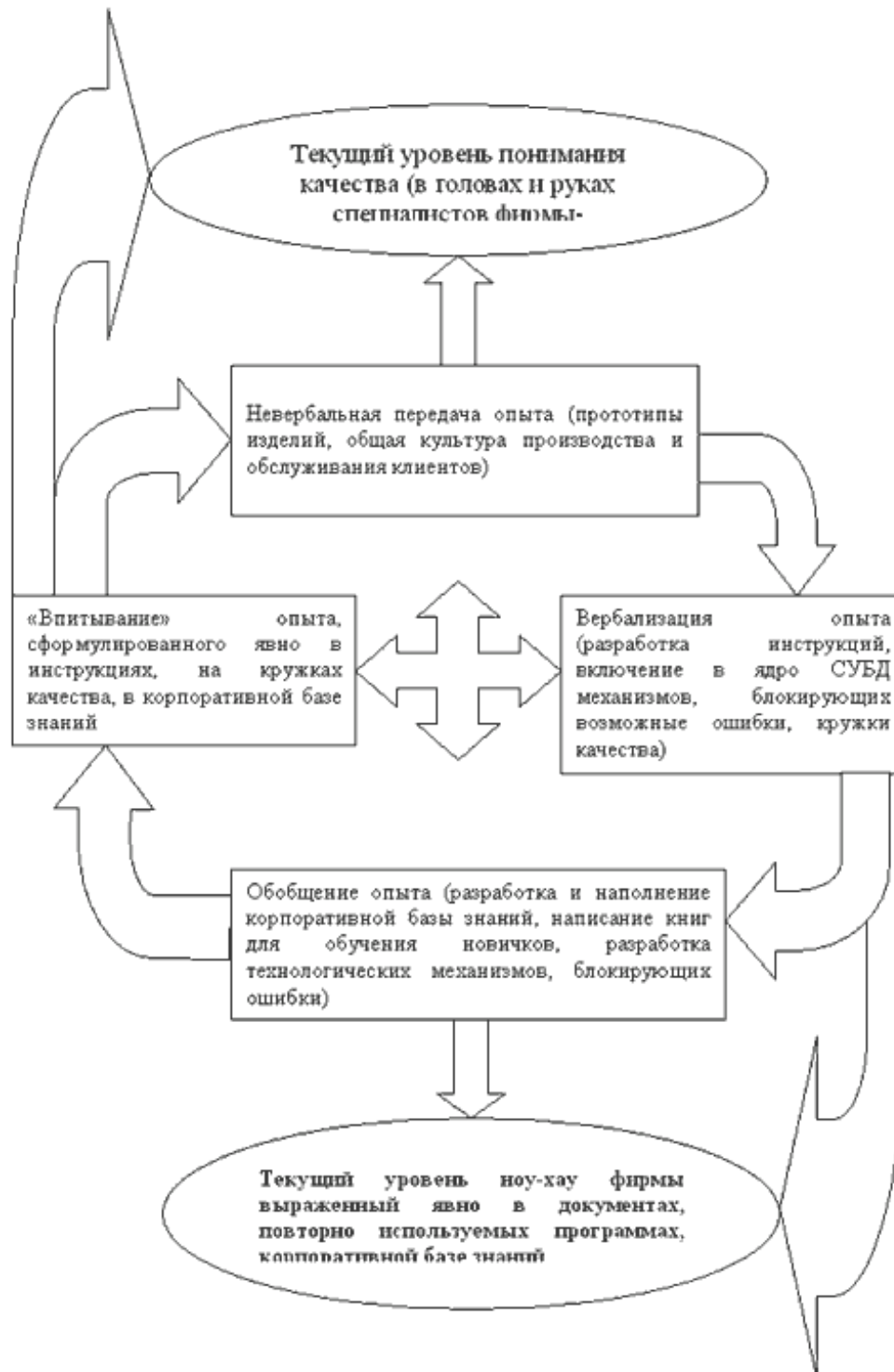


Рис. 1.2. Спіраль розвитку технологій і критеріїв якості програмного забезпечення обчислювальної системи.

З спіралі слід, що в процесі роботи індивідуальні поняття і почуття розробників про критерії якості та їх значення в якості програмного забезпечення обчислювальної системи повинні бути доступні всім учасникам процесу, в тому числі керівникам. Крім того, всі поняття, відчуття, почуття повинні документуватися у вигляді стандартів, описів, бібліотек програм і компонентів, технологій. У цьому випадку вони стають «знаннями» і представляють новий рівень мислення і нові технології і дозволяють підняти загальну культуру роботи і мінімізувати ризики, пов'язані з людським фактором.

Далі підхід запускається по спіралі. Отриманий рівень мислення і технологій дозволяє побачити і узагальнити нові поняття і думки в області якості на ще більш високому рівні. І так далі, коло замикається, спіраль оцінки якості розкручується.

Таким чином, основний метод оцінки якості програмного забезпечення обчислювальних систем, який використовують замовники – кінцеві користувачі – це оцінки зрілості фірми-розробника, оскільки універсальних методів оцінки програмного забезпечення немає і методів, які може використовувати замовник при оцінці якості замовляється програмного забезпечення ще до його замовлення, практично ні.

Відповідно до вимог стандарту ISO 9126 [14] існує два підходи, яким можна слідувати для забезпечення якості програмного забезпечення:

1. переконатися в якості на кожному етапі розробки;
2. оцінити якість кінцевого продукту.

При цьому виділити головний підхід не представляється можливим.

Слід враховувати той факт, що існує тільки один надійний спосіб перевірити якість замовляється програмного продукту – це його використання, тобто «Спробувати». Природно, у випадку з програмним забезпеченням його немає, так як «спробувати» програмне забезпечення можна тільки шляхом його розробки. Тому існує тільки один спосіб – це оцінка розробника, оцінка як довго він працює і як якісно він працює.

1.4. Вимоги до розробника ПЗ

Отже, з вищевикладеного випливає що для розробника програмного забезпечення найбільш важливим критерієм є вимоги до нього самого. Як уже зазначалося, чітких вимог тут немає.

Вимоги та їх значення формує безпосередньо ринок. Розробник повинен бути:

- зрілим;
- фінансово самостійним;
- надійним;
- уважним до клієнта.

Зрілість.

Організація в своєму розвитку повинна дійти до поділу праці, тобто різні види програмного забезпечення та його компоненти повинні проводитися окремо один від одного. Виробництво здійснюється із застосуванням спеціальних програмних засобів. Необхідна наявність діючих і сертифікованих систем управління якістю, як мінімум однієї для впровадження та іншої – для виробництва програмного забезпечення обчислювальних систем. Обов'язкова наявність власної корпоративної інформаційної системи, в рамках якої функціонує база знань фірми. Іншими словами, зрілість – це проходження організацією декількох витків спіралі якості (див. Рис. 3 і 4) з документально оформленими результатами, переданими в руки фахівців.

Фінансова самостійність.

Розробник є юридичною особою зі значною історією існування, володіє власним потужним виробництвом і не є «групою товаришів» або «франчайзі», тобто володіє «правом робити налаштування».

Надійність.

Розробник надає на свою продукцію гарантію і здатний підтвердити виконання гарантійних зобов'язань.

Бути уважним.

Наявність впровадженої технології роботи з клієнтом: прийом заявок, диспетчеризація, нормовані форми листування, терміни відповідей і підготовки технічних завдань на зміни, терміни доробок і їх впровадження.

1.5. Критерії якості програмного забезпечення

Вище було показано, що якість і його оцінка у випадку з програмним забезпеченням обчислювальних систем – це складне, неформалізованих і не має однозначної оцінки поняття. Проте, галузь «управління і оцінка якості» складається, і як наслідок формуються необхідні поняття і уявлення на основі наукових праць, стандартів корпорацій і галузевих державних стандартів.

Всі критерії якості діляться на «зовнішні», які може виявити користувач програмного забезпечення, і «внутрішні», які бачать тільки розробники, що створюють це програмне забезпечення.

Зовнішні фактори, які в тому числі описані в стандарті ISO 9126:

1. коректність
2. стійкість
3. розширюваність
4. повторне використання
5. сумісність, ефективність
6. переносимість
7. простота використання
8. функціональність
9. своєчасність

Необхідно відзначити, що терміни «програмне забезпечення» і «кінцевий користувач» потрібно розуміти не тільки як «кінцеві прикладні програми» і «оператор або користувач обчислювальної системи», а й як «бібліотеки» і «розробник, що використовує бібліотеку». Отже, такі фактори, як розширюваність і повторне використання, не помітні кінцевому користувачеві.

На сьогоднішній день більша частина створюваного програмного забезпечення обчислювальних систем в світі – це бібліотеки і *API* (до таких

продуктів відносяться не тільки бібліотеки стандартних класів, такі як *STL* або *Collection Framework*, а й, наприклад, ядра операційних систем) і вони не є самостійними програмними продуктами.

Таким чином, навіть якщо розробник поодиночі працює над програмним забезпеченням, він все одно використовує величезну кількість програмного забезпечення, вироблене до нього. При роботі в команді все одно створюється велика кількість бібліотек програмного забезпечення, які використовуються або будуть використовуватися колегами або самим розробником.

Отримані бібліотеки можуть бути не настільки жорстко регламентовані, як *API* ядра операційної системи, яке дуже небезпечно міняти від версії до версії. Однак вимоги до якості проектування внутрішніх інтерфейсів ніхто не відміняв, і чим краще вони будуть розроблені, тим менше проблем виникає при їх використанні. Крім того, багато програм мають зовнішні бібліотеки для розширення (так звані *plug-in*), які дуже близькі до *API* операційних систем.

1.6. Висновки до розділу

Однією з тем в програмуванні, до яких інтерес періодично то з'являється, то зникає, є питання метрик коду програмного забезпечення. У великих програмних середовищах час від часу з'являються механізми підрахунку різних метрик. Хвилеподібний інтерес до теми так виглядає тому, що до цих пір в метриках не придумано головного - що з ними робити. Тобто навіть якщо якийсь інструмент дозволяє добре підрахувати деякі метрики, то що з цим робити далі найчастіше незрозуміло. Звичайно, метрики - це і контроль якості коду (не пишемо великі і складні функції), і «продуктивність» (в лапках) програмістів, і швидкість розвитку проекту.

У загальному випадку застосування метрик дозволяє керівникам проектів і підприємств вивчити складність розробленого або навіть розроблюваного проекту, оцінити обсяг робіт, стилістику, що розробляється і зусилля, витрачені кожним розробником для реалізації того чи іншого рішення. Однак метрики

можуть служити лише рекомендаційними характеристиками, ними не можна повністю керуватися, так як при розробці ПЗ програмісти, прагнучи мінімізувати або максимізувати ту чи іншу міру для своєї програми, можуть вдаватися до хитрощів аж до зниження ефективності роботи програми. Крім того, якщо, наприклад, програміст написав мала кількість рядків коду або вніс невелику кількість структурних змін, це зовсім не означає, що він нічого не робив, а може означати, що дефект програми було дуже складно відшукати. Остання проблема, однак, частково може бути вирішена при використанні метрик складності, тому що в більш складною програмою помилку знайти складніше.

РОЗДІЛ 2

МЕТОДИ ОЦІНКИ ЯКОСТІ ПРОГРАМНОГО КОДУ

2.1. Зовнішні чинники якості

Розглянемо зовнішні чинники якості програмного забезпечення (тобто не тільки додатків, але і різних бібліотек чи інші *API*):

- коректність. Програмне забезпечення повинно функціонувати відповідно до технічного завдання, іншими словами, має робити те, що від нього чекають і не містити помилок.

- стійкість. Позаштатні і аварійні ситуації, які можуть виникнути під час експлуатації програмного забезпечення обчислювальних систем, не повинні призводити до плачевних наслідків.

- можливість розширення. «*Software must be soft*» – програмне забезпечення повинно бути гнучким, тобто розвиватися відповідно до потреб користувачів.

- повторне використання. Будь-компонент програмного забезпечення повинен мати можливість повторного використання. Це допомагає уникнути дублювання коду і «розмноження помилок».

- сумісність. Програмне забезпечення повинно коректно працювати в оточенні іншого програмного забезпечення.

Ефективність – це здатність програмного забезпечення якомога менше залежати від ресурсів обладнання [1]. Програма повинна працювати за прийнятний час на якомога ширше коло обчислювальних систем (маються на увазі різні по продуктивності конфігурації, але однієї і тієї ж платформи).

Переносимість. Програмне забезпечення повинно легко переноситися між різними обчислювальними системами.

Простота використання. Освоєння програмного забезпечення не повинно викликати труднощів для кінцевого користувача.

Функціональність. Програмне забезпечення не повинно вміти більше, ніж необхідно, оскільки це робить її громіздкою для кінцевого користувача і

ускладнює подальший розвиток, при цьому програмне забезпечення повинно бути коректним, тобто робити те, що від нього вимагає користувач.

Своєчасність. Програмне забезпечення повинно з'являтися рівно тоді, коли воно необхідне. Якщо вихід затримується, то найімовірніше, в ньому вже не буде сенсу для кінцевого користувача або він зазнає збитків.

Два останніх критерію особливо актуальні для комерційних споживачів, які замовляють розробку програмного забезпечення, хоча вони є актуальними не тільки для них. Програмне забезпечення з недостатньою функціональністю (коректне програмне забезпечення з дуже бідної специфікацією) не зможе конкурувати з більш розвиненими продуктами. Так і програмне забезпечення, створене занадто пізно, дає фору за часом конкурентам, які встигають завоювати ринок і кінцевих користувачів.

При цьому, критерії впливають один на одного і не завжди позитивно. Так надлишкова функціональність впливає, зокрема, і на своєчасність, оскільки розробляти більшу кількість функцій довше і дорожче.

Звісно, що наведений список зовнішніх чинників якості ПЗ не є повним, але він найбільш широко використовується і не суперечить вимогам *ISO 9126*, тобто містить лише найбільш значущі чинники. Можливо запропонувати ряд критеріїв, які можна застосувати виключно до комерційних систем – це, наприклад, вартість, проте ми їх розглядати не будемо.

2.2. Внутрішні чинники якості

Описані вище зовнішні чинники якості програмного забезпечення – це кінцева мета, якої необхідно досягти, щоб розробку і подальше впровадження можна було вважати успішними. Це вимоги до кінцевого результату, описані в технічному завданні.

У довгому і складному процесі отримання програмного забезпечення з технічного завдання є одна проблема: ніхто точно не знає, як вирішити конкретну задачу, щоб гарантувати успіх.

Існують різні технології, що дозволяють досягти мети, але універсальних немає. Хтось із розробників дотримується одних технологій і заперечує інші, у кого-то більш гнучкий підхід. Однак дуже велике число розробників ПЗ у всьому світі використовує саме одні й ті ж ідеї і концепції.

Отже, якщо підтримувати на високому рівні внутрішні показники якості програмного забезпечення, то це може полегшити вихідну задачу, і розробникам буде простіше працювати, тобто легше прагнути до досягнення зовнішнього якості системи.

Можна визначити проектування (або, як зараз кажуть, дизайн) як сукупність властивостей системи, що визначають її внутрішню якість. Внутрішня якість породжує зовнішню.

Одним з найбільш очевидних носіїв внутрішнього якості програмних систем є вихідний код. Він безпосередньо впливає на такі показники внутрішнього якості, як

- читаність;
- легкість модифікації;
- коректність.

Акуратність при створенні вихідного коду захищає від багатьох проблем.

Дуже багато розробників [7] вважають процес моделювання предметної області – проектуванні програмного забезпечення, а написання вихідного коду (тобто реалізації отриманих моделей) (див. Рис. 1) -конструюванням ПЗ.

Розроблені в процесі проектування моделі програмного забезпечення і реалізує їх вихідний код мають багато спільного. По суті, це одне і те ж, тільки написане на різних мовах і ускладнюється від етапу до етапу. Тому вимоги, що пред'являються до вихідного коду і моделей програмного забезпечення, однакові.

З упевненістю можна стверджувати, що функціональність визначає обсяг вихідного коду. Чим більше функціональність, тим більший обсяг вихідного коду буде мати програмне забезпечення. На жаль, ця проблема на сьогоднішній день не має рішення. Тому варто докласти зусиль до того, щоб не ускладнювати код без необхідності.

2.2.1. Оптимізація вихідного коду

Одним з найбільш поширених способів отримання надлишкового ускладнення є оптимізація вихідного коду, що отримується за рахунок підвищення алгоритмічної складності. У більшості випадків (це стосується в першу чергу програмного забезпечення для кінцевих користувачів) набори даних, з якими працює користувач, невеликі (якщо це не база даних, інтерфейс до якої розробляється). Як правило, користувач має справу з даними розміром до сотні елементів.

Винятком є програмне забезпечення, яке працює з свідомо великими наборами даних, таких як СУБД, для моделювання складних фізичних процесів або виведення тривимірної графіки.

2.2.2. Універсальність вихідного коду

Іншим поширеним способом ускладнення вихідного коду є прагнення до універсальності програмного забезпечення. Створюється певний механізм в надії, що він ще стане в нагоді. Часто це ускладнює код. Але набагато більш часто це виправдано.

В даному випадку все набагато складніше, ніж з оптимізацією. Наприклад, прихильники екстремального програмування [9] дотримуються правила: пишіть універсальний механізм тільки тоді, коли він знадобиться. Але вони працюють невеликими командами над невеликими проектами. Існують і інші точки зору.

Якщо завжди розробляти тільки приватні рішення, то доведеться виявляти схожі завдання і переписувати їх, створюючи універсальний механізм. Проблема полягає в тому, що коло таких завдань заздалегідь зазвичай невідомий.

Проте, якщо потрібно універсальний механізм, то необхідно оцінити його складність і, виходячи з цієї оцінки, прийняти рішення: написати його зараз або коли він знадобиться. Тут доведеться скористатися інтуїцією, досвідом та іншими знаннями, які надходили зі зрілістю це неприємно, але неминуче.

Висновок тільки один. Необхідно уникати складності, яка може виявитися надмірною.

2.2.3. Дублювання коду

Важливим завданням при створенні програмного забезпечення є мінімізація дублювання коду.

Копіювання коду з однієї частини програми в іншу веде до небажаного ефекту [10], який полягає в тому, що разом з кодом копіюються всі помилки, допущені в ньому. Через це виявлення та виправлення однієї такої помилки не веде до повного її усунення: помилка залишається в тому місці, куди її скопіювали.

Крім того, повторення однієї і тієї ж функціональності захащує систему однотипними фрагментами коду, що містять, можливо, непомітні, але істотні відмінності. Це ускладнює розуміння пристрою системи, пошук помилок і модифікацію коду.

Найкращий спосіб уникнути дублювання коду – це використовувати стандартну бібліотеку. Багато поширені завдання вже вирішені кваліфікованими людьми, що витратили багато сил на тестування своїх творінь, а стандартні бібліотеки випробувані сотнями і тисячами користувачів.

Володіння стандартними бібліотеками мови, на якому здійснюється розробка ПЗ, обов'язково, так само як і *API* бібліотеками операційних систем. Стандартна бібліотека продумана і протестована краще знову розроблюваного програмного забезпечення. Її використання робить код компактніше і підвищує зрозумілість коду для тих, хто також володіє цією бібліотекою.

Але не всі функції, які необхідні розробнику, реалізовані в стандартній бібліотеці. З іншого боку, якщо така функція знадобилася більш ніж в одному місці програмного забезпечення, то доведеться зробити її «бібліотечною» вже розробнику. Таким чином, в будь-якому великому проекті разом з основною функціональністю розвивається допоміжна бібліотека. Добре, якщо її створюють і тестують вдумливо.

2.2.4. Розкриття поняття «Дірявих» абстракцій

Дуже важливо вміти користуватися стандартною бібліотекою мови програмування. Це істотно підвищує якість вихідного коду, як внутрішнього показника якості. Вміти користуватися – значить уявляти собі, як працює бібліотека, розуміти, як і де краще використовувати той чи інший клас або метод.

Це справедливо для будь-яких механізмів, які використовуються: механізмів мови програмування, операційної системи або чого-небудь ще.

З середовищі розробників програмного забезпечення існує «закон дірявих абстракцій» [11], які загрожують кожному, хто недостатньо добре розуміє механізми роботи своїх інструментів. Закон говорить, що будь-який механізм, який спрощує що-небудь (абстрагується складність якої-небудь дії від програміста), насправді не до кінця приховує свою реалізацію.

Яскравий приклад «дірявої» абстракції – прибирання сміття в *Java*. Логічно припустити, що раз програмне забезпечення розробляється на *Java*, Вам не варто турбуватися про розподіл пам'яті – віртуальна машина все зробить за Вас. Однак деякі проблеми все ж залишаються: абстракція дає текти.

Одна з таких проблем – витік пам'яті (*memory leaks*). В цьому випадку непотрібні об'єкти знищуються, тільки якщо на них немає посилань. Якщо описується, наприклад клас стека, використовуючи масив (не потрібно цього робити, такий клас є в стандартній бібліотеці; приклад взятий з [5]), то є ризик в операції *pop* допустити наступну помилку:

```
public Object pop () {  
    return stack [top--];  
}
```

Помилка в тому, що в тій комірці масиву, з якої «видалили» об'єкт, залишилася посилання на нього. Цей об'єкт не буде видалений, поки в цей осередок масиву не запишуть щось інше або поки сам стік не буде звільнений. Правильніше було б писати так:

```
public Object pop () {  
    Object result = stack [top];
```

```

stack [top] = null; // Обнулити посилання, не утримувати повертається об'єкт
top--;
return result;
}

```

Отже, збирач сміття не врятує, якщо не відомо, як він працює, і не відстежується обнулення посилань – абстракція виявилася дірявим. Така ж проблема може виникнути при недбалому кешуванні проміжних результатів обчислень і в багатьох інших випадках.

Інший приклад: конкатенація рядків (також наводиться в [12]). Є масив рядків, потрібно об'єднати всі рядки з цього масиву в одну. Напишемо так:

```

String result = "";
for (int i = 0; i < array.length; i++) {
    result += array [i];
}

```

В даному випадку творюється дуже багато зайвих об'єктів. Справа в тому, що рядки в *Java* незмінні, тобто не можна приписати один рядок в кінець інший, не створюючи нового об'єкта. Через це на кожному кроці циклу створюється новий об'єкт *String*, в який копіюється вміст обох рядків. Таким чином, трудомісткість цього циклу дуже велика, до того ж створюються непотрібні об'єкти. Абстракція знову дала текти: потрібно думати про те, як це працює.

Для вирішення цього завдання потрібно використовувати спеціальний клас *StringBuffer*, що представляє змінювані рядки.

```

StringBuffer buffer = new StringBuffer ();
for (int i = 0; i < array.length; i++) {
    buffer.append (array [i]);
}

```

Тут не створюється зайвих об'єктів, а додавання нового рядка в буфер займає час, пропорційне довжині цього рядка.

Ще однією дірявої абстракцією є речові числа з плаваючою точкою: тому що неможливо отримати без округлення відповідь 0.1, використовуючи тип *double* в *Java*.

Необхідно враховувати як реалізована та чи інша абстракція.

2.2.5. Локалізація помилок при модифікації коду

Вносячи зміни в код, в нього потенційно вносяться помилки. Від цього важко вберегтися, але можна постаратися зробити так, щоб ці помилки можна було легко виявити і виправити.

Для цього необхідно виконання однієї умови: помилки повинні з'являтися там, де модифікувався код.

Те, що тільки що написали, легко перечитати, проаналізувати, протестувати. Можна перевірити працездатність тих частин системи, які використовують змінений код.

Висновок: необхідно намагатися писати код так, щоб його модифікація зачіпала якомога менше іншого коду.

2.2.6. Мінімізація змін в коді

Кращий засіб для досягнення цієї мети – хороший *API*, тобто добре продумані абстракції. Якщо, змінюючи що-небудь в коді, не зраджуєте *API*, то код клієнтів міняти не доведеться.

Насправді не власне *API*, а інкапсуляція допомагає мінімізувати модифікації клієнтського коду. Якщо заховати реалізацію, то вона не вплине на клієнта.

Максимальна інкапсуляція гарна скрізь, навіть всередині методів. Тому оголошувати локальні змінні бажано рівно там, де вони стають потрібні. У цьому випадку значення змінної, яке зазнало втрат актуальності, не зіпсує життя нікому іншому. Максимально локальне оголошення змінних, як правило, дає наступний ефект:

Змінна не потрібна – ні змінної.

Це, зокрема, «розв'язує руки» збирачеві сміття і зменшує залежність між різними частинами одного і того ж методу.

Звідси випливає ще одне правило:

Якщо поле необхідно тільки для одного методу, воно може виявитися локальною змінною.

Варто замислитися над тим, чи дійсно необхідно поле – можливо, вдасться обійтися локальною змінною. Поле має описувати деяка властивість об'єкта (або щось кешувати, але це інша історія).

Більш низькорівневі засоби мінімізації змін включають в себе такі ідеї, як:

- використання іменованих констант. Якщо константа знадобиться в декількох місцях програми, її потрібно зробити іменованою. Взагалі-то все константи краще робити іменованими – це покращує читаність коду, оскільки ім'я константи зрозуміліше її значення. Іменована константа дає можливість змінити значення в одному місці програми, які не модифікуючи більше нічого. Змінюючи літерали в тисячі місць вручну, Ви можете забути одне-два місця, допустити помилку, а можете і виправити щось не те.

- Використання максимально абстрактних посилань [13]. Всі номери, які використовуються в програмі, повинні мати настільки абстрактний тип, наскільки це можливо. Якщо потрібен список, створіть змінну типу *List* і надайте їй посилання на об'єкт *ArrayList*. Якщо необхідний *LinkedList*, то зміна торкнеться тільки один рядок. Те ж саме стосується параметрів, полів і іншого. Надаємо перевагу інтерфейсів при визначенні типів посилань. Інтерфейс описує абстракцію, яка необхідна розробнику, а не її реалізацію. Це підвищує зрозумілість коду. Якщо немає потреби робити список взагалі, а динамічний масив, використовується посилання типу *ArrayList*.

2.2.7. Небезпечне перевантаження

При зміні коду в декількох місцях, помилки теж можуть виникнути в декількох місцях. Якщо змінився код в одному місці, а помилки все одно виникли в кількох місцях. Це може бути наслідком закону «дірявих» абстракцій (в разі порушення коректності, а не швидкодії, це означає недостатню інкапсуляцію), але може мати й іншу причину, відому як «*Broken dispatch*» [11, 12].

Отже:

- Перевизначенням методу називається зміна реалізації методу в підкласах.
- Перевантаженням методу називається оголошення в одному класі декількох методів з одним ім'ям.

На відміну від перевизначення, перевантаження дозволяється статично: компілятор на стадії складання програми визначає, який метод викликати. Це рішення приймається на основі того, які параметри передаються методу.

Нехай в класі *Thing* був такий метод (так, такий метод спеціально визначати не потрібно, але це тільки приклад):

```
public boolean equals (Object other) {  
    return super.equals (other);  
}
```

Після чого в реалізацію класу додали перевантажений метод:

```
public boolean equals (Thing other) {  
    return this.field == other.field;  
}
```

Необхідно почати з того, що такий метод *equals* порушує угоди платформи *Java* (див. [12]). З того моменту, як з'явився цей метод, стало важко передбачити, який з методів *equals* буде викликаний в тому чи іншому випадку. Якщо посилання на об'єкт *other* має тип *Thing*, то буде викликатися новий метод, а якщо *Object* – старий. Таким чином, клієнтський код був змінений без редагування (деякі виклики стали зв'язуватися з іншим методом).

Гарне правило для перевантаження методів наведено в [12]: не визначати кілька перевантажених методів з однаковим числом параметрів.

Крім описаної проблеми, перевантаження може внести багато плутанини, якщо клієнт переплутає перевантажені версії методів. Часто краще назвати методи по-різному, ніж перевантажувати їх.

2.2.8. Залежності між методами

Якщо кілька методів використовують одні і ті ж глобально доступні змінювані дані (поля), це може загрожувати досить відчутними проблемами,

найбільш грізною з яких є необхідність викликати методи в строго визначеному порядку.

Нехай в класі є такі методи:

- *fill ()*: записує дані у внутрішній масив;
- *sort ()*: сортує внутрішній масив;
- *max ()*: повертає максимальний елемент цього масиву.

Припустимо, що метод *max ()* в реалізації використовує те, що масив відсортований, тобто повертає останній елемент масиву.

Метод *max ()* передбачає, що метод *sort ()* викликався після останнього виклику *fill ()*. Якщо клієнт не викликав *sort ()*, метод *max* буде працювати неправильно.

У цій ситуації виникнення помилки дуже ймовірно, але можливі випадки, коли похибка не буде помічена – якщо максимальний елемент випадково виявиться в кінці масиву.

В данному випадку варто створити методи так, щоб вони не залежали один від одного, тобто щоб будь-який метод можна було викликати в будь-який момент. Однак таке не завжди можливо.

В цьому випадку потрібно або гарантувати, що метод *max ()* викличе *sort ()*, якщо той не був викликаний, або гарантувати, що *sort ()* викликається в кінці методу *fill ()*, або перевіряти в методі *max ()*, що *sort ()* не викликався і кидати виняток. Останній шлях – найменш приємний.

Методи повинні поєднуватись між собою, передаючи один одному параметри. Якщо методи і використовують значення полів, це повинно бути чітко продиктовано тією абстракцією, яку реалізує клас.

Найкращі методи не мають побічних ефектів (в *C ++* вони позначаються модифікатором *const*) – вони не змінюють стану об'єктів і безпечні для свого оточення. Однак обійтися тільки такими методами вдається далеко не завжди.

2.2.9. Недостатня ініціалізація

Всі методи так чи інакше викликаються після конструктора. Це, зокрема, означає, що після створення об'єкта він відразу повинен бути готовий до роботи.

Якщо конструктор НЕ ініціалізує об'єкт повністю, це може привести до помилок. Така небезпека відома як «недостатня ініціалізація» [3].

Найкраще оголосити всі можливі поля як *final* (див. Нижче). Це змусить розробника проініціалізувати їх все в конструкторі.

Щоб виносити ініціалізацію в окремий метод і викликати його в конструкторі можна, тільки якщо цей метод визначено як остаточний (*final*), інакше перевизначення цього методу в підкласах може привести до помилок.

2.2.10. Незмінюваність

Отже, оголошення полів як незмінних (*final*) може бути корисно. До того ж, незмінні методи теж хороші. Звідси можна зробити висновок, що класи, в яких всі методи і поля незмінні, менш схильні до помилок.

У цьому випадку говорять про незмінних об'єктах [12].

Незмінні об'єкти гарні в першу чергу тим, що їх поведінка простіше реалізовувати, простіше дотримуватися коректності. До того ж, їх дані можна відносно безбоязно кешувати.

У стандартній бібліотеці *Java* класи-значення є незмінними (наприклад, *Integer* і *String*). Це дає можливість легко кешувати об'єкти класу *Integer* і повертати один і той же об'єкт багаторазово з методу *Integer.valueOf()*. Тому не варто викликати конструктор класу *Integer* безпосередньо.

Незмінні об'єкти корисні при реалізації перерахувань [12].

Якщо все поля оголошені як *final*, це ще не означає, що об'єкти будуть незмінними. Якщо об'єкт зберігає посилання на змінюваний об'єкт і повертає її клієнтам, то ці клієнти можуть змінити стан змінюваного об'єкта, а як наслідок, зміниться і стан «незмінного».

Таких випадків виникає багато. Особливо часто вони зустрічаються при використанні колекцій і масивів. Тому повертаючи назовні колекцію, варто «загорнути» її за допомогою методу *Collections.unmodifiableCollection()*.

Повертаючи масив, як і будь-який інший змінний об'єкт, його потрібно клонувати.

Інший випадок втрати незмінності – отримання змінюваного об'єкта ззовні. Може зберігатися посилання на змінюваний об'єкт, але доступ нікому не надається. Необхідно визначити, звідки взявся цей об'єкт. Якщо він створений розробником, то немає проблем, але якщо він прийшов від клієнта, то клієнт може зберігати на нього посилання і змінювати об'єкт.

Отримуючи змінювані об'єкти від клієнтів, їх доцільно зберігати не власними ці об'єкти, а їх копії.

2.2.11. Повернення нульовий посилання та єдність дизайну

Найчастіше, якщо метод, який повинен повертати об'єкт, не знаходить цього об'єкта, він повертає *null*. Якщо не перевірити результат методу і звертаються до нього відразу, що може привести до виключення *NullPointerException* [11, 13].

Обробка такого виключення займає досить багато часу і захаращує код. У деяких випадках можна все ж повернути об'єкт, що повністю підтримує інтерфейс відповідної абстракції, але ведучий себе як «відсутність об'єкта».

Наприклад, якщо метод повертає масив або колекцію, краще повернути порожній масив або порожню колекцію. Щоб не створювати багато порожніх об'єктів, можна створити їх один раз і запам'ятати. Порожні колекції вже створені і доступні в класі *Collections*.

Однак не тільки в разі колекції можна створювати «порожні» об'єкти. Наприклад, при реалізації зв'язкових структур замість нульової посилання можна використовувати спеціальні об'єкти, що реалізують кінець списку або лист червоно-чорного дерева. Такі об'єкти не дозволять продовжити ітерацію за межі структури і позбавлять від перевірок на *null*.

Принцип, що забезпечує внутрішню якість, формулюється як єдність дизайну (проектування, конструювання, розробки). Яку б конструкцію не мала програма, необхідно створювати всі її частини однаково.

2.2.12. Якість коду

Внутрішня якість починається з якості вихідного коду програми. Якість коду визначається якістю ідей, викладених в цьому коді. Основним фактором якості вихідного коду програми є його читаність і зрозумілість.

Код може читати будь-який розробник з організації з будь-якою метою:

- з'ясувати, чому зовні все так погано або добре виглядає;
- зрозуміти, що не реалізовано чи, що може стати в нагоді колезі.

Форматування необхідно, щоб підтримувати читаність коду на належному рівні. Правила форматування коду повинні бути єдиними у всьому проекті. А краще – у всій організації і в усьому світі, тому що розробники часто читають вихідний код інших проектів. Згадаймо ще раз про єдність дизайну. Це одне з його проявів. Код необхідно писати відповідно до прийнятих стандартів кодування для *Java*. Багато середовищ розробки (наприклад, *Eclipse* і *IntelliJ IDEA*) можуть автоматично формувати код. Це може допомогти при включенні чужого або старого коду в проект, але новий код необхідно створювати одразу правильно.

Не всі вимоги стандарту кодування очевидні. Наприклад, стандарт регламентує обов'язкове використання фігурних дужок при оформленні структурних операторів, навіть якщо тіло оператора складається з одного рядка. Це необхідно для підвищення читабельності.

2.2.13. Ідіоми мови програмування

Не тільки форматування і конвенції іменування визначають читабельність коду. У будь-якій мові програмування існують так звані ідіоми, тобто застосовуються способи використання тих чи інших конструкцій.

У разі мови *Java* до таких ідіом можна віднести створення обробників подій елементів призначеного для користувача інтерфейсу (*GUI*) в анонімних внутрішніх класах або форму записи циклу *for* для ітерації по колекції:

```
for (Iterator i = collection.iterator (); i.hasNext ();) {  
    Object o = i.next ();
```

}

Використання ідіом дозволяє читачеві пропускати очевидні фрагменти коду і зосередитися на змістовні речі, а також знаходити в коді потрібні фрагменти з характерними конструкціями.

До ідіом відносяться і такі практики, як форма порівняння рядка з літеральної константою:

```
if ( "java" .equals (languageName)) { }
```

Тут ідіома гарантує нам, що порівняння не призведе до виключення *NullPointerException*.

Щоб приступити до програмування на новій мові, необхідно вивчити його синтаксис та ознайомитися з прийнятими в ньому ідіомами.

2.2.14. Документація

Для того, щоб код був більш зрозумілий, варто наблизити його до природної мови. Оскільки ключові слова, як правило, пишуться англійською, найкраще наближатися до англійської мови. Ніколи не користуйтеся транслітом.

Робочий код зазвичай зрозумілий сам по собі, без додавання коментарів. Правильне (осмислене) іменування класів, полів, методів і змінних дає можливість розуміти текст програми без додаткових пояснень. Крім того, розбиття на методи дозволяє виділяти фрагменти коду, що реалізують певні функції, і ім'я методу само по собі є коментарем до такого фрагменту коду.

Однак вчитуватися в код методу зазвичай важче, ніж читати опис, тому класи, якими повинні користуватися інші (а краще – взагалі все класи), повинні бути задокументовані.

Інструмент *JavaDoc* є непоганим засобом для документування коду на *Java*, дозволяючи, з одного боку, створювати документацію, придатну для використання окремо від коду, а з іншого – надаючи всі необхідні пояснення прямо в коді.

Документація повинна містити контракти всіх елементів системи (класів, методів, полів), тобто опис тих умов, при яких ці елементи будуть працювати правильно.

2.2.15. Метафори і зростаючі складні системи

Є можливість використовувати в коді повні назви – вони бувають, наприклад, занадто довгими або взагалі не оформляються інакше, ніж в поширеному реченні.

Для складно описуваних понять має сенс виробити метафори, тобто короткі назви, які виражають суть поняття (можливо, за допомогою аналогії).

Розробники цих практик подбали про те, щоб забезпечити кожен характерний прийом досить наочною метафорою.

Сучасні обчислювальні системи використовують технології, на основі яких знаходяться платформи *.NET: Windows Presentation Foundation, ASP.NET Silverlight* та ін.

Багато факторів внутрішнього якості пов'язані з тими труднощами, які виникають при розвитку програмної системи. Лише невелика кількість програм створюється один раз, передається замовнику і ніколи паче не модифікується. Частина програмних систем розвивається враховуючи зміни вимог, умов експлуатації та кола завдань, що вирішуються системою.

Це створює багато проблем, оскільки спочатку не можна передбачити всіх негараздів, які чекають систему при розробці, як і всіх помилок, що здійснюються при її створенні.

Велику систему досить важко розвивати і підтримувати. Тому необхідно дотримуватися певних правил (як «у великому», так і «в малому»), щоб в подальшому система була досить гнучкою.

Одна з причин існування таких величезних проектів (*bloatware*; наприклад, *Microsoft Word*) пов'язана з тим, що такий зовнішній фактор якості ПЗ, як функціональність, важко визначити однозначно. Тут в силу вступає так званий закон 80/20: «80% користувачів використовують тільки 20% функцій системи».

Чому б не залишити тільки 20% і не викинути інші 80%? Справа в тому, що для кожного користувача ці 20% можуть складатися з різної функціональності [7].

Ця проблема стосується в основному коробкових продуктів, що створюються не для конкретного замовника, а для широкого кола користувачів.

Так чи інакше, більшість програм є неухильно ростуть складні системи [16], які потрібно розширювати, розвивати і випускати вчасно, і в цьому в значній мірі допомагає хороший дизайн.

2.2.16. Метрики

Що ж все-таки таке «хороший дизайн»? Формально кажучи, це такий дизайн, який з гарантією забезпечує досягнення гарного зовнішнього якості системи. Але є одна проблема: швидше за все, єдиного хорошого дизайну немає.

Ніхто досі не запропонував зводу правил, який би гарантував зовнішню якість. Є лише деякі ідеї. Причому мова йде не тільки про дизайн, але і про архітектуру і про організацію розробки всього життєвого циклу ПЗ.

На практиці необхідно хоча б відрізнити поганий дизайн. Для цих цілей існують численні метрики: кількісні характеристики систем, нібито відображають їх внутрішню якість. Ось деякі приклади метрик:

- Середня (або максимальна) довжина методу в рядках. Треба прагнути до зменшення цієї величини.

- Кількість доступних клієнтам елементів класу. Теж варто зменшувати, оскільки з надто великим *API* важко працювати.

Запропоновано досить багато метрик. Деякі з них дійсно корисні як рядовим розробникам, так і людям, які керують розробкою. Існує ряд критеріїв внутрішнього якості, які не беруться кількісно. Це скоріше принципи, ніж метрики.

2.2.17. Модульність

Для того, щоб з великою системою можна було працювати, її доведеться розбити на частини. Це потрібно для того, щоб зрозуміти, як вона працює (добре б зосередитися на одній такій частині, забувши про решту). І, звичайно, без цього не обійтися для того, щоб створити систему: вони не зможуть ні придумати одне

велике рішення для величезної проблеми, ні втілити його, якщо все разом працюватимуть з усією системою разом.

Навіть середня програма не поміщається цілком в голові у однієї людини, тому необхідно розбивати програми на невеликі частини. Будемо називати такі частини модулями.

Розбиття великого завдання на більш дрібні називається декомпозицією задачі. Розумно ділити систему на модулі так, щоб кожен модуль вирішував деяку щодо ізольовану завдання, опис якої можна утримати в голові.

Насправді поняття модуля на різних рівнях абстракції потрібно трактувати по-різному. Справа в тому, що дуже велике завдання стоїть розбити на завдання поменше, ці в свою чергу – на ще більш дрібні, і так далі, поки розмір завдання не стане підвладний одній людині. Кожне розбиття відповідає зниженню рівня абстракції, оскільки при такому «подрібненні» завдання все більш і більш конкретизуються (термін «конкретний» – антонім терміна «абстрактний»). На кожному рівні завдання зіставляється якийсь модуль. Спочатку це підсистеми, потім, можливо, пакети, підпакети і, нарешті, класи. У будь-якому випадку кожен модуль повинен вирішувати своє завдання і тільки її. Потім з реалізацій маленьких завдань необхідно скласти рішення вихідної задачі. Цей процес називається композицією. Композиція – це фактично підйом за рівнями абстракції (збірка більших модулів з більш дрібних). Можна подумати, що процеси декомпозиції і композиції проходять окремо і рознесені в часі. Як правило, це не так. Всі завдання вирішуються поступово: виділяються підзадачі, вони вирішуються, інтегруються в систему, виділяються нові підзадачі.

2.2.18. Повторне використання

У процесі виділення невеликих завдань часто з'ясовується, що деякі з них вже були вирішені: розробником, іншим виробником ПЗ (постачальником бібліотеки) або розробниками стандартної бібліотеки мови. Іноді (особливо це стосується коду, написаного командою) рішення потрібно злегка видозмінити (узагальнити), щоб його можна було використовувати повторно. Повторне

використання економить час на вирішенні тих же завдань, на розумінні стандартних рішень, на пошуку і виправлення помилок.

Навіть якщо в повторно використовуваному коді знайшлася помилка, її потрібно виправити тільки один раз – і все її прояви зникнуть. Для ряду систем повторне використання може бути критерієм зовнішнього якості. Зазвичай це фактор внутрішній. Необхідно проектувати систему так, щоб всі її компоненти могли бути легко використані повторно. Одне з негативних явищ – дублювання коду. І не тільки коду: дублювання ідей, проектних рішень, понять – все це шкодить загальній справі. ООП дуже допомагає при повторному використанні коду: успадкування і поліморфізм – ефективні інструменти для вирішення таких завдань.

2.2.19. Інкапсуляція

Всі модулі мають поводитися так, щоб звести до мінімуму можливість здійснення помилки клієнтами. Це стосується як призначеного для користувача інтерфейсу, так і *API*. Для мінімізації помилок клієнтів необхідно зменшити розмір видимої для клієнта частини *API*. У цьому випадку простіше захистити і забезпечити безпомилкову роботу *API*. Зменшити вихідний код не можна, так як це позначиться на функціональності кінцевого продукту, найефективніше рішення – це заховати сторонніх очей все, що можна заховати. Шкода, але заховати все не вийде: треба ж дати можливість користуватися нашим модулем. Звідси випливає важливий принцип: «Все, що не є необхідним для користувача, має бути приховане». Все те, що користувачеві необхідно, повинно працювати так, щоб він не зміг допустити помилки. Причому, найкраще захищатися навіть від таких помилок, які ще не відомі.

Принцип приховування всіх деталей реалізації називається інкапсуляцією. Його можна сформулювати і так: «Мій модуль робить це. Вам не потрібно знати, як ». Яскравим прикладом є правило приховування полів в класах. У них не повинно бути полів у відкритому доступі (константи в *Java* не можна вважати полями в цьому сенсі). Доступ до полів повинні забезпечувати спеціальні методи (*accessors*). Навіть якщо спочатку ці методи будуть працювати тривіально, у Вас

буде можливість додати в них більш складну поведінку, не зламавши клієнтський код. А ще можна зробити поле доступним тільки для читання, і все це зрозуміють: немає методу на запис – значить, немає. Це ще одна ілюстрація необхідності єдності дизайну: якщо у всій системі доступ до полів надається через методи, всі думатимуть в цих термінах і не будуть намагатися шукати спосіб записувати значення в поле, для якого немає методу запису. Крім того, інкапсуляція покращує якість абстракції: залишаючи тільки необхідне, ми зближуємо уявлення про об'єкт з його інтерфейсом. До того ж, приховану від чужих очей реалізацію можна замінити більш ефективною, не порушивши працездатності клієнтів.

2.2.20. Залежності між модулями

Композиція (як і програмування взагалі) – справа непроста. Одні модулі використовують інші, через що виникають складні залежності. І ось уже програмне забезпечення не можна розглядати як набір модулів, оскільки це моноліт, зібраний з того, що замишлялося як модулі, і обв'язаний навколо залежностями, при цьому жоден модуль не може працювати без інших.

Модифікувати таку програмне забезпечення не просто важко, а неможливо. Для боротьби з цим явищем існує кілька методів. Один з них називається законом Деметри (назва проекту, в якому розвинулася ця техніка), він формулюється так: «*Never talk to strangers*» – «Ніколи не розмовляйте з невідомими». Сенс закону Деметри в тому, що довгі виклики на кшталт *a.getB ()*. *GetC ()*. *GetD ()*. *doE ()* шкодять структурі системи. Проектувати класи потрібно так, щоб виклики були не довше, ніж *a.doB ()*, де *a* – елемент поточного об'єкта. Це радикальне правило. Проектувати системи таким чином складно, хоча можливо. До цього правила слід ставитися як до рекомендації.

Більш дієве правило формулюється як «*Low coupling, high cohesion*» – «Низька зв'язність, високе зачеплення». Кожен модуль повинен бути зв'язковим (логічно єдиним) – елементи всередині модуля повинні бути сильно пов'язані між собою, інакше його варто розділити на кілька модулів. Різні модулі повинні бути пов'язані як можна менше. Ідеальним є випадок, коли зв'язки між модулями

зовсім не утворюють циклів: *B* не залежить від *A* (прямо або побічно, через інші модулі), якщо *A* залежить від *B*. Те ж саме можна виразити в термінах горизонтальних і вертикальних залежностей [16]. Нехай по вертикалі йдуть рівні абстракції, а по горизонталі на кожному рівні – модулі, відповідні цьому рівню. Так ось горизонтальних зв'язків має бути якомога менше (краще, щоб не було зовсім), в той час як вертикальні зв'язку (наслідок композиції) повинні бути чіткими.

Виконання цього правила полегшує композицію і підтримує систему у вигляді стрункого набору модулів.

2.3. Стратегії тестування

Метою тестування є виявлення помилок в програмі.

Тестування програмного забезпечення охоплює цілий ряд видів діяльності, аналогічних послідовності процесів розробки програмного забезпечення. У нього входять:

- постановка задачі для тесту,
- проектування тесту,
- написання тестів,
- тестування тестів,
- виконання тестів,
- вивчення результатів тестування.

Вирішальну роль відіграє проектування тестів. Можливий цілий ряд підходів до стратегії проектування тестів. Щоб орієнтуватися в них, розглянемо два крайніх підходу. Перший полягає в тому, що тести проектуються на основі зовнішніх специфікацій програм і модулів, або специфікацій сполучення програми або модуля. Програма при цьому розглядається як чорний ящик (стратегія «чорного ящика»). Суть такого підходу – перевірити, чи відповідає програма зовнішнім специфікаціям. При цьому логіка модуля зовсім не береться до уваги.

Другий підхід заснований на аналізі логіки програми (стратегія «білого ящика»). Суть підходу – в перевірці кожного шляху, кожної гілки алгоритму. При цьому зовнішня специфікація до уваги не береться. Альтернатива «чорний ящик» – «білий ящик» є досить загальною, що підтверджується її застосуванням не тільки при тестуванні, але, наприклад, в альтернативних напрямках досліджень в області штучного інтелекту (II). У цій області прихильники однієї точки зору («чорний ящик») переконані, що важливо збіг поведінки штучно створених і природних інтелектуальних систем, а внутрішні механізми формування поведінки розробник II зовсім не зобов'язаний копіювати. Цей напрямок II називають машинним інтелектом.

Інша точка зору («білий ящик») – вивчення механізмів природного мислення та аналіз даних про способи формування розумного поведінки людини є основою побудови III. Цей напрямок одержав назву штучного розуму.

Яскравим представником першого напрямку є «*Deer Blue*», яку «навчили» грі в шахи на рівні (а може бути і вище) світових гросмейстерів. Представниками другого напрямку є нейрокомп'ютери.

Жоден з цих підходів не є оптимальним. З аналізу істоти першого підходу ясно, що його реалізація зводиться до перевірки всіх можливих комбінацій значень на вході програми. Розглянемо як приклад завдання тестування тривіальної програми, яка отримує на вході три числа і обчислює їх середнє арифметичне. Тестування цієї програми для всіх значень вхідних даних неможливо, так як їх безліч. Як правило, вичерпне тестування для всіх вхідних даних програми нездійснено, тому обмежуються меншим. При цьому виходять з максимальної віддачі тесту в порівнянні з витратами на його створення. Вона вимірюється ймовірністю того, що тест виявить помилки, якщо вони є в програмі. Витрати вимірюються часом і вартістю підготовки, виконання та перевірки результатів тесту.

Проаналізуємо тепер другий підхід до тестування. На рис. 2.1 зображені можливі шляхи невеликого програмного модуля. Квадратами представлені послідовні сегменти, а стрілками – передачі управління (за допомогою розвилок або циклів). Число шляхів в модулі має порядок (для порівняння, вік всесвіту в

секундах оцінюється як 4). Але навіть якщо припустити, що виконані тести для всіх шляхів, можна стверджувати, що модуль задовільно б не протестували.

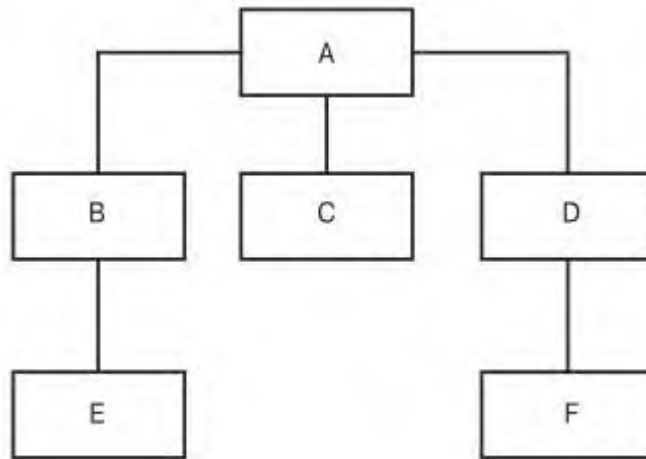


Рис. 2.1. Алгоритмічні шляхи в програмному модулі

Очевидне підставу цього твердження полягає в тому, що виконання всіх шляхів не гарантує відповідності програми її специфікаціям. Припустимо, якщо потрібно написати програму для обчислення кубічного кореня, а програма фактично обчислює корінь квадратний, то програма буде абсолютно неправильною, навіть якщо перевірити всі шляхи. Друга проблема – відсутні шляхи. Якщо програма реалізує специфікації в повному обсязі (наприклад, відсутня така спеціалізована функція, як перевірка на негативне значення вхідних даних програми обчислення квадратного кореня), ніяке тестування існуючих шляхів не виявить такої помилки. І, нарешті, проблема залежності результатів тестування від вхідних даних. Шлях може правильно виконуватися для одних даних і неправильно для інших. наприклад,

$$IF (A + B + C) / 3 = A,$$

то воно буде вірним не для всіх значень A , B і C (помилка виникає в тому випадку, коли з двох значень B або C одне більше, а інше на стільки ж менше A). Якщо концентрувати увагу лише на тестуванні шляхів, немає гарантії, що ця помилка буде виявлена.

Таким чином, повне тестування програми неможливо. Тест для будь-якої програми буде обов'язково неповним, тобто тестування не гарантує відсутність усіх помилок. Стратегія проектування тестів полягає в тому, щоб спробувати зменшити цю неповноту наскільки це можливо. При цьому ключовим питанням є наступний: яке підмножина всіх можливих тестів має найвищу ймовірність виявлення помилок при обмежених часу, трудових витратах, вартості, машинному часу і т.п. Найгіршою з усіх методологій є випадковий набір тестів, так як він має малу ймовірність бути оптимальним.

Рекомендується наступна процедура розробки тестів:

- розробляти тести, використовуючи методи стратегії «чорного ящика»;
- додаткове тестування, використовуючи методи стратегії «білого ящика».

2.4. Методи тестування програмного забезпечення

Існує кілька методів тестування:

1. Тестування програм методом «чорного ящика» (*Black box testing*).
2. Тестування ПЗ методом «білого ящика» (*White box*).
3. Тестування ПЗ методом «сірого ящика» (*Grey box*).

До перевірки не функціональних аспектів ПЗ.

2.4.1 Тестування програми методами «білого ящика» і «чорного ящика»

У термінології професіоналів тестування (програмного і деякого апаратного забезпечення) фрази «тестування білого ящика» і «тестування чорного ящика» ставляться до того, чи має розробник тестів і тестувальник доступ до вихідного коду тестованого ПЗ або ж тестування виконується через інтерфейс або прикладний програмний інтерфейс, наданий тестованим модулем.

При тестуванні «білого ящика» (англ. *White-box testing*, також говорять – прозорого ящика) розробник тесту має доступ до вихідного коду і може писати код, який пов'язаний з бібліотеками тестованого ПЗ. Це типово для юніт-тестування (англ. *Unit testing*), при якому тестуються лише окремі частини

системи. Воно забезпечує те, що компоненти конструкції працездатні і стійкі, до певної міри.

При тестуванні «чорного ящика» (англ. *Black-box testing*) тестувальник має доступ до ПЗ тільки через ті ж інтерфейси, що і замовник або користувач, або через зовнішні інтерфейси, що дозволяють іншого комп'ютера або іншому процесу підключитися до системи для тестування. Наприклад, тестує модуль може віртуально натискати клавіші або кнопки миші в тестованій програмі за допомогою механізму взаємодії процесів, з упевненістю в тому, чи все йде правильно, що ці події викликають той же відгук, що й реальні натискання клавіш і кнопок миші. Як правило, тестування чорного ящика ведеться з використанням специфікацій або інших документів, що описують вимоги до системи.

Якщо альфа- і бета-тестування відносяться до стадій до випуску продукту (а також, неявно, до обсягу тестирующего спільноти і обмеженням на методи тестування), тестування «білого ящика» і «чорного ящика» має відношення до способів, якими тестувальник досягає мети.

Бета-тестування в цілому обмежена технікою «чорного ящика» (хоча постійна частина тестувальників зазвичай продовжує тестування «білого ящика» паралельно бета-тестування). Таким чином, термін «бета-тестування» може вказувати на стан програми (ближче до випуску ніж «альфа»), або може вказувати на деяку групу тестувальників і процес, що виконується цією групою. Отже, тестувальник може продовжувати роботу з тестування «білого ящика», хоча ПЗ вже «в беті» (стадія), але в цьому випадку він не є частиною «бета-тестування» (групи / процесу).

2.4.2. Тестування не функціональних параметрів програми

Існують спеціальні методи для тестування аспектів програм, які не є функціональними, тобто що не відносяться до працездатності самих програм:

1. Тестування продуктивності програмного забезпечення – подивитися працездатність, якщо програма управляє великою кількістю даних або має велике

число користувачів. Це безпосередньо відноситься до поняття масштабованості додатків.

2. Тестування «Юзабіліті» – тестування інтерфейсу користувача, його зручності, практичності і легкості для освоєння звичайним користувачем.

3. Тестування безпеки програм важливо для програм, що мають справу з конфіденційними даними для запобігання використанню вразливостей хакерами.

4. Тестування якості інтернаціоналізації та локалізації програмного забезпечення.

Користуватися цими методами можна і потрібно, щоб програма була якісною.

2.5. Методи стратегії «білого ящика».

Тестування за принципом «білого ящика» характеризується ступенем, який тести виконують або покривають логіку (вихідний текст програми).

2.5.1. Метод покриття операторів

Якщо відмовитися повністю від тестування всіх шляхів, можна показати, що критерієм покриття є виконання кожного оператора програми хоча б один раз. Це необхідна, але недостатня умова для прийняттого тестування за принципом «білого ящика».

У цій програмі можна виконати кожен оператор, записавши один-єдиний тест, який реалізував би шлях *ACE*. Якби на вході було: $A = 2$, $B = 0$, $X = 3$, кожен оператор виконався б один раз. Але цей критерій насправді гірше, ніж він здається на перший погляд. Нехай в першому умови замість "*and*" написаний оператор "*or*" і в другому умови замість $x > 1$ або $x < 1$. В цьому випадку, при тестуванні помилка виявлена не буде. Результати тестування наведені в табл. 1. Зверніть увагу: очікуваний результат визначається за алгоритмом на рис. 2.2, що не відповідає фактичному. Як видно з таблиці, жодна з цих помилок не буде виявлена.

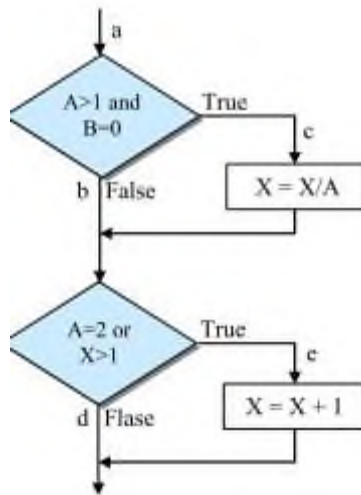


Рис. 2.2. Алгоритм програми, що тестується

2.5.2. Метод покриття рішень (покриття переходів)

Сильніший метод тестування відомий як покриття рішень (покриття переходів). Відповідно до даного методу повинно бути написано достатню кількість тестів, таке, що кожен напрямок переходу має бути реалізовано принаймні один раз. Покриття рішень зазвичай відповідає критерію покриття операторів. Оскільки кожен оператор лежить на деякому шляху, що виходить або з оператора переходу, або з точки входу програми, при виконанні кожного напрямку переходу кожен оператор повинен бути виконаний.

Для програми покриття рішень може бути виконано двома тестами, які покривають шляху $\{ace, abd\}$, або $\{acd, abe\}$. Шляхи $\{acd, abe\}$ крою, вибравши такі вихідні дані: $\{A = 3, B = 0, X = 3\}$ і $\{A = 2, B = 1, X = 1\}$.

2.5.3. Метод покриття умов

Кращі результати в порівнянні з попередніми може дати метод покриття умов. В цьому випадку записується число тестів, достатню для того, щоб всі можливі результати кожної умови в рішенні виконувалися принаймні один раз. У попередньому прикладі маємо чотири умови: $\{A > 1, B = 0\}$, $\{A = 2, X > 1\}$. Отже, потрібна достатня кількість тестів, таке, щоб реалізувати ситуації, де $A > 1$, $A \leq 1$,

$B = 0$ і $B \neq 0$ в точці а і $A = 2$, $A \neq 2$, $X > 1$ і $X \leq 1$ в точці В. Тести, задовольняють критерію покриття умов і відповідні їм шляхи:

- a. $A = 2$, $B = 0$, $X = 4$ *ace*;
- b. $A = 1$, $B = 1$, $X = 0$ *abd*.

2.5.4. Критерій покриття рішень (умов)

Критерій покриття рішень / умов вимагає такого достатнього набору тестів, щоб всі можливі результати кожної умови в рішенні виконувалися принаймні один раз, всі результати кожного рішення виконувалися принаймні один раз і, крім того, кожній точці входу передавалося управління

- a. $A = 2$, $B = 0$, $X = 4$ *ace*;
- b. $A = 1$, $B = 1$, $X = 0$ *abd*.

відповідають і критерієм покриття рішень / умов. Це є наслідком того, що одні умови наведених рішень приховують інші умови в цих рішеннях. Так, якщо умова $A > 1$ буде хибним, транслятор може не перевіряти умови $B = 0$, оскільки при будь-якому результаті умови $B = 0$, результат рішення $((A > 1) \& (B = 0))$ прийме значення брехня. Отже, недоліком критерію покриття рішень / умов є неможливість його застосування для виконання всіх результатів всіх умов.

Інша реалізація даного прикладу приведена на рис. 20. Умови рішення вихідної програми розбиті на окремі рішення і переходи. Найбільш повне покриття тестами в цьому випадку виконується так, щоб виконувалися всі можливі результати кожного простого рішення. Для цього потрібно покрити шлях *HILP* (тест $A = 2$, $B = 0$, $X = 4$), *HIMKT* (тест $A = 3$, $B = 1$, $X = 0$), *HJKT* (тест $A = 0$, $B = 0$, $X = 0$), *HJKR* (тест $A = 0$, $B = 0$, $X = 2$).

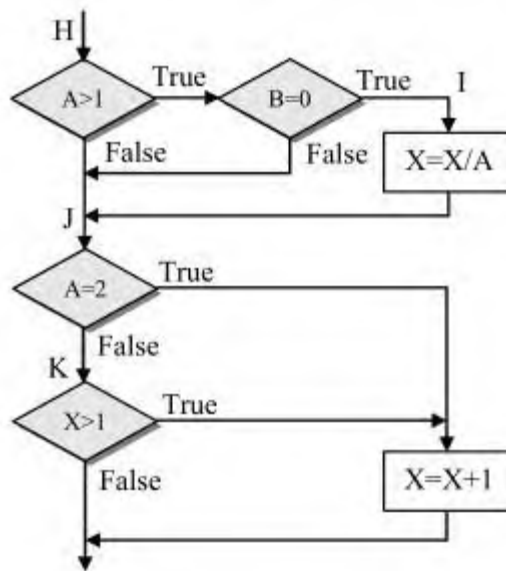


Рис. 2.3. Алгоритм машинного коду

Протестувавши алгоритм на рис. 2.3, неважко переконатися в тому, що критерії покриття умов і критерії покриття рішень / умов недостатньо чутливі до помилок в логічних виразах.

2.5.5. Метод комбінаторного покриття умов

Для підвищення чутливості в логічних виразах використовують інший критерій: комбінаторное покриття умов. Критерій вимагає створення такого числа тестів, щоб всі можливі комбінації результатів умови в кожному рішенні виконувалися принаймні один раз. Набір тестів, які відповідають критерію комбінаторного покриття умов, задовольняє також і критеріями покриття рішень, покриття умов і покриття рішень / умов.

За цим критерієм в даному прикладі повинні бути покриті тестами наступні вісім комбінацій:

1. $A > 1, B = 0$.
2. $A > 1, B \neq 0$.
3. $A \leq 1, B = 0$
4. $A \leq 1, B \neq 0$.
5. $A = 2, X > 1$.
6. $A = 2, X \leq 1$.

$$7. A \neq 2, X > 1.$$

$$8. A \neq 2, X \leq 1.$$

Для того щоб протестувати ці комбінації, необов'язково використовувати всі 8 тестів. Фактично вони можуть бути покриті чотирма тестами:

$$- A = 2, B = 0, X = 4 \text{ \{покриває а, д\};}$$

$$- A = 2, B = 1, X = 1 \text{ \{покриває б, е\};}$$

$$- A = 0,5, B = 0, X = 2 \text{ \{покриває в, ж\};}$$

Те, що чотирма тестами відповідають чотири різних шляхи на рис. 18, 19 є випадковим збігом. Насправді представлені вище тести не покривають всіх шляхів, вони пропускають шлях *acd*. Наприклад, потрібно вісім тестів для тестування наступної програми:

```
if ((x == y) && (z == 0) && end)
```

```
  j = 1;
```

```
  else
```

```
    i = 1;
```

хоча вона задовольняється лише двома шляхами. У разі циклів число тестів для задоволення критерію комбінаторного покриття умов зазвичай більше, ніж число шляхів.

Таким чином, для програм, що містять тільки одну умову на кожне рішення, мінімальним є критерій, набір тестів якого:

1. викликає виконання всіх результатів кожного рішення, принаймні, один раз;
2. передає управління кожній точці входу (наприклад, точки входу, *case*-одиниці) принаймні один раз (щоб забезпечити виконання кожного оператора програми принаймні один раз).

Для програм, що містять рішення, кожне з яких має більше однієї умови, мінімальний критерій складається з набору тестів, що викликають виконання всіх можливих комбінацій результатів умов в кожному рішенні і передавальних управління кожній точці входу програми.

Слово «можливих» вжито тут тому, що деякі комбінації умов можуть бути нереалізованим; наприклад, у виразі $(a > 2) \ \&\& \ (a < 10)$ можуть бути реалізовані тільки три комбінації умов.

2.6. Тестування методом чорного ящика

Тестування методом «чорного ящика» – це проста технологія, яка може надати значний ефект на якість коду. У статті [15] показується навмисне впровадження випадкових неправильні дані в додаток і невірні результати роботи. Також пояснюється використання технологій створення безпечного коду: контрольних сум, сховищ даних *XML* і перевірки коду, які захищають програми від випадкових даних. Короті того, демонструється технологія захисту, а саме вправу по імітації підходів зломщиків коду: пошкоджені файлів, які можуть викликати збій будь-якої програмної системи, наприклад *Microsoft Word*. Кілька невідповідних байтів – і робота програми більш неможлива. Раніше, в епоху операційних систем без механізмів захисту пам'яті, через це могла припинитися робота і всього комп'ютера. Чому *Word* не може розпізнати, що отримані дані не годяться, і просто видати повідомлення про помилку? Чому це додаток пошкоджує свій власний стек та інші області пам'яті тільки через те, що помінялися кілька бітів? *Word* – це не єдина програма, яка поводить себе настільки жахливо при зустрічі з невірно сформованими файлами.

При тестуванні методом «чорного ящика» ви атакуєте програму випадковими невірними даними (які називаються спотворенням (*fuzz*)), а потім чекаєте і дивіться, що зламалося. Хитрість такого методу полягає в тому, що тестування нелогічно. Замість того, щоб намагатися вгадати, які дані можуть спровокувати помилку (як зазвичай надходять люди, що займаються тестуванням), автоматизований тест просто видає програмі як можна більш випадковий «сміття». Помилки, які визначаються подібним тестуванням, звичайно шокують програмістів, оскільки ні один логічно мисляча людина не міг їх навіть очікувати. Проста методика може виявляти в програмах важливі помилки. Вона може знаходити ті види відмов, які виникають при реальній

експлуатації, і сигналізувати про можливі шляхи атак, які необхідно перекрити ще до того, як програмний продукт потрапить до замовників.

Тестування за методом «чорного ящика» реалізувати досить просто:

1. Підготуйте коректний файл, призначений для введення в програму;
2. Замініть деякі частини цього файлу випадковими даними;
3. Відкрийте файл в програмі;
4. Подивіться, що не працює.

Випадкові дані можна змінювати будь-якими способами. Наприклад, можна змінювати не просто частина файлу, а весь файл замінити випадковими даними. Можна обмежити вміст файлу лише *ASCII*-текстом або ненульовими байтами. Головне – це подати з додатком побільше випадкових даних і подивитися, що буде.

Хоча початкові тести можна проводити і вручну, для досягнення максимального ефекту тестування слід автоматизувати. У цьому випадку спочатку потрібно визначити правильну поведінку при помилку, коли додаток отримує невірні вхідні дані. Якщо буде виявлено, що програма взагалі не потурбувалася визначити, що сталося, коли подані вхідні невірні, то це перша помилка. Потім подаються випадкові дані в програму до тих пір, поки не знайдеться файл, у відповідь на який не з'являється правильне діалогове вікно помилки, повідомлення, виняткова ситуація і т.д. необхідно зберегти і зареєструвати цей файл, щоб пізніше можна було відтворити проблему. алгоритм повторюється.

Хоча тестування за методом «чорного ящика» зазвичай вимагає деякого ручного написання коду, існують інструментальні засоби, які можуть допомогти в цьому. Наприклад, Лістинг 1 демонструє простий *Java*-клас, TM який випадковим чином модифікує певну довжину файлу. Зазвичай змінюють файл де-небудь після перших кількох байт, оскільки програми мають тенденцію визначати помилку на самому початку даних, а не потім.

Зламати файл легко. Передати його додатком зазвичай не набагато важче. Для написання цієї частини тесту часто найкращим вибором будуть такі мови сценаріїв, як *AppleScript* або *Perl*. Для програм з графічним інтерфейсом

найважчою частиною може бути розпізнавання того факту, що додаток вказало правильний режим помилки. Іноді найпростіше посадити кого-небудь перед монітором і змусити його позначати кожен тест як вдалий чи невдалий. Обов'язково окремо називайте і зберігайте все згенеровані випадкові контрольні приклади, щоб потім можна було відтворити всі помилки, виявлені за допомогою цієї процедури.

Тестування за методом «чорного ящика» може виявити наявність помилок в програмі. Воно не доводить, що таких помилок в програмі немає. Однак проведення такого тестування значно підвищує впевненість в тому, що додаток надійно і безпечно по відношенню до непередбачених вхідних даних. Якщо тестування програми за цим методом проводилося протягом 24 годин і вона як і раніше працює, то навряд чи подальші атаки подібного роду викличуть помилку. (Зауважте: це не неможливо, а просто менш імовірно.) Якщо тестування все-таки виявило помилки в програмах, їх треба виправити. Замість того, щоб виправляти випадково виявлені помилки у міру їх появи, більш продуктивним може бути фундаментальне виправлення формату файлу на предмет розумного використання контрольних сум, *XML*, очищення пам'яті і / або форматів файлів на базі граматики.

Тестування за методом «чорного ящика» є важливим засобом знаходження в програмах реальних помилок. Всі програмісти, які налаштовані на забезпечення безпеки і надійності своїх програм, повинні користуватися цим засобом.

2.7. Рефакторинг і блокові тести

Основні ідеї будь-якого тестування:

- коли виявляти помилки – чим раніше, тим краще;
- не залишати помилок у себе за спиною;
- задіяти компілятор;

Використовувати краще статичні перевірки, ніж будь-які інші, оскільки вони:

- автоматичні;

- гарантовані (немає випадку, коли програма не дійде до виконання рядка з помилкою);

- помилки виявляються дуже рано.

Модифікації вихідного коду необхідно здійснювати під контролем компілятора, тобто компілятор – головний помічник при модифікації коду. Добре спроектований *API* перекладає базові перевірки на компілятор. В [16] наводяться базові рекомендації при реалізації і модифікації програмних систем:

- поліморфізм краще, ніж *instanceof*;
- компілятор допомагає дізнатися, що ще потрібно змінити;
- дуже часто компілятор покаже нам всі ділянки програми, порушені внесеними змінами;
- додали параметр – міняти треба там, де виникли помилки;
- як видалити (або просто знайти) всі підкласи? Оголосити клас як *final* і подивитися, де видасть помилки компілятор.

Додаткові проблеми виникають при проектуванні і підтримці інтерфейсу програмних систем. Основні етапи життєвого циклу вихідного коду, що реалізує інтерфейс:

- проектування (вигадкування);
- втілення;
- виправлення помилок;
- додавання нових функцій;
- виправлення помилок.

Процес розвитку коду веде до деградації інтерфейсу. Старий інтерфейс не дозволяв зробити щось красиво, тому реалізовувалося некрасиво з різних причин: не було часу робити красиво; хтось не розібрався, як правильно; і зробив неякісно.

2.7.1. Рефакторинг.

Основним способом боротьби з деградацією є рефакторинг, або реорганізація коду – процес зміни внутрішньої структури програми, що не зачіпає її зовнішньої поведінки і має на меті полегшити розуміння її роботи. В

основі рефакторінга лежить послідовність невеликих еквівалентних (тобто зберігають поведінку) перетворень. Оскільки кожне перетворення маленьке, програмісту легше простежити за його правильністю, і в той же час вся послідовність може привести до істотної перебудови програми і поліпшенню її узгодженості і чіткості.

Основні ідеї постійного рефакторінга:

- сумнівний код;
- дублювання коду;
- спадкування;
- виділення методів.

Рефакторинг потрібно застосовувати постійно при розробці коду. Основними стимулами його проведення є наступні завдання:

1. Необхідно додати нову функцію, яка недостатньо вкладається в прийняте архітектурне рішення;
2. Необхідно виправити помилку, причини виникнення якої відразу не ясні;
3. Подолання труднощів у командній розробці, які обумовлені складною логікою програми.

Багато в чому при рефакторінгу краще покладатися на інтуїцію, засновану на досвіді. Проте є деякі видимі проблеми в коді, що вимагають рефакторінга:

1. Довгий метод або великий клас:
 - виділення класів;
 - виділення підкласів.
2. Довгий список параметрів:
 - отримання необхідних значень у параметрів-об'єктів;
 - агрегування групи параметрів в об'єкт.
3. Розходяться модифікації (один клас модифікується з різних приводів):
4. Довгий список параметрів:
 - виділення класів, що відповідають за різні речі.
5. «Стрільба дробом»: щоб зробити щось, потрібно внести багато маленьких змін в різних місцях:

- переміщення членів;

- вбудовування.

6. «Заздрісні функції»: метод користується іншим класом більше, ніж своїм:

- переміщення членів;

7. Групи даних:

- агрегація параметрів в об'єкт;

- отримання даних у одного об'єкта.

8. Одержимість елементарними типами:

- заміна примітивів із зовнішніми операціями на об'єкти.

9. Оператор *switch*:

- заміна логіки поліморфізмом.

10. Паралельні ієрархії успадкування.

11. «Лінійний клас»:

- видалення класу.

12. Теоретична спільність:

- видалення складних загальних механізмів.

13. Тимчасове поле:

- виділення часових полів в окремий клас.

14. Ланцюжки повідомлень, закон Деметри:

- переміщення методу.

15. посередники:

- вбудовування методів;

- видалення класу.

16. Недоречна близькість:

- переміщення членів;

- виділення класу.

17. Альтернативні класи з різними інтерфейсами:

- перейменування методу;

- переміщення методу;

- видалення класу.

18. Неповнота бібліотечного класу:

- локальне розширення.

19. Класи даних:

- при зайвому доступі зовні – видалення встановлює методу;
- переміщення методів в класи даних.

20. Відмова від спадкування: зайва функціональність в базовому класі.

21. Коментарі:

- перейменування;
- пояснює метод або змінна;
- вбудовування змінної;
- введення затвердження;
- розробка через рефакторінг.

Головна ідея – зробити простіше, щоб потім можна було переробити. І це не порожня трата часу – так найчастіше швидше.

Основними недоліками рефакторінга є:

1. Головне: поведінка змінюватися не повинно.

2. проблеми:

- багато часто дуже дрібних змін;
- дрібні зміни можуть призводити до важко що виявляється помилок;
- зміни можуть зачіпати різні частини системи.

При своїх недоліках рефакторінг гарантує коректність в наступному:

1. Твердження:

- їх часто доводиться змінювати;
- до них важко дістатися при виконанні.

2. Автоматичні тести:

- приймальні;
- блокові або модульні.

В цілому, при розробці програмних систем і оцінці якості термін рефакторинг означає зміну вихідного коду програми без зміни його зовнішнього поведінки. У багатьох методологіях рефакторинг є невід'ємною частиною життєвого циклу розробки ПЗ: розробники поперемінно то створюють нові тести

і функціональність, то виконують рефакторинг коду для поліпшення його логічності і прозорості. При рефакторингу автоматичне юніт-тестування перевірить існуючу функціональність і відсутність або наявність змін в ній.

Однак рефакторинг спочатку не призначений для виправлення помилок і додавання нових функцій. Він взагалі не змінює поведінку програмного забезпечення і це допомагає уникнути помилок і полегшити додавання функціональності. Він застосовується для поліпшення зрозумілості коду або зміни його структури, для видалення «мертвого коду» – все це для того, щоб в майбутньому код було легше підтримувати і розвивати. Зокрема, додавання в програму нового поведінки може виявитися складним з існуючою структурою – в цьому випадку розробник може виконати необхідний рефакторинг, а вже потім додати нову функціональність.

Це може бути переміщення поля з одного класу в інший, винесення фрагмента коду з методу і перетворення його в самостійний метод або навіть переміщення коду по ієрархії класів. Кожен окремий крок може здатися елементарним, але сукупний ефект таких малих змін в стані радикально поліпшити проект або навіть запобігти розпаду погано спроектованої програми.

Найбільш вживані [4] методи рефакторинга:

- Зміна сигнатури (*Change Method Signature*)
- 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*)
- Підйом методу (*Pull Up Method*)
- Спуск методу (*Push Down Method*)
- Перейменування методу (*Rename Method*)

- Переміщення методу (*Move Method*)
- Заміна умовного оператора поліморфізмом (*Replace Conditional with Polymorphism*)
- Заміна успадкування делегуванням (*Replace Inheritance with Delegation*)
- Заміна коду типу підкласами (*Replace Type Code with Subclasses*)

2.7.2. Блокові тести

Блочне тестування найбільш зрозуміло для програміста. Фактично це тестування методів якогось класу програми в ізоляції від решти програми і є комбінацією методів «білого» і «чорного» ящиків. Однак, далеко не кожен клас легко покрити блоковими тестами. При проектуванні тестів необхідно враховувати можливість тестованості і залежно класу проектувати і реалізовувати явними.

Щоб гарантувати тестованість, можна застосовувати методологію *TDD*, яка наказує спочатку писати тест, а потім код реалізації тестованого методу. Тоді архітектура виходить, що тестується. Розплутування залежностей можна здійснити за допомогою Інспектора Залежностей (*Dependency Injection, Microsoft*). Тоді кожної залежності явно зіставляється інтерфейс і явно визначається, як інjektується залежність – в конструктор, в властивість або в метод.

Для здійснення блочного тестування існують спеціальні *API* і фреймворки. Наприклад, *NUnit* або тестовий фреймворк з середовища *Visual Studio*. Для можливості тестування класів в ізоляції існують спеціальні «*Mock*» фреймворки. Наприклад «*Rhino Mocks*». Вони дозволяють по інтерфейсів автоматично створювати заглушки для класів-залежностей, задаючи у них необхідну поведінку.

За блоковим тестуванням написано багато статей [15-20]. Нижче наводяться основні ідеї та концепції методологій блочного тестування.

Основні аспекти при проектуванні блокових тестів:

- Вплив тестів на дизайн (*API*), тест – перший клієнт розробляється *API*.

1. Тести як документація.

2. Як часто потрібно запускати тести?

3. Коли додавати тести:

- при розробці контракту класу;
- при виявленні помилки.

4. Як змінювати тести при рефакторінгу?

- це допомагає знайти проблеми, з якими зіткнуться клієнти.

5. Повнота тестування:

- прагнути до повних тестів;
- зосередитися на проблемних режимах;
- перевіряти реакції на порушення контракту (виключення і коди помилок).

6. Що неможливо протестувати:

- затвердження;
- приймальні випробування.

Методологія *JUnit*

– Клас тестів *TestCase*.

– *testMethod ()*.

– Методи *assertTrue ()*, *assertFalse ()*, *assertEquals ()*, *assertNull ()*, *assertNotNull ()*, *assertSame ()*.

– Метод *fail ()*, тестування винятків.

– *setUp ()*, *fixture*.

– *tearDown ()*, *external fixture*.

1. *TestSuite*, *JUnit 4*, *TestNG*, використання анотацій залежності між тестами.

Методологія *TDD (Test Driven Development)*

1. Зв'язок з екстремальним програмуванням.

2. Чистий код, який працює.

3. Спочатку пишуться тести, потім код:

- замовлення *API* від клієнта;
- спочатку подумайте, потім напишіть;
- документація контракту;
- впевненість у змінах.

4. Цикл *TDD*:

- червоний;
- зелений;
- рефакторинг.

5. П'ять кроків:

- написання тесту, компіляція;
- червона смуга;
- модифікація;
- зелена смуга;
- усунення дублювання.

6. Дії на кожному кроці.

Методика *TDD*.

1. Написавши тест, зробити мінімум дій, необхідних для компіляції.

2. Упевнитися: що не повинно працювати, не працює. Якщо працює – розберіться, чому.

3. Мінімальна модифікація. Якщо через вашу модифікації доводиться писати новий тест, значить вона занадто велика:

- підробка реалізації;
- тестування тесту.

4. Необхідно домогтися проходження тесту перед тим, як писати новий код (тест).

5. Усунення дублювання в усьому: в коді, константи, тестах.

6. Вплив *TDD* на дизайн:

- Тест – це специфікація;
- Виявлення проблем і завдань;
- Дублювання де б то не було – привід для рефакторинга.
- зелена смуга;
- усунення дублювання.

Основні патерни *TDD*:

1. ізолювати тести
2. Список тестів – список завдань

3. Спочатку пишеться тест
4. Почати тест з *assert*
5. Зрозумілі тестові дані
6. Коли треба писати новий тест, вибирати зі списку той, який:
 - можна написати;
 - буде корисний для розуміння завдань на даному етапі.
7. яка пояснювала б тест
8. Тест для вивчення бібліотеки
9. Будь-яка стороння думка – привід для додавання рядка в список тестів.
10. Знайшли помилку – пишемо тест, який її відтворює
11. Якщо тест занадто великий, розділіть його на частини

Підроблені об'єкти: якщо потрібно тестувати щось дуже складне і некероване (наприклад, призначений для користувача інтерфейс), можна підробити (імітувати) це своїм тестовим класом.

2.8. Висновки до розділу

У загальному випадку застосування метрик дозволяє керівникам проектів і підприємств вивчити складність розробленого або навіть розроблюваного проекту, оцінити обсяг робіт, стилістику, що розробляється і зусилля, витрачені кожним розробником для реалізації того чи іншого рішення. Однак метрики можуть служити лише рекомендаційними характеристиками, ними не можна повністю керуватися, так як при розробці ПЗ програмісти, прагнучи мінімізувати або максимізувати ту чи іншу міру для своєї програми, можуть вдаватися до хитрощів аж до зниження ефективності роботи програми. Крім того, якщо, наприклад, програміст написав мала кількість рядків коду або вніс невелику кількість структурних змін, це зовсім не означає, що він нічого не робив, а може означати, що дефект програми було дуже складно відшукати. Остання проблема, однак, частково може бути вирішена при використанні метрик складності, тому що в більш складною програмою помилку знайти складніше.

У другому розділі було розглянуто архітектуру для вилучення метрик разом із їх формальним описом за допомогою інструментів *sucj* як *OCL* та метамоделі *UML*. Більше того, бібліотека заходів *FLAME* служить вхідним матеріалом для формалізації метрик. Метою проекту *FLAME* є перетворення розвитку щільних бібліотек лінійної алгебри з мистецтва, зарезервованого для експертів, у науку, зрозумілу як новачкові, так і експерту. Замість того, щоб бути лише бібліотекою, проект охоплює нову нотацію для вираження алгоритмів, методологію систематичного виведення алгоритмів, інтерфейси прикладних програм (*API*) для представлення алгоритмів у коді та інструменти для механічного виведення, реалізації та аналізу алгоритмів та реалізації.

Також у цій главі було введено кілька схем, щоб розпочати проектування системи. Це діаграми використання, діаграми стану та діаграми класів. Вони дають розуміння системи та перші кроки до її реалізації.

РОЗДІЛ 3

РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ОЦІНКИ ЯКОСТІ КОДУ

3.1. Розробка метричних лічильників

3.1.1. Використання *XML* під час реалізації метрів метрик.

Розширювана мова розмітки (*XML*) – це мова розмітки, яка визначає набір правил для кодування документів у форматі, який читається як людиною, так і машинно. Це визначено в специфікації *XML* 1.0, виробленій W3C, та декількох інших відповідних специфікаціях, усі безкоштовні відкриті стандарти.

Цілі дизайну *XML* підкреслюють простоту, загальність та зручність використання через Інтернет. Це формат текстових даних із потужною підтримкою мов світу через *Unicode*. Хоча дизайн *XML* зосереджений на документах, він широко використовується для представлення довільних структур даних, наприклад у веб-сервісах.

Багато інтерфейсів прикладного програмування (*API*) розроблено для розробників програмного забезпечення для обробки даних *XML*, і існує кілька системних схем, які допомагають у визначенні мов на основі *XML*.

Розроблено сотні мов на основі *XML*, включаючи *RSS*, *Atom*, *SOAP* та *XHTML*. Формати на основі *XML* стали типовими для багатьох інструментів для підвищення офісної продуктивності, включаючи *Microsoft Office* (*Office Open XML*), *OpenOffice.org* (*OpenDocument*) та *iWork* від *Apple*. *XML* також використовується як основна мова для протоколів зв'язку, таких як *XMPP*.

При реалізації дипломного завдання було вирішено використовувати *XML*-файли для зберігання метрик. Причини полягали в тому, що це забезпечить легку розширюваність системи разом із простотою запитань щодо зберігання. Потім код, що зберігається у файлах *XML*, розпізнається як текст і записується у файл

txt. У свою чергу, *txt*-файл динамічно компілюється і остаточно завантажується в програму як клас, відповідний для обчислення метрик. Це найпростіший метричний метр *LOC* (рядки коду), реалізований мовою *JAVA*:

```
public class metric1
{
    public int calculateLOC(String[] str,int i)
    {

        for (int j=0;j<i;j++)

            {

                if (!str[j].isEmpty()) count++;

            }

        return count;

    }

    int count=0;
}
```

Метричний параметр представляє клас, що називається *metric1*, який містить одну функцію *public int izračunatiLOC (String [] str, int i)*, який приймає за параметри масив рядків і лічильник його членів, що повертає кількість непорожніх рядків у даному файлі.

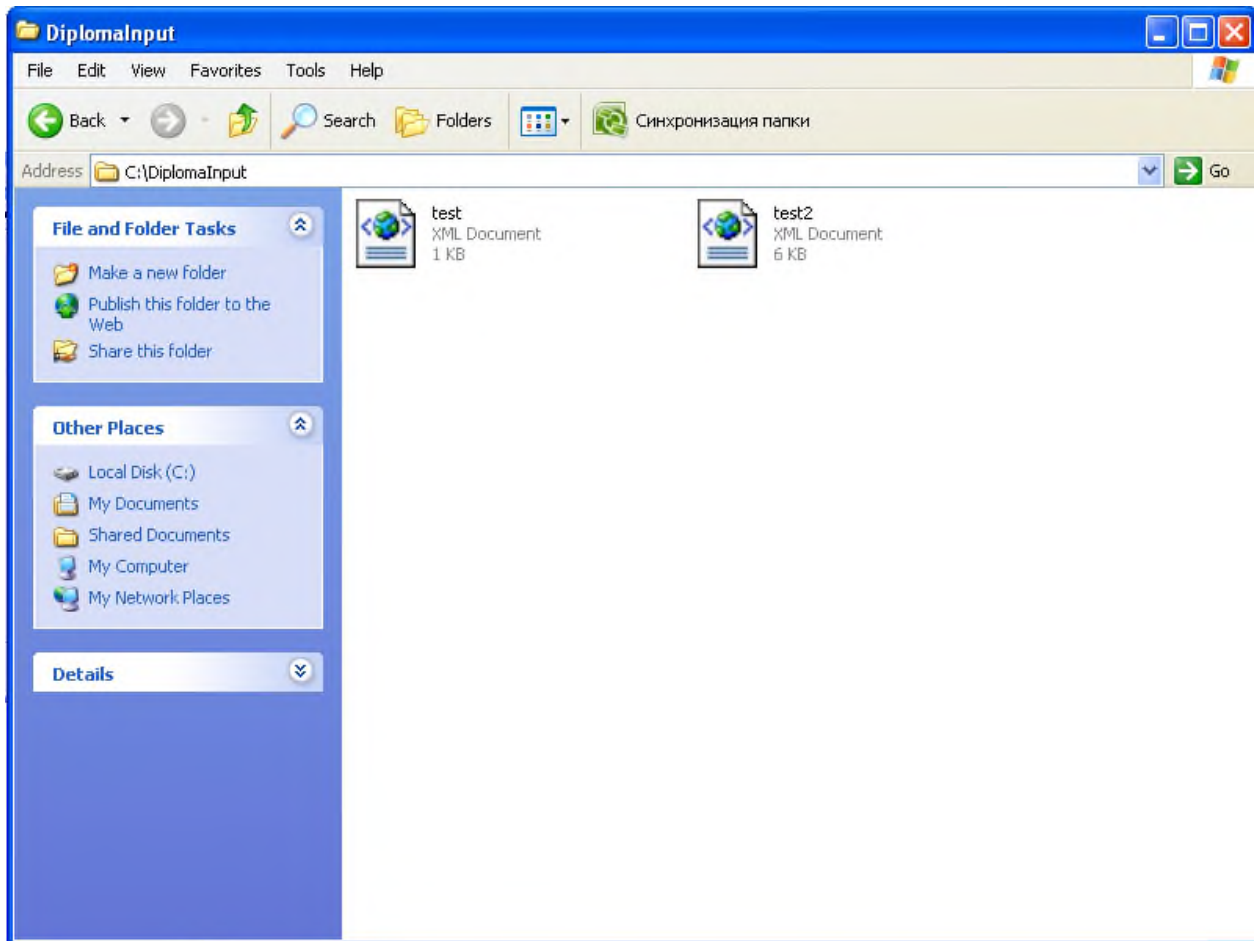


Рис. 3.1. Вхідна папка з файлами *XML*, де зберігаються метрики

3.2. Розробка системи автоматизованих вимірювань метрик

3.2.1. Завантаження файлу

Для того, щоб створити таку систему, яка була б легко розширюваною та зручною у використанні, потрібно було продумати принципи взаємозв'язку її модулів. Перша проблема полягала в забезпеченні ефективного завантаження файлів у додаток та подальших маніпуляцій з ним.

Слід зазначити, що програма, що розглядається, стосується двох типів файлів: вихідних файлів *JAVA* (*.java*) та вихідних файлів *C #* (*.cs*). Як видно на рисунку 3.2. є два прапорці для вибору типу файлів, які ви хочете завантажити в програму.

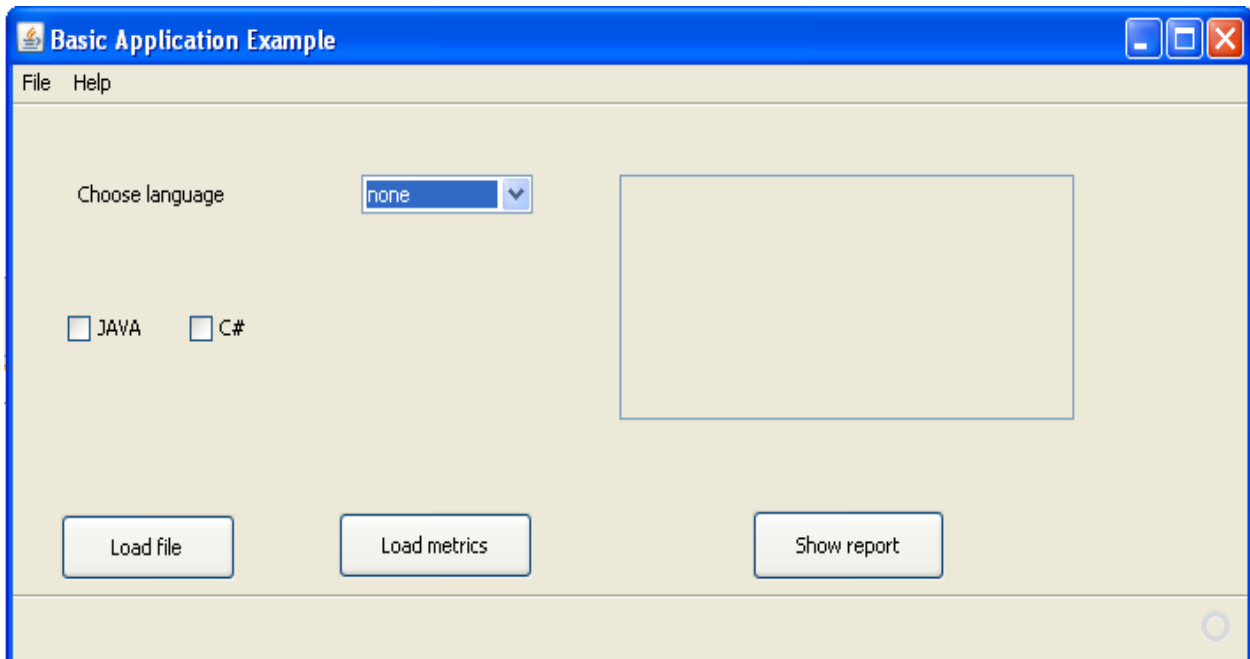


Рис. 3.2. Основний фрейм реалізованої програми

Завантаження виконується за допомогою класу *JfileChooser JAVA*, який надає просте у використанні діалогове вікно, яке пропонує вибрати файл для завантаження. Ось частина інтерфейсу, створена для завантаження файлів.

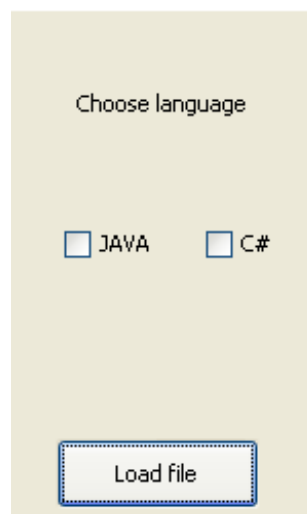


Рис. 3.3. Частина, відповідальна за завантаження файлів *JAVA* або *C #*.

JFileChooser – це діалогове вікно для вибору файлу або файлів. Повернене значення трьох методів є одним із наступних:

- *JFileChooser.CANCEL_OPTION*, якщо користувач натискає кнопку Скасувати.

- *JFileChooser.APPROVE_OPTION*, якщо користувач натискає кнопку ОК / Відкрити / Зберегти.

- *JFileChooser.ERROR_OPTION*, якщо користувач закриває діалогове вікно.

Повернене значення *JFileChooser.APPROVE_OPTION* вказує на те, що ви можете викликати його *getSelectedFile* або *getSelectedFiles* методи:

- *public java.io.File getSelectedFile ()*
- *public java.io.File [] getSelectedFiles ()*

JFileChooser має допоміжні класи: клас *FileFilter*, клас *FileSystemView*, *FileView*.

Клас *FileFilter* призначений для обмеження файлів і каталогів, які будуть перераховані в *FileView JFileChooser*. *FileView* контролює, як каталоги та файли перелічуються в *JFileChooser*. *FileSystemView* – це абстрактний клас, який намагається приховати особливості операційної системи, пов'язані з файловою системою, від засобу вибору файлів. Ось частина коду, що описує, як працює *JFileChooser*:

```
chooser1 = new JFileChooser();
FileNameExtensionFilter filter = new FileNameExtensionFilter("JAVA
files","java");
//Filter initializing
FileNameExtensionFilter filter2 = new FileNameExtensionFilter("C# files",
"cs");
chooser1.setFileFilter(filter);
chooser1.setCurrentDirectory(new File("C://")); //sets current directory
int result =
chooser1.showOpenDialog(Metrics2App.getApplication().getMainFrame());
if (result == JFileChooser.APPROVE_OPTION)
{
    name = chooser1.getSelectedFile().getName();
    //gets name of selected file
}
```

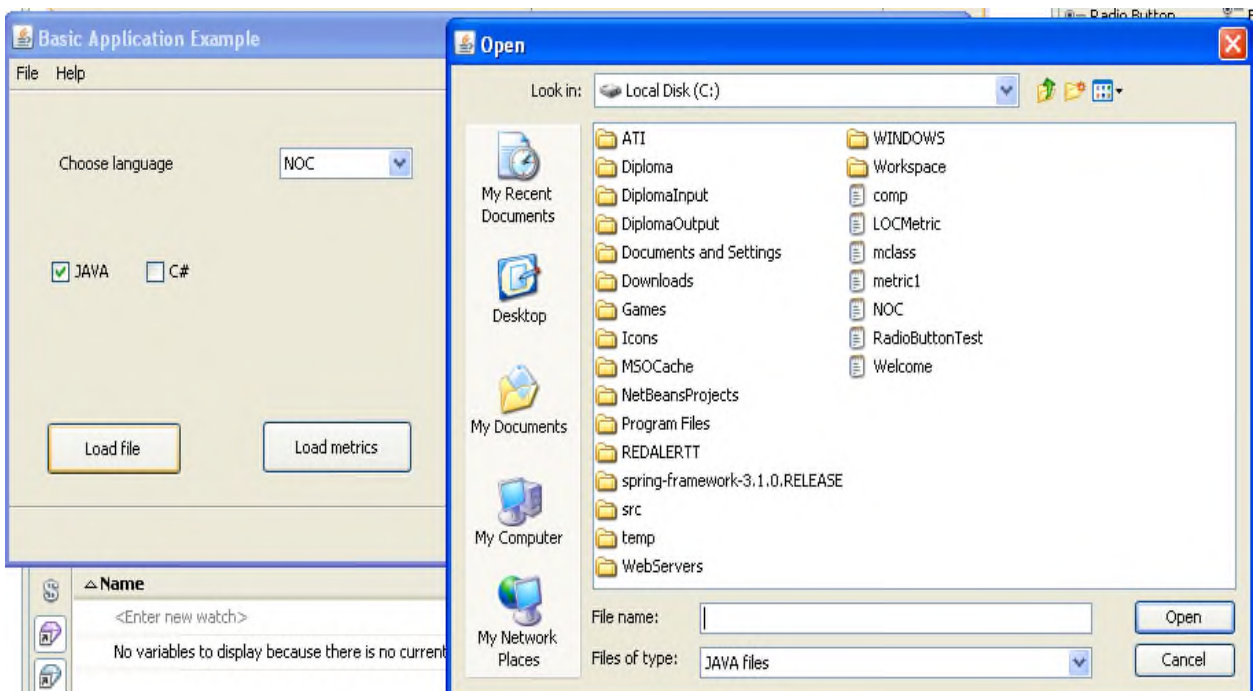


Рис. 3.4. Вибір файлів *JAVA*.

І ось що станеться, якщо користувач встановить прапорець *C #* у головному фреймі програми (рис. 3.5).

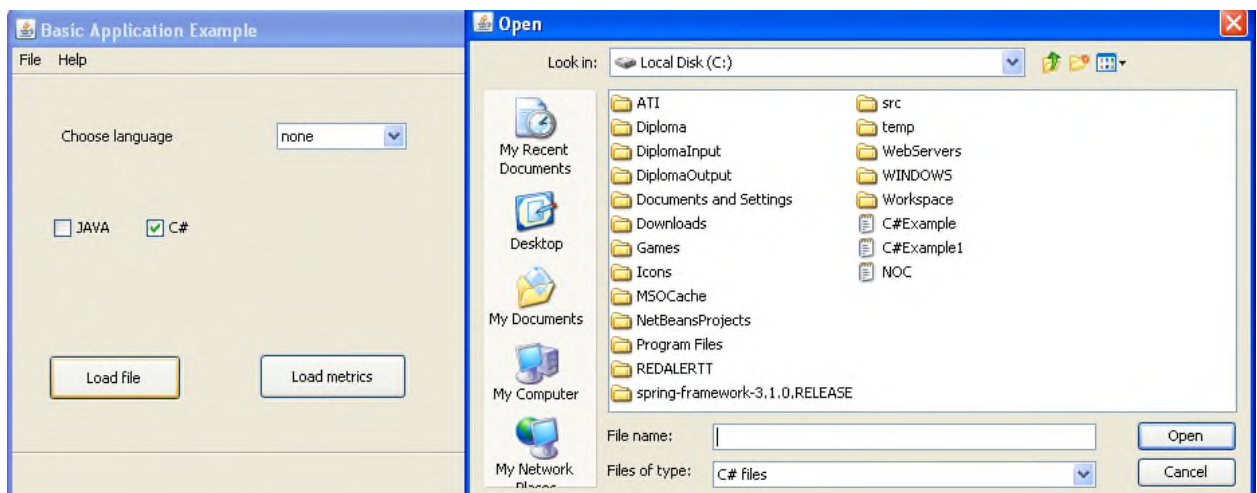


Рис. 3.5. Вибір файлів *C #*.

Після завантаження файлу необхідно відсканувати його рядки для подальших маніпуляцій. Для цього використовується спеціальний клас Сканер, тобто простий сканер тексту, який може аналізувати примітивні типи та рядки за допомогою регулярних виразів.

Сканер розбиває свої дані на маркери, використовуючи шаблон роздільника, який за замовчуванням відповідає пробілу. Потім отримані маркери можуть бути перетворені у значення різних типів за допомогою різних наступних методів. Методи *next ()* і *hasNext ()* та їх супутні методи примітивного типу (наприклад, *nextInt ()* та *hasNextInt ()*) спочатку пропускають будь-який ввід, що відповідає шаблону роздільника, а потім намагаються повернути наступний маркер. Обидва методи *hasNext* і *next* можуть блокувати очікування подальшого введення. Чи блоки методу *hasNext* не мають зв'язку з блокуванням пов'язаного ним наступного методу чи ні.

Методи *findInLine (java.lang.String)*, *findWithinHorizon (java.lang.String, int)* та пропуск (*java.util.regex.Pattern*) працюють незалежно від шаблону роздільника. Ці методи намагатимуться збігатись із зазначеним шаблоном, не враховуючи роздільників у вхідних даних, і, отже, можуть бути використані в особливих обставинах, коли роздільники не мають значення. Ці методи можуть блокувати очікування додаткового введення.

Коли сканер видає *InputMismatchException*, сканер не передає маркер, що спричинив виняток, щоб його можна було отримати або пропустити за допомогою іншого методу.

Залежно від типу шаблону розмежування можуть бути повернуті порожні маркери. Наприклад, шаблон "*\\ s +*" не поверне жодного порожнього маркера, оскільки він відповідає кільком екземплярам роздільника. Шаблон розмежування "*\\ s*" може повертати порожні маркери, оскільки він пропускає лише один пробіл за раз.

Сканер може зчитувати текст з будь-якого об'єкта, що реалізує зручний для читання інтерфейс. Якщо виклик базового методу *Readable.read (java.nio.CharBuffer)*, що читається, викидає *IOException*, тоді сканер припускає, що кінець вводу досягнутий. Найновіший *IOException*, викинутий базовим читабельним способом, можна отримати за допомогою методу *IOException ()*. Ось деякий код, що представляє використання класу *Scanner*.

```
Scanner in = new Scanner(chooser1.getSelectedFile(), "UTF-8");  
if (i!=0) i=0;
```

```

while(in.hasNextLine())
{
    str[i]=in.nextLine();
    i++;
    count1=i;
}

```

Отже, використання класу *Scanner* виявляється дуже корисним для запису рядкових даних у масив, який згодом передається методам метричних вимірювачів.

3.2.2.Завантаження метрики.

Після маніпуляцій з файлом дуже важливо успішно завантажити метрики метрики в основну програму, щоб виконати відповідні обчислення з файлом.

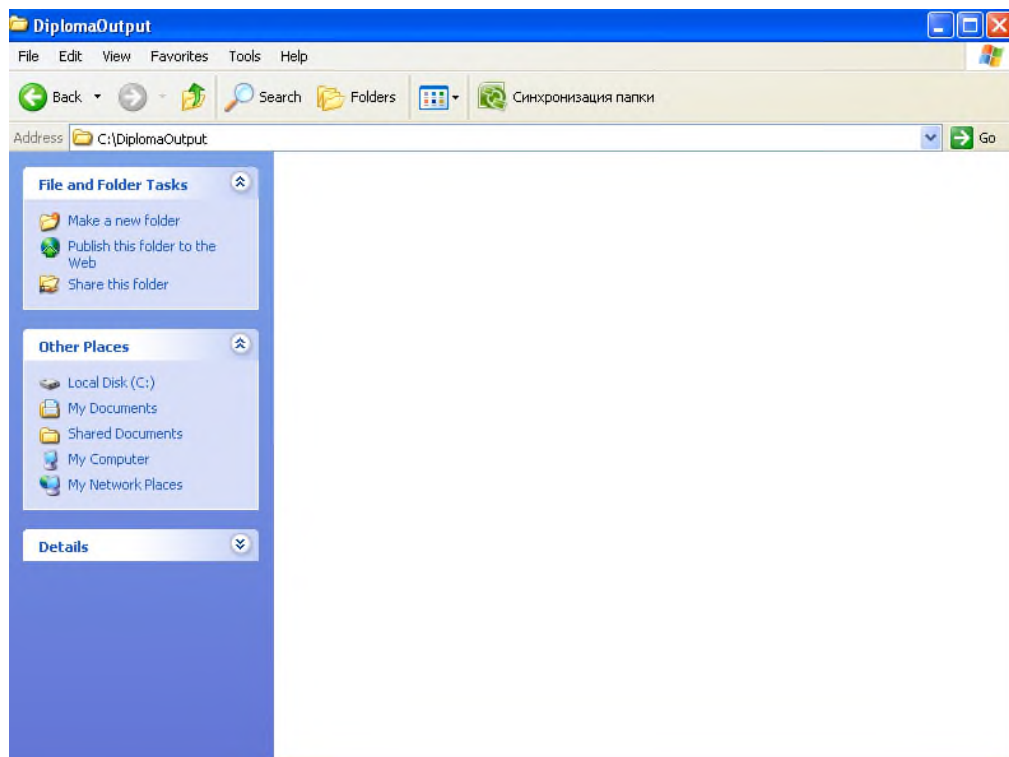


Рис. 3.6.Вихідна папка перед завантаженням метрик

Для цього був розроблений спеціальний клас, який називається *GetMetric class*, що містить три методи, перший із яких (*writetotxtfile*) слід розглянути зараз.

```

    public void writetotxtfile(String readxmlfrom, String writetonafile) throws
    ParserConfigurationException, SAXException, IOException
    {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        File f = new File(readxmlfrom);
        Document doc = builder.parse(f);
        Element root = doc.getDocumentElement();
        String text = root.getTextContent();
        PrintWriter out = new PrintWriter(new
        FileWriter(writetonafile));
        out.print(text);
        out.close();
    }

```

Як можна бачити, головна роль у виконанні цих завдань для двох класів: *DocumentBuilder* та *PrintWriter*:

```

public abstract class DocumentBuilder
extends Object

```

Визначає *API* для отримання екземплярів документа *DOM* з документа *XML*. Використовуючи цей клас, програміст програми може отримати файлДокумент з *XML*.

Екземпляр цього класу можна отримати в *DocumentBuilderFactory.newDocumentBuilder* ()метод. Отримавши екземпляр цього класу, *XML* можна проаналізувати з різних джерел вхідних даних. Такими джерелами вхідних даних є *InputStreams*, Файли, *URL*-адреси та *SAX InputSources*.

Зверніть увагу, що цей клас повторно використовує кілька класів з *API SAX*. Це не вимагає, щоб реалізатор базової реалізації *DOM* використовував синтаксичний аналізатор *SAX* для синтаксичного аналізу *XML*-документа в

документ. Він просто вимагає, щоб реалізація спілкувалась із додатком за допомогою цих існуючих *API*.

Те, що було розроблено, реалізовано за допомогою так званого синтаксичного аналізатора *DOM*. Інтерфейс *DOM* – це найпростіший *XML*-аналізатор для розуміння та використання. Він аналізує весь *XML*-документ і завантажує його в пам'ять, моделюючи його за допомогою *Object* для легкого обходу або маніпуляцій.

Хоча синтаксичний аналізатор *DOM* працює повільно і споживає багато пам'яті, якщо завантажує *XML*-документ, що містить багато даних, для даного завдання це правильний вибір:

```
public class PrintWriter  
extends Writer
```

Друкує відформатовані подання об'єктів до потоку текстового виводу. Цей клас реалізує всі методи друку, знайдені в *PrintStream*. Він не містить методів для написання необроблених байтів, для яких програма повинна використовувати некодовані потоки байтів.

На відміну від класу *PrintStream*, якщо ввімкнено автоматичну змивку, це буде зроблено лише тоді, коли буде викликаний один із методів *println*, *printf* або *format*, а не всякий раз, коли випадково виводиться символ нового рядка. Ці методи використовують власне поняття платформи про роздільник рядків, а не символ нового рядка.

Методи цього класу ніколи не створюють винятків вводу-виводу, хоча деякі з його конструкторів можуть. Клієнт може запитати, чи не сталися помилки, викликавши *checkError* ().

Друга функція в цьому класі:

```
public void compiletxtfile(String s) throws IOException  
  
{
```

```
    JavaCompiler jc = ToolProvider.getSystemJavaCompiler();
```

```
jc.getStandardFileManager(null, null, null);  
    File javaFile = new File(s);  
    // getJavaFileObjects' param is a vararg  
    Iterable fileObjects = sm.getJavaFileObjects(javaFile);  
    jc.getTask(null, sm, null, null, null, fileObjects).call();  
    // Add more compilation tasks  
    sm.close();  
}
```

Тут всю увагу слід звернути на інтерфейси *StandardJavaFileManager* та *JavaCompiler*. За допомогою інтерфейсу *JavaCompiler* програмаможливість динамічної компіляції коду *JAVA* може бути реалізована:

```
public interface JavaCompiler  
    extends Tool, OptionChecker
```

Інтерфейс для виклику компіляторів мови програмування *Java* із програм.

Компілятор може генерувати діагностику під час компіляції (наприклад, повідомлення про помилки). Якщо надається діагностичний прослуховувач, він буде переданий слухачеві. Якщо прослуховувач не надається, діагностика буде відформатована у невизначеному форматі та записана у вихідний файл за замовчуванням, тобто *System.err*, якщо не вказано інше. Навіть якщо постачається прослуховувач діагностики, деякі засоби діагностики можуть не поміститися в діагностиці і будуть записані на вихідні дані за замовчуванням.

Інструмент компілятора має відповідний стандартний файловий менеджер, який є диспетчером файлів, який є природним для інструмента (або вбудованого). Стандартний файловий менеджер можна отримати, зателефонувавши *getStandardFileManager*.

Інструмент компілятора повинен функціонувати з будь-яким файловим менеджером доти, доки виконуються будь-які додаткові вимоги, як описано в

наведених нижче методах. Якщо менеджер файлів не передбачений, інструмент компілятора використовуватиме стандартний менеджер файлів, такий як той, який повертає *getStandardFileManager*.

Компілятор покладається на дві служби: діагностичний слухач і файловий менеджер. Хоча більшість класів та інтерфейсів у цьому пакеті визначає *API* для компіляторів (та інструментів загалом) інтерфейсів *DiagnosticListener*, *JavaFileManager*, *FileObject*, і *JavaFileObject* не призначені для використання в додатках. Натомість ці інтерфейси призначені для реалізації та використання для надання індивідуальних послуг для компілятора і, таким чином, визначають *SPI* для компіляторів:

```
public interface StandardJavaFileManager  
extends JavaFileManager
```

Кожен компілятор, який реалізує цей інтерфейс, забезпечує стандартний менеджер файлів для регулярної роботи файлів. Інтерфейс *StandardJavaFileManager* визначає додаткові методи створення файлових об'єктів із звичайних файлів.

Стандартний файловий менеджер має дві цілі:

- основний будівельний блок для налаштування того, як компілятор читає та пише файли,
- спільне використання між кількома завданнями компіляції.

Повторне використання файлового менеджера може потенційно зменшити накладні витрати на сканування файлової системи та читання файлів *jar*. Отже, після використання цього ми отримуємо наступне:

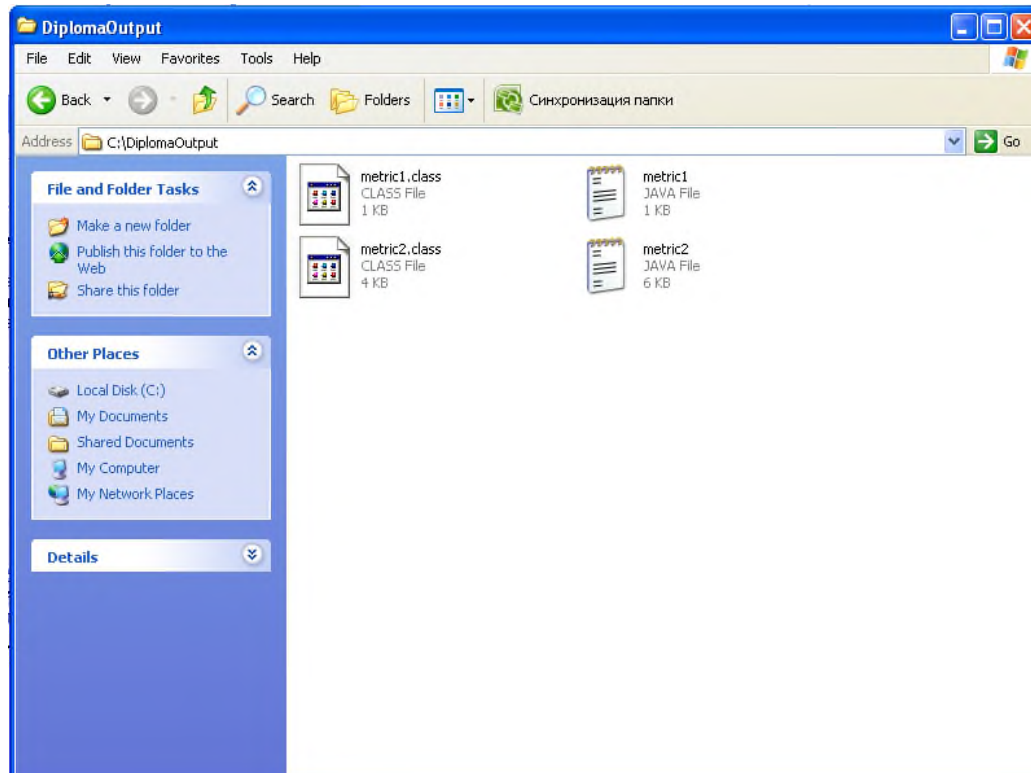


Рис. 3.7. Вихідна папка після завантаження метрик

Третя функція має справу з кодом, що зберігається у файлі *xml*:

```
public Class getMethodFromMetric(String outputdirectory, String classname)
throws MalformedURLException, ClassNotFoundException
```

```
{
```

```
    File outputDir = new File(outputdirectory);
```

```
    URL[] urls = new URL[]{outputDir.toURI().toURL()};
```

```
    //from directory to URL
```

```
    URLClassLoader ucl = new URLClassLoader(urls, null);
```

```
    Class clazz = ucl.loadClass(classname);
```

```
    return clazz; }
```

Ця функція повертає клас, що зберігається у файлі *xml*. Тут слід враховувати класи *URLClassLoader* та *Class*.

```
public class URLClassLoader
```

```
extends SecureClassLoader
```

```
implements Closeable
```

Цей завантажувач класів використовується для завантаження класів та ресурсів із шляху пошуку *URL*-адрес, що посилаються як на файли *JAR*, так і на каталоги. Будь-яка *URL*-адреса, яка закінчується символом */*, передбачає посилання на каталог. В іншому випадку передбачається, що *URL*-адреса посилається на файл *JAR*, який буде відкрито за потреби.

AccessControlContext потоку, який створив екземпляр *URLClassLoader*, буде використовуватися при подальшому завантаженні класів та ресурсів.

Завантажені класи за замовчуванням отримують дозвіл лише на доступ до *URL*-адрес, вказаних під час створення *URLClassLoader*:

```
public final class Class<T> extends Object
implements Serializable, GenericDeclaration, Type
```

Екземпляри класу *Class* представляють класи та інтерфейси в запущеному додатку *Java*. Перелік – це свого роду клас, а анотація – це своєрідний інтерфейс. Кожен масив також належить до класу, який відображається як об'єкт Класу, який спільно використовується усіма масивами з однаковим типом елемента та кількістю вимірів. Примітивні типи *Java* (*boolean*, *byte*, *char*, *short*, *int*, *long*, *float* та *double*) та ключове слово *void* також представлені як об'єкти класу. Клас не має публічного конструктора. Натомість об'єкти класу автоматично будуються віртуальною машиною *Java* під час завантаження класів та викликами методу *defineClass* у завантажувачі класів.

Отже, після виклику цієї функції легко отримати доступ до методів, що містяться в ній, викликавши функцію *metric2.newInstance* (), яка має справу зі спеціальним механізмом, який називається відображенням, тобто динамічним створенням екземпляра класу.

Кожна метрика має свої унікальні характеристики, і тепер вона буде показана на прикладі двох простих метрик: *LOC* (рядки коду) – загальна метрика і *NOC* (кількість дочірніх класів) – суто об'єктно-орієнтована метрика. надати деяку інформацію про метрики Використовується об'єкт *JComboBox*. Наприклад, якщо вибрати метрику *LOC*, не лише інформація про неї відображається, але й одиниці виміру додаються до інтерфейсу (рис. 3.8).

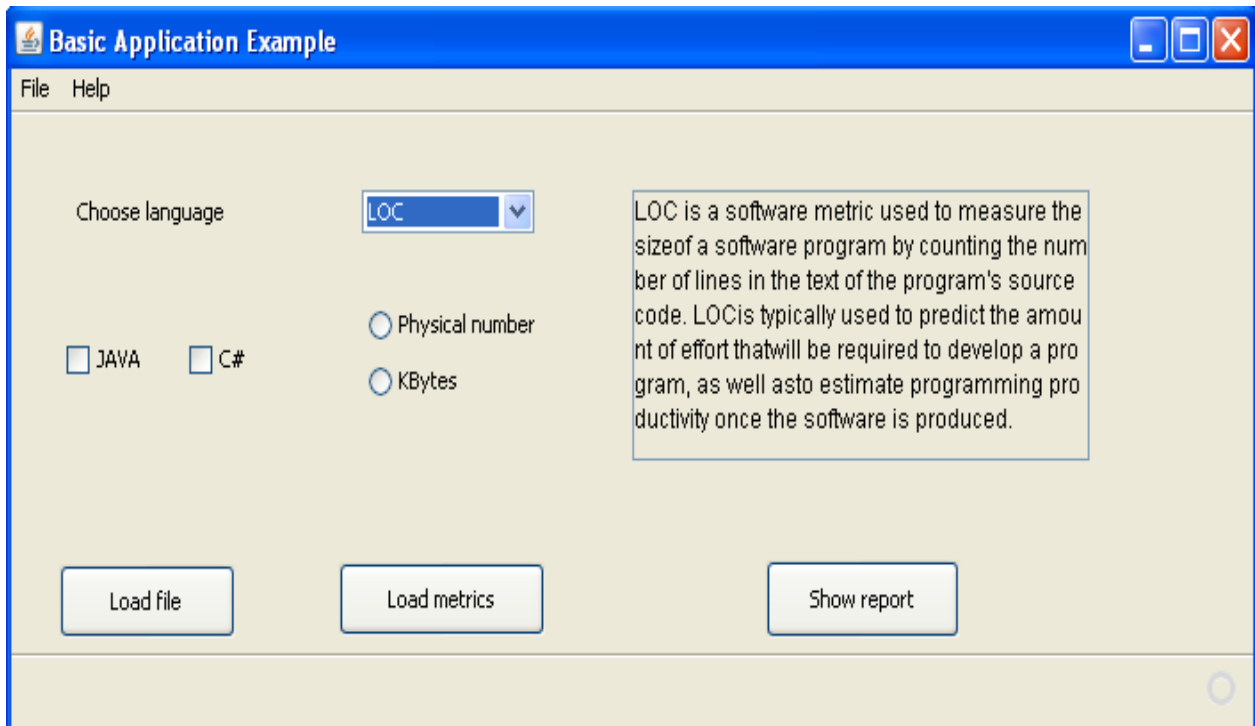


Рис. 3.8. Інформація, показана для метрики *LOC*

При виборі метричного інтерфейсу *NOC* зміни змінюються, оскільки *NOC* забезпечується лише одним типом одиниць виміру. Це впливає на звіт, про що піде мова далі.

Ось ще одна фігура, що показує, як інтерфейс змінюється відповідно до вибору метрик.

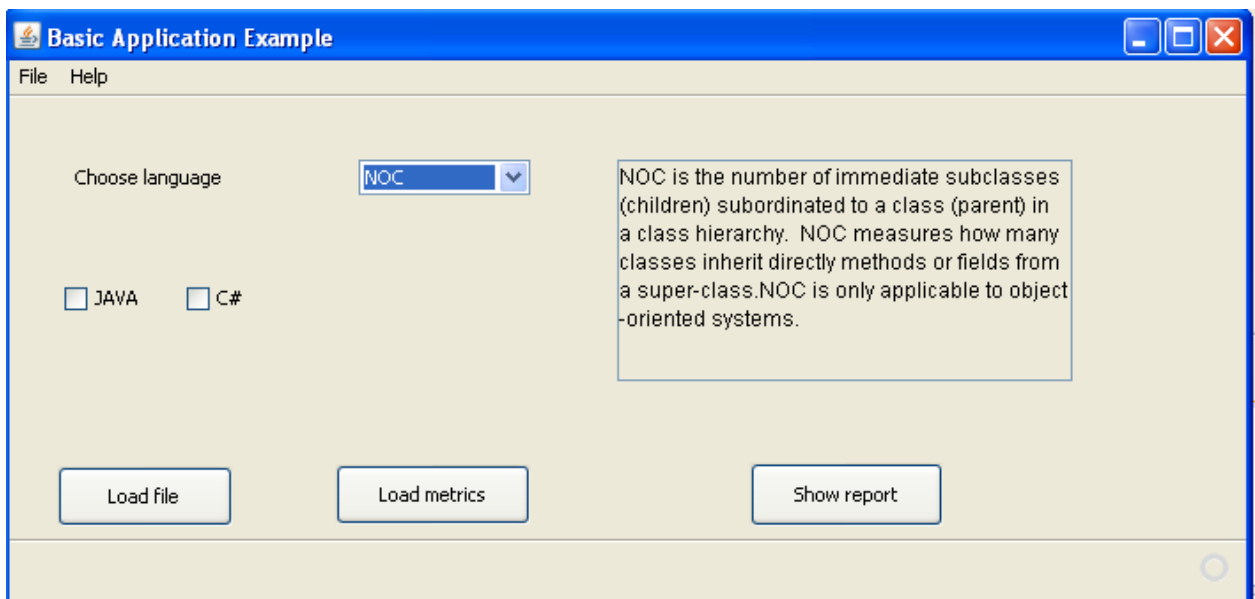


Рис. 3.9. Інформація, показана для метрики *NOC*

3.2.3. Створення звіту

Врешті-решт, як файл, так і вибраний метричний метр завантажуються в програму, і залишається лише представити результат у відповідній формі. Він виконується за допомогою таких класів, як *Jdialog* та *JtextArea*.

```
public class Jdialog  
extends Dialog  
implements WindowConstants, Accessible, RootPaneContainer
```

Основний клас для створення діалогового вікна. Ви можете використовувати цей клас для створення власного діалогового вікна або викликати безліч методів класу в *JoptionPane*, щоб створити різноманітні стандартні діалоги.

Компонент *Jdialog* містить *JrootPane* як єдину дочірню організацію. *ContentPane* повинен бути батьком будь-яких дочірніх елементів *Jdialog*. Як зручне додавання та його варіанти, видалення та *setLayout* були замінені для пересилання до *ContentPane* за необхідності.

У додатку використовуємо дочірній клас *Jdialog*:

```
public class InfoDialog extends Jdialog
```

Це робиться для того, щоб створити спеціальне діалогове вікно, що відображає інформацію про результати розрахунків.

```
public class JtextArea  
extends JtextComponent
```

JtextArea – це багаторядкова область, яка відображає звичайний текст. Він призначений для полегшеного компонента, що забезпечує сумісність джерела з класом *java.awt.TextArea*, де це можна розумно зробити.

Цей компонент має можливості, яких немає в класі *java.awt.TextArea*. Для отримання додаткових можливостей слід звернутися до суперкласу. Альтернативними багаторядковими класами тексту з більшими можливостями є *JtextPane* та *JeditorPane*.

Java.awt.TextArea внутрішньо обробляє прокрутку. *JtextArea* відрізняється тим, що він не управляє прокруткою, але реалізує поворотний прокручуваний інтерфейс. Це дозволяє розміщувати його всередині *JscrollPane*, якщо бажана поведінка прокрутки, і використовувати безпосередньо, якщо прокрутка не потрібна.

Java.awt.TextArea має можливість виконувати перенесення рядків. Це контролювалося політикою горизонтальної прокрутки. Оскільки прокрутка не виконується *JtextArea* безпосередньо, зворотна сумісність повинна надаватися іншим способом. *JtextArea* має властивість прив'язки для переносу рядків, яка визначає, чи буде вона обертати рядки. За замовчуванням для властивості обтікання рядків встановлено значення *false* (не обертається).

Java.awt.TextArea має два властивості рядки та стовпці, які використовуються для визначення бажаного розміру. *JtextArea* використовує ці властивості, щоб вказати бажаний розмір області перегляду при розміщенні всередині *JscrollPane*, щоб відповідати функціональності, що надається *java.awt.TextArea*. *JtextArea* має бажаний розмір того, що потрібно для відображення всього тексту, щоб він нормально функціонував усередині *JscrollPane*. Якщо значення для рядків або стовпців дорівнює нулю, бажаний розмір вздовж цієї осі використовується для бажаного розміру області перегляду вздовж тієї ж осі.

Наприклад, якщо взяти до уваги метрику *LOC* та її одиниці виміру, ось результати, отримані у звіті.

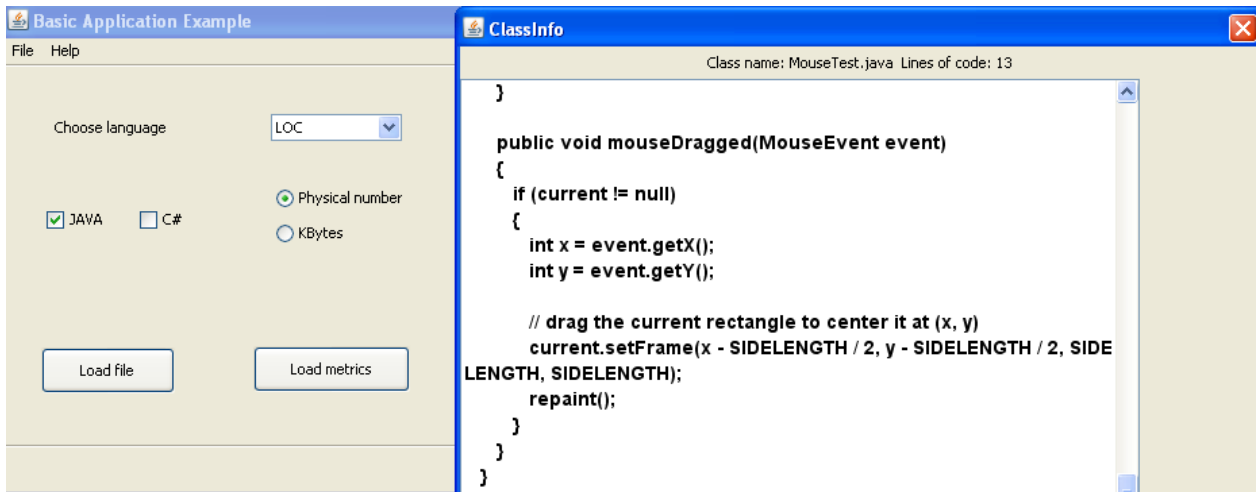


Рис. 3.10. Звіт про файл *JAVA* після використання метрики *LOC* у фізичних рядках коду

Якщо користувач вибере радіогайку в кбайт, звіт буде змінено наступним чином.

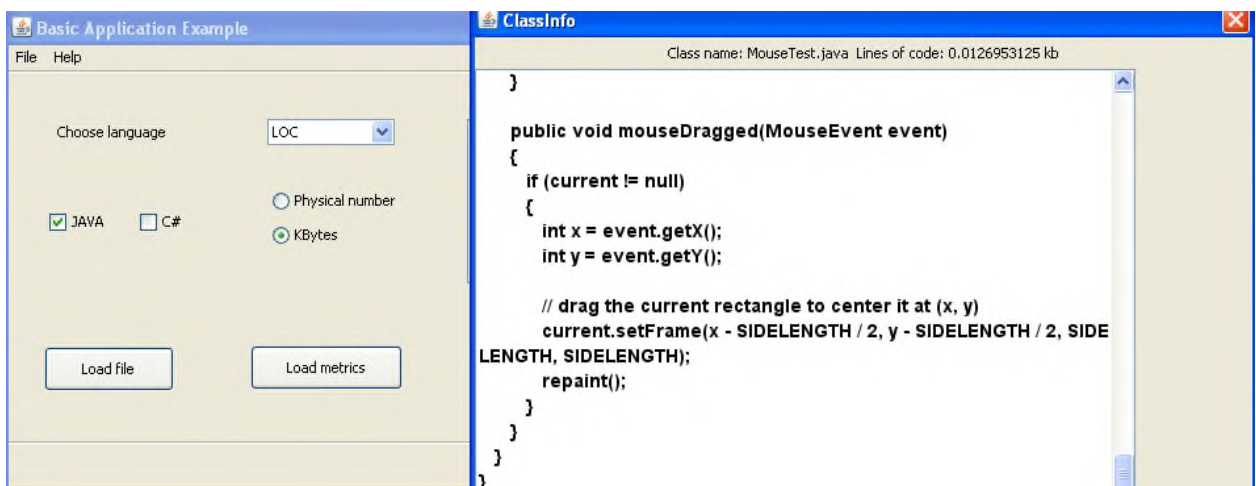


Рис. 3.11.Звіт про файл *JAVA* після використання метрики *LOC* у кбайт

Нарешті до програми була додана додаткова функція. Іноді важливо не тільки дати результати, але і вказати, що саме було виміряно в коді. Для цього був використаний інтерфейс *Highlighter*.

Highlighter h ;

*Highlighter.HighlightPainter hp = new
DefaultHighlighter.DefaultHighlightPainter(Color.CYAN);*

h=text.getHighlighter();

h.addHighlight(dot,dot+ str[i].length(), hp);

public interface Highlighter

Інтерфейс для об'єкта, який дозволяє розмітити фон кольоровими областями.

За допомогою цього інтерфейсу можна виділити деякі частини коду, щоб підкреслити, що робить ця метрика.

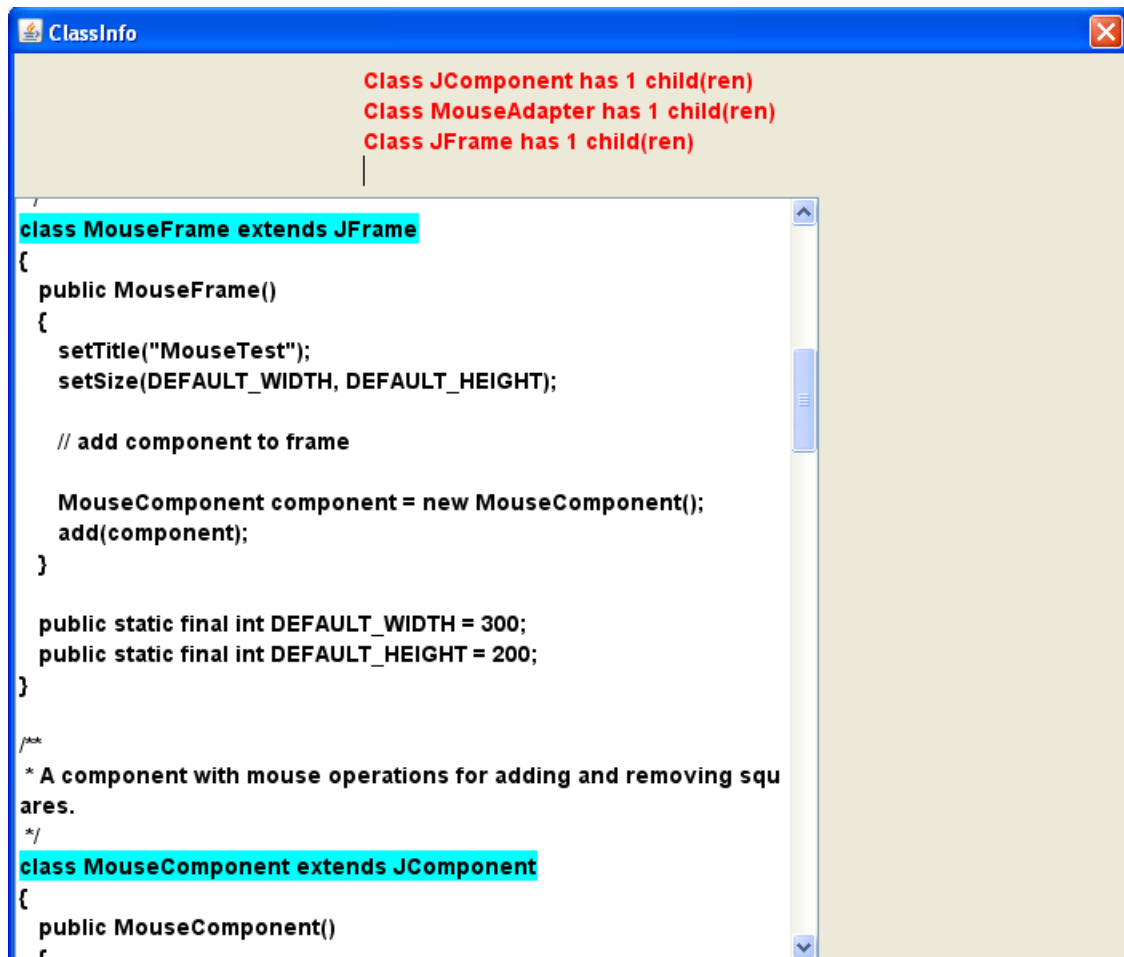


Рис. 3.12. Звіт про файл *JAVA* після використання метрики *NOC*

3.3. Дослідження та порівняльна характеристика даного додатка з відповідними професійними інструментами.

Перш за все, буде розглянуто *RSM (Resource Standard Metrics)*.

Resource Standard Metrics (RSM) – це метрики вихідного коду та інструмент аналізу якості. *RSM* забезпечує стандартний метод аналізу вихідного коду *C*, *ANSI C ++*, *C #* та *Java* у операційних системах. Він має унікальну здатність

підтримувати практично будь-яку операційну систему. Він надає найшвидший, найбільш гнучкий та простий у використанні інструмент для допомоги у вимірюванні якості коду та показників.

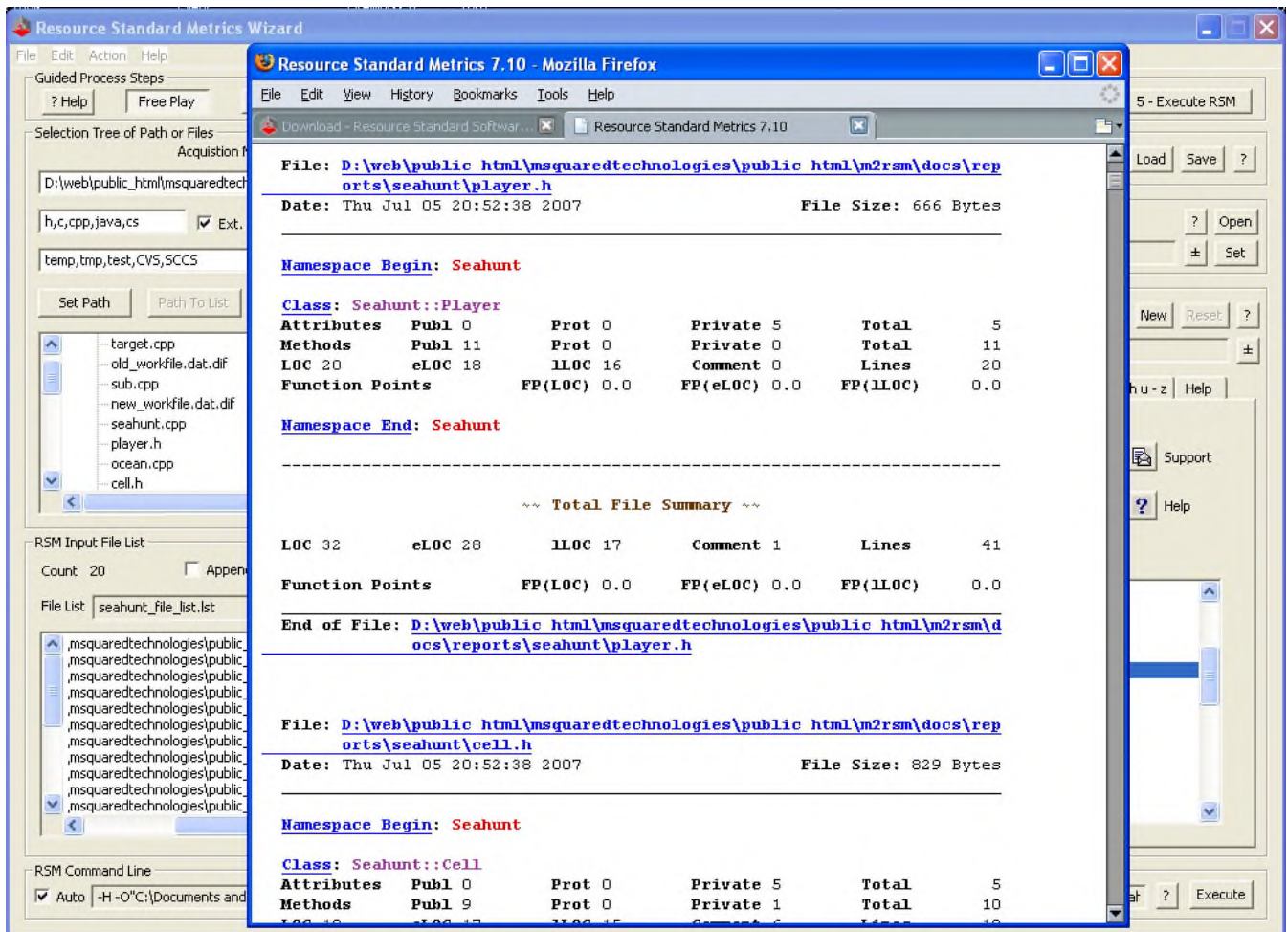


Рис. 3.13. Приклад звіту про стандартні показники ресурсів

Очевидно, що *RSM* надає більше можливостей вимірювання метрик, ніж додаток до диплома, хоча він має ряд недоліків, таких як:

- він не є відкритим кодом, це означає, що він не може бути змінений або вдосконалений будь-яким способом;
- він не забезпечує аналіз вихідного файлу з точки зору звіту з виділеними частинами коду, що беруть участь у вимірюванні метрики.

На відміну від *RSM*, заявку на диплом можна продовжити, додавши до неї нові показники. А також звіт, наданий заявкою, може бути більш повним, ніж звіт *RSM*.

Отже, програмне забезпечення *RSM* набагато функціональніше та ефективніше, що стосується проблем вимірювання чистих метрик, тим не менш, застосування дипломних програм є більш гнучким при розгляді можливостей розширення та освітніх делікатесів.

3.4. Висновки до розділу

У цьому розділі було реалізоване ПЗ оцінки якості програмного коду. Були описані можливості системи та інструменти, що використовуються для її розробки. Були проаналізовані найважливіші класи, що беруть участь у коді, та розглянута структура програми.

Додаток включає декілька класів, наприклад, клас *GetMetric*, відповідальний за завантаження метричних лічильників у програму. Він надає можливість створювати вичерпні звіти з виділеними частинами коду файлу.

Також у проекті беруть участь *XML*-файли, оскільки вони забезпечують простий та ефективний спосіб зберігання метрик. Програма не тільки вимірює метрики, але також дає можливість вибрати одиниці виміру для деяких з них, наприклад *LOC* вимірюється як у фізичних лініях, так і в Кбайт. Також пропонується коротке порівняння між даним додатком та професійним інструментом *RSM*.

ВИСНОВКИ

В результаті роботи над дипломною роботою був проведений огляд літератури з метою пошуку існуючих методів оцінки якості програмного коду.

Якість програмного забезпечення (*Software quality*) – це те наскільки програмне забезпечення задовольняє пропонованим до нього вимогам. Висунуті вимоги можуть залежати від багатьох критеріїв, що визначаються виходячи зі сфери застосування програмного продукту.

Існує набір стандартів *ISO 9000*, Який регулює загальні принципи забезпечення якості у всіх галузях. Найбільш важливими стандартами в розробці програмного забезпечення є:

- *ISO 9000: 2000 Quality management systems – Fundamentals and vocabulary*;
- *ISO 9001: 2000 Quality management systems – Requirements. Models for quality assurance in design, development, production, installation, and servicing*;
- *ISO 9004: 2000 Quality management systems – Guidelines for performance improvements*;
- *ISO / IEC 90003: 2004 Software engineering – Guidelines for the application of ISO 9001: 2000 to computer software*.

Так само в кожній компанії можуть бути розроблені свої стандарти якості програмного забезпечення, що відповідають конкретної специфіки роботи і відповідно до її вимог.

Можна виділити кілька основних критеріїв оцінки якості програмного забезпечення:

1. Якість вихідного коду.
 - Відповідність коду стандартам;
 - Легкість підтримки;
 - Мале число попереджень при компіляції.
2. Якість програмного продукту.
 - функціональність;

- надійність;
- зручність використання;
- ефективність;
- безпека.

Стає зрозуміло, що пред'являються вимоги повинні задовольняти потребам, як розробників програмного забезпечення, так і його користувачів.

Важливим аспектом є перевірка пропонованих вимог, їх повнота і правильність оцінки результатів. Для оцінки якості можуть застосовуватися різні методи:

- тестування (модульне, інтеграційне, системне, регресійне);
- статичний аналіз коду;
- динамічний аналіз коду;
- перегляд вихідних кодів програми (огляд коду).

Отримані з їх допомогою метрики характеризують різні параметри: швидкодія програми, споживані ресурси (оперативна пам'ять, завантаження процесора і т.д.), відповідність стандартам кодування, кількість помилок на одиницю рядків коду і багато іншого. Тестувати необхідно всі параметри, до яких пред'являються вимоги.

Важливо відзначити, що перевірка якості програмного забезпечення повинна проводитися на всіх етапах життєвого циклу. Це забезпечить максимальну якість розроблюваного програмного коду і як результат кінцевого програмного продукту. Не можна почати розробляти неякісний програмний продукт, і задуматися про його якість вже перед завершенням розробки.

У першому розділі були розглянуті об'єктно-орієнтовані метрики разом із інструментами для їх вимірювання. Існує багато різних підходів до класифікації об'єктно-орієнтованих метрик, починаючи від найпростіших, таких як кількість рядків коду, і закінчуючи більш складними, як відсутність згуртованості методів. У той же час існує багато програмного забезпечення, що служить для вимірювання та аналізу цих показників. Основна проблема полягає в тому, що більшість з них є або комерційними, або надають незначну допомогу в освітніх процесах, тому основною метою цього диплому є розробка програмного

забезпечення, яке не тільки має справу з метриками, але й підкреслює помилки та допомагає їх вирішити. Для створення такого продукту необхідно не лише ознайомитись з об'єктно-орієнтованими метриками, але й навчитися об'єднувати їх в єдину систему.

У другому розділі було розглянуто архітектуру для вилучення метрик разом із їх формальним описом за допомогою інструментів *sucj* як *OCL* та метамоделі *UML*. Більше того, бібліотека заходів *FLAME* служить вхідним матеріалом для формалізації метрик. Метою проекту *FLAME* є перетворення розвитку щільних бібліотек лінійної алгебри з мистецтва, зарезервованого для експертів, у науку, зрозумілу як новачкові, так і експерту. Замість того, щоб бути лише бібліотекою, проект охоплює нову нотацію для вираження алгоритмів, методологію систематичного виведення алгоритмів, інтерфейси прикладних програм (*API*) для представлення алгоритмів у коді та інструменти для механічного виведення, реалізації та аналізу алгоритмів та реалізації.

Також у цій главі було введено кілька схем, щоб розпочати проектування системи. Це діаграми використання, діаграми стану та діаграми класів. Вони дають розуміння системи та перші кроки до її реалізації.

У третьому розділі було обговорено питання про виконання диплому. Тут коротко були описані можливості системи та інструменти, що використовуються для її розробки. Були проаналізовані найважливіші класи, що беруть участь у коді, та розглянута структура програми.

Додаток включає декілька класів, наприклад клас *GetMetric*, відповідальний за завантаження метричних лічильників у програму. Він надає можливість створювати вичерпні звіти з виділеними частинами коду файлу.

Також у проекті беруть участь *XML*-файли, які забезпечують простий та ефективний спосіб зберігання метрик. Програма не тільки вимірює метрики, але також дає можливість вибрати одиниці виміру для деяких з них, наприклад *LOC*, вимірювані як у фізичних лініях, так і в Кбайт. Також пропонується коротке порівняння між даним додатком та професійним інструментом *RSM*.

Результати дипломної роботи рекомендується використовувати при розробці програмного модулю оцінки якості коду.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Мейер, Бертран. Объектно-ориентированное конструирование программных систем / Б. Мейер. – М. : Издательско-торговый дом "Русская редакция", 2005.
2. *Code Conventions for the Java™ Programming Language* / Режим доступа: <http://www.oracle.com/technetwork/java/index.html>
3. Блох, Джошуа. *Java™. Эффективное программирование* / Д. Блох. – М. : Лори, 2002.
4. Эккель, Брюс. *Философия Java. Библиотека программиста*. – 3-е издание / Б. Эккель. – СПб. : Питер, 2003.
5. Фаулер, Мартин. *Рефакторинг. Улучшение существующего кода* / М. Фаулер. – СПб. : Символ-Плюс, 2003.
6. Гамма, Эрих. *Приемы объектно-ориентированного проектирования. Паттерны проектирования* / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссиде. – СПб. : Питер, 2004.
7. Спольски, Джоэл. *Джоэл о программировании* / Д. Спольски. – СПб. : Символ-Плюс, 2006.
8. Буч, Гради. *Объектно-ориентированный анализ и проектирование с примерами приложений на C++* / Г. Буч. – Режим доступа: <http://khpriip.mipk.kharkiv.edu/library/case/buch/index.html>
9. Макконнелл, С. *Совершенный код. Мастер-класс* / С. Макконнелл. – М. : Издательско-торговый дом "Русская редакция"; СПб. : Питер, 2005.
10. Бек, Кент. *Экстремальное программирование* / К. Бек. – СПб. : Питер, 2002.
11. Аллен, Эрик. *Типичные ошибки проектирования. Библиотека программиста* / Э. Аллен. – СПб. : Питер, 2003.>
12. Спольски, Джоэл. *Джоэл о программировании* / Д. Спольски. – СПб. : Символ-Плюс, 2006.

13. Блох, Джошуа. *Java™. Эффективное программирование* / Д. Блох. – М. : Лори, 2002.
14. *Taylor, M.J. Soft Issues in IS Projects: Lessons from an SME Case Study* // *M.J. Taylor, J.L. DaCosta. – Systems Research and Behavioral Science.* – 1999. – V. 16, № 3.
15. *Kitchenham, B. Software quality: the elusive target* // *B. Kitchenham, S. Pfleeger. – IEEE Software*, 1996. – № 13(1)
16. ИСО 9000-3: ИСО 9001. Общее руководство качеством и стандарты по обеспечению качества. Часть 3: Руководящие указания по применению ИСО 9001 при разработке, поставке и обслуживанию программного обеспечения. Международная организация стандартов, Женева, 1991.
17. *Hellens, L. A. Information systems Quality versus Software Quality- A discussion from a managerial, an organizational and an engineering viewpoint* / *L. A. Hellens* // *Information and Software technology.* – 1998. – V. 39, № 12.
18. *Nonaka, I. The Knowledge-Creating Company – How Japanese Companies. Create the Dynamics of Innovation* / *I. Nonaka, H. Takeuchi.*- *Oxford University Press, Oxford*, 1995.
19. *Tervonen, I. Towards deeper co-understanding of software quality/ I. Tervonen, P. Kerola* // *Information and Software Technology.* – 1999. -V. 39, № 14-15.
20. ИСО/МЭК 9126. Информационные технологии. Оценка продукции программного обеспечения. Характеристики качества и инструкции по их применению. Международная организация стандартов, Женева, 1991.
21. *Paulk, M. C. Capability maturity model, version 1.1* // *M. C. Paulk, B. Curtis, M. B. Chissis, C. V. Weber* // *IEEE Software.* – 1993. – №10(4).
22. *Humphrey, W. S. CMM: Capability Maturity Model – A method for Assessing the Software Engineering Capability of Contractors* // *W. S. Humphrey.* – *Carnegie Mellon University, Software Engineering Institute.* – 1996.
23. Ситник Л.Ю. Система оцінки якості програмного коду веб-проектів// Тези доповідей наук.-практ. конф. “Сучасні тенденції розвитку системного програмування” (25-26 листопада 2020 р.). – К.: НАУ, 2020. – С. 28.

24. НД ТЗІ 1.1-003-99. Термінологія у області захисту інформації в комп'ютерних системах від несанкціонованого доступу. // Департамент спеціальних телекомунікаційних систем і захисту інформації Служби безпеки України. – Київ, 1999.

25. Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проекти) випускників Національного авіаційного університету. – К.: НАУ, 2017. – 63 с.

Додаток А

Схема алгоритму визначення метрик рядків коду

