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: Планування та розрахунок надійності програмного забезпечення

Виконавець: Соколюк Б.А.

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)

Соколюк Богдан Андрійович
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема роботи: "Планування та розрахунок надійності програмного забезпечення"
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 коду програмного забезпечення

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) Діаграма використання системи;
2) діаграма станів системи;
3) діаграма класів системи;
4) робочі вікна системи;
5) схема алгоритму роботи створеної програмної системи;
6) схема алгоритму розрахунку метрики *LOC*.

6. Календарний план

№ п/п	Етапи виконання дипломної роботи	Термін виконання етапів	Примітка
1		05.10.2020 – 06.10.2020	
2		07.10.2020 – 10.11.2020	
3		11.11.2020 – 14.11.2020	
4		15.11.2020 – 19.11.2020	
5		29.11.2020 – 13.12.2020	
6		21.12.2020 – 25.12.2020	

7. Дата видачі завдання 05.10.2020

Керівник Нечипорук В.В.
(підпис)

Завдання прийняв до виконання Соколюк Б.А.
(підпис студента)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи “Планування та розрахунок надійності програмного забезпечення”: 81 с., 16 рис., 25 літературних джерел, 1 додаток.

МЕТРИКИ НАДІЙНОСТІ, НАДІЙНІСТЬ ПРОГРАМНОГО КОДУ, ПРОГРАМНИЙ ПРОЕКТ, ОЦІНКА ЯКОСТІ

Мета дослідження – розробити систему оцінки надійності коду програмних проектів.

Об’єкт дослідження – програмний код веб-проектів.

Предмет дослідження – система оцінки надійності коду програмних проектів.

Метод дослідження – аналіз, синтез, моделювання і програмна реалізація системи оцінки надійності програмного коду програмних 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 для оцінки їх надійності. Розвиток проекту передбачається за рахунок розширення групи параметрів оцінки якості коду.

Публікація: Соколюк Б.А. Система розрахунку надійності програмного забезпечення// Тези доповідей наук.-практ. конф. “Сучасні тенденції розвитку системного програмування” (25-26 листопада 2020 р.). – К.: НАУ, 2020. – С. 37.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ВСТУП

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

Під якісним програмним забезпеченням розуміється програмне забезпечення, яке не містить помилок або дефектів, доставлено вчасно і в рамках встановленого бюджету, відповідає вимогам і / або очікуванням і є ремонтопрігодним. У контексті розробки програмного забезпечення якість програмного забезпечення відображає як функціональне якість, так і структурний якість.

- Функціональне якість програмного забезпечення- відображає, наскільки добре воно відповідає заданому дизайну, на підставі функціональних вимог або специфікацій.
- Структурний якість програмного забезпечення.Воно стосується обробки не функціональних вимог, які підтримують виконання функціональних вимог, таких як надійність або ремонтопридатність, і ступеня, в якій програмне забезпечення було вироблено правильно.
- Software Quality Assurance- Software Quality Assurance (SQA) - це комплекс заходів, спрямованих на забезпечення якості процесів розробки програмного забезпечення, які в кінцевому підсумку призводять до якісних програмних продуктів. Діяльність встановлює і оцінює процеси, які виробляють продукти. Це включає процесно-орієнтовані дії.
- Контроль якості програмного забезпечення- Контроль якості програмного забезпечення (SQC) - це комплекс заходів щодо забезпечення якості програмних продуктів. Ці дії спрямовані на виявлення дефектів у фактичній продукції. Це включає в себе дії, орієнтовані на продукт.

Функціональне якість програмного забезпечення- відображає, наскільки добре воно відповідає заданому дизайну, на підставі функціональних вимог або специфікацій.

Структурний якість програмного забезпечення.Воно стосується обробки не функціональних вимог, які підтримують виконання функціональних вимог, таких як надійність або ремонтопридатність, і ступеня, в якій програмне забезпечення було вироблено правильно.

Software Quality Assurance- Software Quality Assurance (SQA) - це комплекс заходів, спрямованих на забезпечення якості процесів розробки програмного забезпечення, які в кінцевому підсумку призводять до якісних програмних продуктів. Діяльність встановлює і оцінює процеси, які виробляють продукти. Це включає процесно-орієнтовані дії.

Контроль якості програмного забезпечення- Контроль якості програмного забезпечення (SQC) - це комплекс заходів щодо забезпечення якості програмних продуктів. Ці дії спрямовані на виявлення дефектів у фактичній продукції. Це включає в себе дії, орієнтовані на продукт.

Проблема якості програмного забезпечення

В індустрії програмного забезпечення розробники ніколи не заявлять, що програмне забезпечення не має дефектів, на відміну від інших виробників промислової продукції. Ця різниця обумовлена наступними причинами.

складність продукту

Це кількість режимів роботи, які допускає продукт. Зазвичай промисловий продукт допускає тільки кілька тисяч режимів роботи з різними комбінаціями настроювань машини. Проте, програмні пакети надають мільйони операційних можливостей. Отже, забезпечення всіх цих операційних можливостей є серйозною проблемою для індустрії програмного забезпечення.

видимість продукту

Оскільки промислові вироби видно, більшість його дефектів можна виявити в процесі виробництва. Також відсутність деталі в промисловому продукті може бути легко виявлено в продукті. Однак дефекти в програмних продуктах, які зберігаються на дискетах або компакт-дисках, невидимі.

Розробка продукту і виробничий процес

У промисловому продукті дефекти можуть бути виявлені на наступних етапах -

• Розробка продукту- На цьому етапі проектувальники і співробітники відділу забезпечення якості (QA) перевіряють і тестують прототип продукту для виявлення його дефектів.

• Планування виробництва продукції. нацьому етапі виробничий процес і інструменти розробляються і готуються. Цей етап також дає можливість оглянути продукт, щоб виявити дефекти, які залишилися непоміченими на етапі розробки.

• виробництво- на цьому етапі застосовуються процедури QA для виявлення збоїв самих продуктів. Дефекти продукту, виявлені в першому періоді виробництва, зазвичай можуть бути виправлені шляхом зміни конструкції або матеріалів продукту або виробничих інструментів таким чином, щоб усунути такі дефекти в продуктах, що випускаються в майбутньому.

Розробка продукту- На цьому етапі проектувальники і співробітники відділу забезпечення якості (QA) перевіряють і тестують прототип продукту для виявлення його дефектів.

Планування виробництва продукції. нацьому етапі виробничий процес і інструменти розробляються і готуються. Цей етап також дає можливість оглянути продукт, щоб виявити дефекти, які залишилися непоміченими на етапі розробки.

виробництво- на цьому етапі застосовуються процедури QA для виявлення збоїв самих продуктів. Дефекти продукту, виявлені в першому періоді виробництва, зазвичай можуть бути виправлені шляхом зміни конструкції або матеріалів продукту або виробничих інструментів таким чином, щоб усунути такі дефекти в продуктах, що випускаються в майбутньому.

Однак в разі програмного забезпечення єдина фаза, в якій можуть бути виявлені дефекти, - це фаза розробки. У разі програмного забезпечення етапи планування виробництва і виготовлення продукту не потрібні, оскільки виготовлення копій програмного забезпечення і друк посібників з програмного забезпечення здійснюються автоматично.

Різні чинники, які впливають на програмне забезпечення, називаються програмними факторами. Їх можна широко розділити на дві категорії. Перша

категорія факторів - це ті, які можуть бути виміряні безпосередньо, наприклад, число логічних помилок, а друга категорія об'єднує ті чинники, які можуть бути виміряні тільки побічно. Наприклад, ремонтопридатність, але кожен з факторів повинен бути визначений для перевірки змісту та контролю якості.

Кілька моделей чинників якості програмного забезпечення і їх категоризація були запропоновані за ці роки. Класична модель чинників якості програмного забезпечення, запропонована McCall, складається з 11 факторів (McCall et al., 1977). Аналогічно, моделі, що складаються з 12-15 чинників, були запропоновані Дойчем і Віллісом (1988) і Евансом і Марсініаком (1987).

Всі ці моделі істотно не відрізняються від моделі Маккола. Факторна модель Макколла надає практичний, сучасний метод класифікації вимог до програмного забезпечення (Pressman, 2000).

Модель Фактора Маккола

Ця модель класифікує всі вимоги до програмного забезпечення по 11 показникам якості програмного забезпечення. Ці 11 факторів згруповані в три категорії - експлуатація продукту, перегляд продукту і фактори переходу продукту.

• Фактори роботи продукту- правильність, надійність, ефективність, цілісність, зручність використання.

• Фактори перегляду продукту- ремонтопридатність, гнучкість, тестованих.

• Фактори переходу продукту- Переносимость, Можливість повторного використання, Сумісність.

Фактори роботи продукту- правильність, надійність, ефективність, цілісність, зручність використання.

Фактори перегляду продукту- ремонтопридатність, гнучкість, тестованих.

Фактори переходу продукту- Переносимость, Можливість повторного використання, Сумісність.

Фактори якості програмного забезпечення роботи продукту

Згідно з моделлю Макколла, категорія роботи продукту включає п'ять факторів якості програмного забезпечення, які відповідають вимогам, які

безпосередньо впливають на щоденну роботу програмного забезпечення. Вони полягають в наступному -

правильність

Ці вимоги стосуються правильності виведення програмного забезпечення системи. Вони включають в себе -

- Вихідна місія
- Необхідна точність виведення, на яку можуть негативно вплинути неточні дані або неточні розрахунки.
- Повнота виведеної інформації, на яку можуть вплинути неповні дані.
- Актуальність інформації, яка визначається як час між подією і відповіддю системи програмного забезпечення.
- Доступність інформації.
- Стандарти кодування і документування програмного забезпечення системи.

Вихідна місія

Необхідна точність виведення, на яку можуть негативно вплинути неточні дані або неточні розрахунки.

Повнота виведеної інформації, на яку можуть вплинути неповні дані.

Актуальність інформації, яка визначається як час між подією і відповіддю системи програмного забезпечення.

Доступність інформації.

Стандарти кодування і документування програмного забезпечення системи.

надійність

Вимоги до надійності пов'язані з відмовою обслуговування. Вони визначають максимально допустиму частоту відмов програмної системи і можуть ставитися до всієї системи або до однієї або декількох її окремих функцій.

ККД

Він стосується апаратних ресурсів, необхідних для виконання різних функцій системи програмного забезпечення. Він включає в себе можливості

обробки (в МГц), ємність зберігання (в МБ або ГБ) і можливість передачі даних (в МБПС або ГБПС).

Він також стосується часу між підзарядкою портативних блоків системи, таких як блоки інформаційної системи, розташовані в портативних комп'ютерах, або метеорологічні блоки, розміщені на відкритому повітрі.

цілісність

Цей фактор пов'язаний з безпекою системи програмного забезпечення, тобто для запобігання доступу сторонніх осіб, а також для розмежування групи людей, яким даються дозволу на читання і запис.

юзабіліті

Вимоги юзабіліті пов'язані з кадровими ресурсами, необхідними для навчання нового співробітника і експлуатації системи програмного забезпечення.

Фактори якості редакції продукту

Згідно з моделлю Маккола, три категорії якості програмного забезпечення включені в категорію редакції продукту. Ці чинники полягають в наступному -

ремонтопридатність

Цей фактор враховує зусилля, які будуть потрібні користувачам і обслуговуючому персоналу для визначення причин збоїв програмного забезпечення, виправлення збоїв і перевірки успішності виправлень.

гнучкість

Цей фактор пов'язаний з можливостями і зусиллями, необхідними для підтримки адаптивного обслуговування програмного забезпечення. Це включає в себе адаптацію поточного програмного забезпечення до додаткових обставин і клієнтам без зміни програмного забезпечення. Вимоги цього фактора також підтримують вчинені дії з обслуговування, такі як зміни і доповнення програмного забезпечення, щоб поліпшити його обслуговування і адаптувати його до змін в технічній або комерційної середовищі фірми.

здатність бути свідком у суді

Вимоги до тестованості відносяться як до тестування програмної системи, так і до її роботи. Він включає попередньо певні проміжні результати, файли

журналу, а також автоматичну діагностику, виконувану програмної системою перед запуском системи, щоб з'ясувати, чи всі компоненти системи знаходяться в робочому стані, і отримати звіт про виявлені несправності. Інший тип цих вимог стосується автоматичних діагностичних перевірок, які застосовуються фахівцями з технічного обслуговування для виявлення причин збоїв програмного забезпечення.

Фактор якості програмного забезпечення для переходу продукту

Згідно з моделлю Макколла, три категорії якості програмного забезпечення включені в категорію переходу продукту, яка стосується адаптації програмного забезпечення до інших середах і його взаємодії з іншими програмними системами. Ці чинники полягають в наступному -

портативність

Вимоги переносимості мають тенденцію до адаптації програмної системи до інших середовищ, що складається з іншого обладнання, різних операційних систем і так далі. Програмне забезпечення повинно мати можливість продовжувати використовувати один і той же базове програмне забезпечення в різних ситуаціях.

Повторне використання

Цей фактор пов'язаний з використанням програмних модулів, спочатку розроблених для одного проекту, в новому проекті Програми, який розробляється в даний час. Вони можуть також дозволити майбутнім проектам використовувати даний модуль або групу модулів розробленого в даний час програмного забезпечення. Очікується, що повторне використання програмного забезпечення дозволить заощадити ресурси розробки, скоротити період розробки і забезпечити більш якісні модулі.

Interoperability

Вимоги до функціональної сумісності спрямовані на створення інтерфейсів з іншими програмними системами або з іншими апаратними засобами обладнання. Наприклад, вбудоване програмне забезпечення виробничого обладнання та випробувального обладнання взаємодіє з програмним забезпеченням управління виробництвом.

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

Мета дослідження – розробити систему оцінки надійності коду програмних проектів.

Об’єкт дослідження – програмний код веб-проектів.

Предмет дослідження – система оцінки надійності коду програмних проектів.

Метод дослідження – аналіз, синтез, моделювання і програмна реалізація системи оцінки надійності програмного коду програмних проектів.

РОЗДІЛ 1
АНАЛІЗ МЕТОДІВ РОЗРАХУНКУ НАДІЙНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Формальний опис об’єктно-орієнтованих метрик.

1.1.1. Аналіз методів отримання метрик.

Показники проектування є корисними засобами для підвищення якості програмного забезпечення. Ряд об'єктно-орієнтованих показників було запропоновано як корисні для ідентифікації класів, схильних до несправностей, для прогнозування необхідних зусиль з технічного обслуговування, для оцінки продуктивності та для оцінки зусиль з переробки.

Для отримання метрик проектування програмного забезпечення, що розробляється, більшість існуючих підходів вимірюють метрики шляхом аналізу вихідного коду програмного забезпечення. Такі підходи можна застосовувати лише на пізній фазі розробки програмного забезпечення, обмежуючи тим самим корисність метрик проектування на ранніх фазах життєвого циклу розробки. Інша проблема полягає в тому, що такі показники описуються неофіційно, що обмежує підтримку інструменту.

Сучасний рівень техніки для додання точності об’єктно-орієнтованому моделюванню за допомогою *OCL* зображено на рисунку 1.1. Комерційні засоби моделювання *UML* – як *Rational*, *Objectory*, *JDeveloper*, *QuickUML*, *PowerDesigner* тощо – забезпечують деякі редактори графічних діаграм, що дозволяють будувати моделі систем. Моделі – на рисунку представлені *X*, *Y* та *Z* – зберігаються у сховищі інструментів.

Мова обмеження об’єктів (*OCL*) – це декларативна мова для опису правил, що застосовуються до моделей уніфікованої мови моделювання (*UML*), розроблених в *IBM* і нині частиною стандарту *UML*.

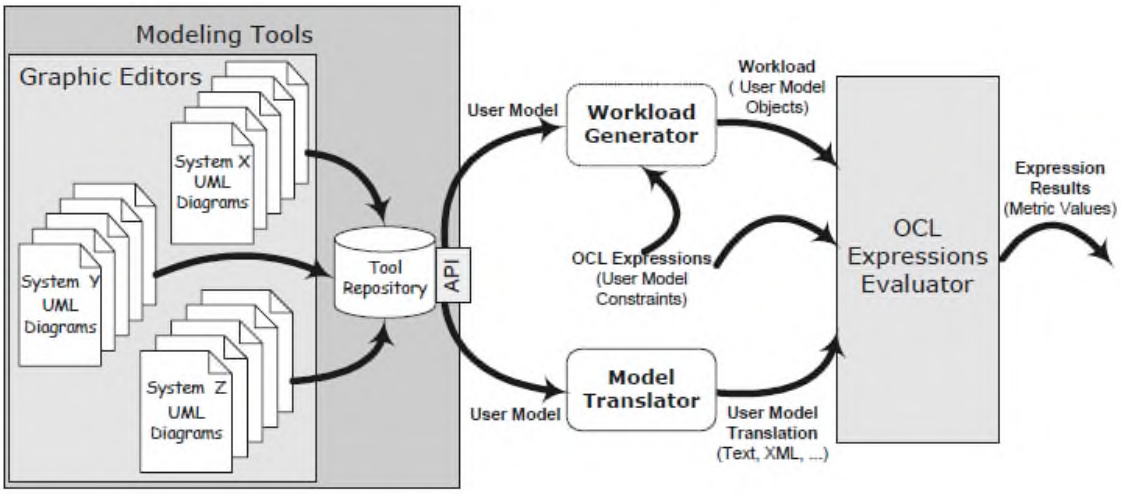

Рис. 1.1. Архітектура модельного рівня

Спочатку *OCL* був лише формальним розширенням мови специфікації до *UML*. *OCL* тепер може використовуватися з будь-якою мета-моделлю групи об'єктів управління об'єктами (*MOF*) (*OMG*), включаючи *UML*. Мова обмеження об'єктів – це точна мова тексту, яка забезпечує вираження обмежень та запитів об'єктів для будь-якої моделі *MOF* або метамоделі, яка інакше не може бути виражена схематичним позначенням. *OCL* є ключовим компонентом нової стандартної рекомендації *OMG* щодо трансформації моделей, специфікації *Queries* / *Views* / *Transformations* (*QVT*).

На сьогодні засоби моделювання не пропонують засобів для оцінки виразів *OCL* порівняно з моделями у сховищі. Тим не менше, декілька інструментів – наприклад, *Use, Cybernetic Parser, Elixer, ModelRun* тощо – виходять із дослідницьких проектів і можуть бути використані для формалізації моделей за умови, що їх можна експортувати з відповідним форматом введення в інструменти *OCL*. Зазвичай текстовий файл, що представляє модель, генерується перекладачем (*XML* може бути використаний як приклад для представлення моделі).

Після перетворення файлу моделі (у подання, яке можна зрозуміти засобами *OCL*), створюються реальні екземпляри сутностей на діаграмі, і модель заповнюється (тобто велика кількість об'єктів, що відповідають сутностям у моделі, створюються). Ці екземпляри є основою тверджень, побудованих за

допомогою *OCL*. Для цього процесу дуже допоміг би один інструмент генератора робочого навантаження (див. Рис. 3.1), оскільки часто екземпляри моделі *UML* виконуються «вручну».

Діаграми, що складають моделі та їх відповідні об'єкти, служать вхідними даними для інструменту оцінки *OCL*, який приймає перетворене представлення діаграми, додані обмеження *OCL* та екземпляри моделі та оцінює кожне з обмежень, показуючи результати. Оцінювач *OCL* повинен мати можливість перевіряти, порушено чи ні обмеження, для певного навантаження екземплярів моделі користувача. Більше того, він повинен оцінювати кожне твердження окремо та надавати зворотний зв'язок щодо того, які тестові випадки проектування відповідають або порушують обмеження.

Хоча архітектура, що зображена на рисунку 1.1, відповідає оцінці рівня моделі, архітектура, зображена на рисунку 1.2, пов'язана з оцінкою рівня метамоделі. В архітектурі рівня мета-моделі зберігаються всі функціональні можливості рівня моделі. Незважаючи на це, є два основних доповнення: одне – це введення відповідних діаграм класівдомета-модель *UML*. Іншим є введення автоматичного генератора екземплярів, який буде приймати мета-модель і автоматично генерувати всі екземпляри для її заповнення. Використовуючи ці функції (метамодель та відповідні екземпляри), формалізовано та перевірено декілька наборів метрик дизайну, які можна знайти в літературі, виражених як вирази *OCL* для метамоделі *UML*.

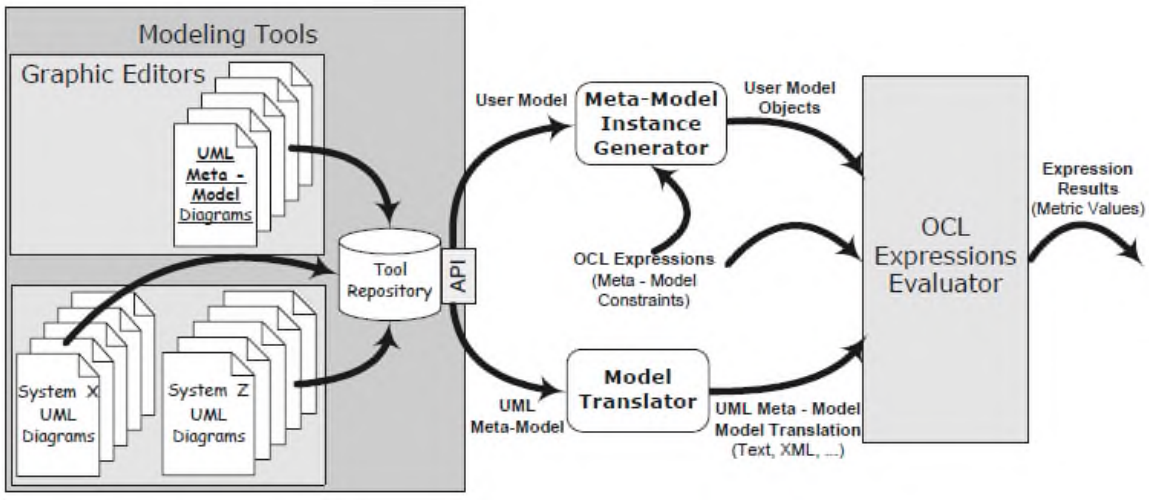

Рис. 1.2. Архітектура рівня мета-моделі

Текстову версію (наприклад, у форматі *XML*) метамоделі *UML* можна отримати з діаграми класів метамоделі *UML*, використовуючи архітектуру, представлену на рисунку 1.2. Члени групи *QUASAR* розробили генератор екземпляра метамоделі для створення екземплярів об'єктів, що відповідають метакласам.

1.1.2. Метрики для об'єктно-орієнтованого проектування

Більшість розробок програмного забезпечення стикаються з ризиками зникнення графіка та / або перевитрати коштів. Таким чином, ефективний розподіл ресурсів, зменшення складності проектування та прийняття ефективних технологій програмної інженерії є ключовими для вирішення або зменшення таких ризиків. Метрики проектування, які є кількісними показниками складності програмного забезпечення або дизайну, пропонуються як корисні засоби для досягнення цих цілей.

Ряд метрик проектування було вивчено і продемонстровано як корисне з кількох аспектів, таких як розуміння, оцінка та оцінка складності конструкцій, оцінка складності програмного забезпечення на основі його конструкції, виявлення програмних блоків, схильних до несправностей, і правдоподібні типи несправностей , та оцінка необхідних зусиль з технічного обслуговування.

Тим не менше, незважаючи на наукові дослідження, метрики проектування не отримали широкого застосування, як очікувалося, у галузі програмного забезпечення. Однією з основних проблем, яка обмежила їх використання, є відсутність доступних інструментів для вимірювання метрик, що, в свою чергу, може бути наслідком їх неточної специфікації.

Цей розділ знайомить як з неформальними, так і з офіційними визначеннями чотирьох добре відомих наборів метрик проектування, а саме: *MOOD* і *MOOD*2 – Метрики для об'єктно-орієнтованого проектування, *MOOSE* – Метрики для об'єктно-орієнтованого програмного забезпечення, *EMOOSE* – Розширені *MOOSE* та *QMOOD* – Модель якості для об'єктно-орієнтованих – метричних наборів.

Більшість метрик у цих наборах спочатку були визначені неофіційно, використовуючи природну мову, і основний внесок цієї роботи залишається у підвищенні точності, через їх офіційні визначення. Формалізація здійснюється за допомогою *OCL* та метамоделі *UML*. Більше того, бібліотека заходів *FLAME* служить вхідними даними для формалізації метрик.

Замість використання контексту класу мета-класу, коли він неодмінно згадується в неофіційних визначеннях метрик, було прийнято контекст класифікатора. Така зміна робить формалізацію метрик більш гнучкою, оскільки визначення можна застосовувати в підкласах Класифікатора (включаючи переважно метакласи *Class* та *DataTypes*). Наприклад, розглянемо випадок, коли користувач хоче визначити нову структуру типів даних у своїй моделі, включаючи класи як Дата та час, або Клас відсотків як підтип класу *Real*. За допомогою формалізації в контексті класифікатора можна оцінити нову структуру типів даних користувача. Це було б неможливо, якби визначення були обмежені контекстом Класу.

У деяких випадках формалізація метрики неможлива. Коли це відбувається, область, що відповідає полю “Формальне визначення”, залишається лише з підписом метрики. Для кожного випадку пояснюються причини, що забороняють оформлення.

1.1.3. Методика визначення надійності програмного коду *MOOD*

Набір показників *MOOD* було описано в 1-му розділі.Після деяких експериментів стало очевидним, що деякі важливі аспекти конструкції не вимірюються за допомогою метрик *MOOD*, а саме існування поліморфізму та кількість повторного використання. Крім того, набір *MOOD* враховував лише показники, розраховані в межах заданої специфікації та людинир виконувані системи (додатки) зазвичай складаються за кількома специфікаціями.

Це призвело до народження набору *MOOD*2, метрики якого були розділені на дві групи: метрики внутрішньої специфікації та метрики між специфікаціями. До першої групи належать ті показники, які посилаються лише на специфікацію контексту і визначення яких покладається на інформацію, що міститься

виключно на ній. Тому вони не мають параметрів. Друга група включає ті метрики, які визначення стосується взаємозв'язку між специфікацією контексту та тією, яка передається як аргумент. Таким чином, деякі метрики (успадковування та зчеплення) відображають внутрішні (в межах специфікації) аспекти проектування, тоді як інші відображають зовнішні (серед окремих специфікацій).

У таблицях 1.1 та 1.2 представлені набори метрик *MOOD* та *MOOD*2. Нові показники (на *MOOD*2) позначені зірочкою. Кілька оригінальних показників *MOOD* були перейменовані для узгодження імен.

Таблиця 1.1.

Внутрішньо-специфікаційні метрики

Скорочення	**Позначення**
AIF	Фактор успадкування атрибутів
OIF	Фактор успадкування операцій
IIF	Внутрішній фактор успадкування
AHF	Фактор приховування атрибутів
OHF	Фактор приховування операцій
AHEF	Атрибути Фактор приховування ефективності
OHEF	Операції, що приховують фактор ефективності
БНФ	Фактор поведінкового поліморфізму
PPF	Фактор параметричного поліморфізму
CCF	Фактор зв'язку класу
ICF	Внутрішній фактор зчеплення

Таблиця 2.2.

Метрики між специфікаціями

Скорочення	**Позначення**
EIF (*S*)	Зовнішній фактор успадкування
ECF (*S*)	Зовнішній фактор зчеплення
PRF (*S*)	Потенційний фактор повторного використання
ARF (*S*)	Фактичний коефіцієнт повторного використання
REF (*S*)	Повторне використання коефіцієнта ефективності

Показники *MOOD*2 зберігають основні характеристики вихідного набору. Усівизначаються як частки, де чисельник представляє фактичне значення проектної характеристики, що вимірюється, тоді як знаменник представляє її теоретичне максимальне значення. В результаті вони приймають значення у відсотковому масштабі (реальні числа в інтервалі [0, 1]).

Ще одним удосконаленням набору *MOOD*2 є те, що їх визначення було зроблено композиційним способом, заснованим на наборі допоміжних функцій, на різних рівнях абстракції, а саме атрибуті, експлуатації, класі та специфікації. Кожен із цих рівнів відповідає одному метакласу в метамоделі *GOODLY*.

Замість використання метамоделі *GOODLY* та *MOODlib* у цьому документі використовується метамодель *UML* та бібліотека *FLAME*. Оскільки *UML* стає загальновідомим стандартом, ці заміни покращать підтримку інструментів та витяг метрик.

1.1.4. Методика визначення надійності програмного коду *MOOSE*

Одним із найпопулярніших метричних номерів є набір *MOOSE*, запропонований Чідамбером та Кемерером.

Незважаючи на те, що цей набір широко поширений, не всі метрики є дизайнерськими, і деякі з них не можуть бути формалізовані на *UML*-моделі. У будь-якому випадку було проведено кілька досліджень для перевірки метрик *MOOSE* і показало, що вони є корисними показниками якості для прогнозування класів, що піддаються несправностям, та зусиль з технічного обслуговування, а також є важливими економічними показниками.

Набір показників *MOOSE* не повністю пов'язаний з дизайном. Це обмежує формалізацію. Наприклад, метричний *LCOM* залежить від вихідного коду. Більше того, слід зазначити одне обмеження, враховуючи цей набір. Метричний *WMC* забезпечує лише найпростішу реалізацію щодо складностей (він розглядає всі складності методів як унітарні). Однак можуть бути запропоновані різні реалізації.

1.1.5. Методика визначення надійності програмного коду *EMOOSE*

Цей набір був задуманий як продовження метрик *MOOSE*. Показники *EMOOSE* (*Extended MOOSE*) були створені Вей Лі, Саллі Генрі.

Показники *EMOOSE* містять ті, що визначені в наборі *MOOSE*, а також ті, що проілюстровані нижче. Набір має обмежену деталізацію, оскільки метрики застосовуються лише до контексту Класу (Класифікатора).

Як встановлено *MOOSE, EMOOSE* не суто пов'язаний з дизайном. Це означає, що деякі показники не можуть бути формалізовані за допомогою нашого підходу (*OCL* на метамоделі *UML*), як це відбувається з *MPC* та *SIZE* 1.

Зверніть увагу, що функція *NOM* дасть той самий результат, що і *WMC* у групі *MOOSE*. Це означає, що функція *NOM* у розширеній групі (*EMOOSE*) має сенс, коли *WMC* обчислює складність методів по-іншому, інакше результати метрики дублюються.

1.1.6. Методика визначення надійності програмного коду *QMOOD*

QMOOD – це модель якості для оцінки зовнішніх атрибутів якості на високому рівні, таких як багаторазове використання, функціональність та гнучкість об’єктно-орієнтованих конструкцій на основі внутрішніх властивостей компонент дизайну *C ++*.

Він визначає набір метрик, які можна застосовувати як в контексті системи, так і класів (*Package* або *Classifier*, при використанні метамоделі *UML*). Цей набір метрик має кілька проблем, головним чином тому, що він залежить від мови та тому, що деякі визначення однакові, навіть мають різні назви. Крім того, ці показники мають велику різницю в шкалах, що використовуються в результатах, змішуючи цілі числа, дійсні та відсоткові значення.

Нарешті, їх визначення природничою мовою може дати різні тлумачення результатів, що ускладнило нашу роботу. Коли це трапляється, представлені альтернативні тлумачення в полі “Коментарі”.

Набір показників *QMOOD* страждає від деяких недоліків. По-перше, він поєднує дизайн з кодом (як *MOOSE* та *QMOOD*), тому деякі показники не можуть бути формалізовані. По-друге, він був створений з урахуванням лише моделей *C ++*, і він використовує деякі концепції, які не підтримуються або не є стандартом для інших мов, як вбудовані та віртуальні методи. По-третє, деякі показники можуть мати більше одного тлумачення, що ускладнює їх стандартизацію та перевірку. По-четверте, деякі показники здаються однаковими, навіть мають різні назви.

Хоча можна було формалізувати більшість метрик у наборі, в деяких випадках, коли виникають різні інтерпретації, деякі метрики все одно можуть бути неправильно визначені.

1.2. Вимоги до об’єктно-орієнтованих засобів оцінки коду.

Основними вимогами до розробленої системи є розширюваність, простота у використанні та надання вичерпних звітів.

У програмній та системній інженерії варіант використання – це перелік етапів, що зазвичай визначають взаємодію між роллю (відомою в *UML* як "актор") та системою для досягнення мети. Актор може бути людиною або зовнішньою системою. У системній інженерії випадки використання використовуються на вищому рівні, ніж у програмній інженерії, часто представляючи місії або цілі зацікавлених сторін.

Як показано на рисунку 1.3, у будь-якої людини, яка бажає користуватися системою, є кілька основних вимог до неї.

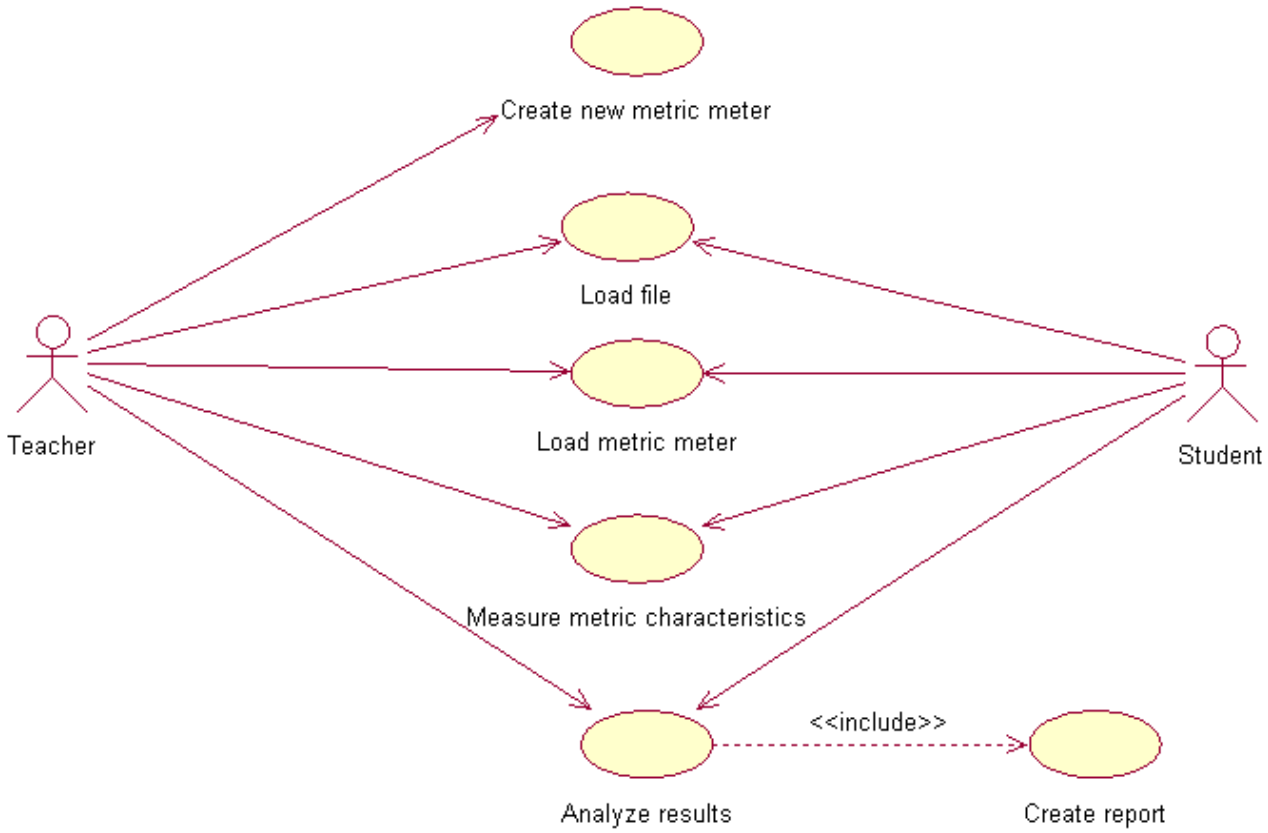

Рис. 1.3. Діаграма використання системи

Розглянемо метрики специфікацій для оцінки надійності програмного коду:

1) Специфікація випадку використання: створити новий метричний лічильник.

Назва: створити новий метричний метр.

Короткий зміст: цей варіант використання дозволяє вчителю написати новий метричний лічильник, який буде доданий до системи з метою розширення системи.

Актори: Вчитель.

2) Специфікація випадку використання: метричний навантажувач.

Назва: метричний набір навантаження.

Короткий зміст: служить для завантаження метричного лічильника в систему.

Актори: Викладач, студент.

3) Специфікація випадку використання: завантажити файл.

Назва: завантажити файл.

Короткий зміст: цей варіант використання дозволяє завантажувати файл в систему для маніпулювання вихідним кодом, вказаним у ній.

Актори: Викладач, студент.

4) Специфікація випадку використання: вимірювання метричних характеристик.

Назва: виміряти метричні характеристики.

Короткий зміст: цей варіант використання дозволяє вимірювати об'єктно-орієнтований код для метрик, доданих до системи раніше.

Актори: Викладач, студент.

5) Специфікація випадку використання: проаналізуйте результати

Назва: аналізувати результати.

Короткий зміст: аналізує кінцеві результати, отримані після метричного вимірювання.

Актори: Викладач, студент.

6) Специфікація випадку використання: створити звіт.

Назва: створити звіт.

Короткий зміст: дозволяє сформувати вичерпний звіт на основі раніше проведених вимірювань.

Актори: Викладач, студент

Для правильної розробки системи вводиться діаграма стану (рис. 1.4). Діаграма стану – це тип діаграми, що використовується в інформатиці та суміжних областях для опису поведінки систем. Діаграми станів вимагають, щоб описана система складалася з кінцевої кількості станів. Діаграми стану використовуються для абстрактного опису поведінки системи. Ця поведінка аналізується і представляється у серії подій, які можуть відбуватися в одному або кількох можливих станах.

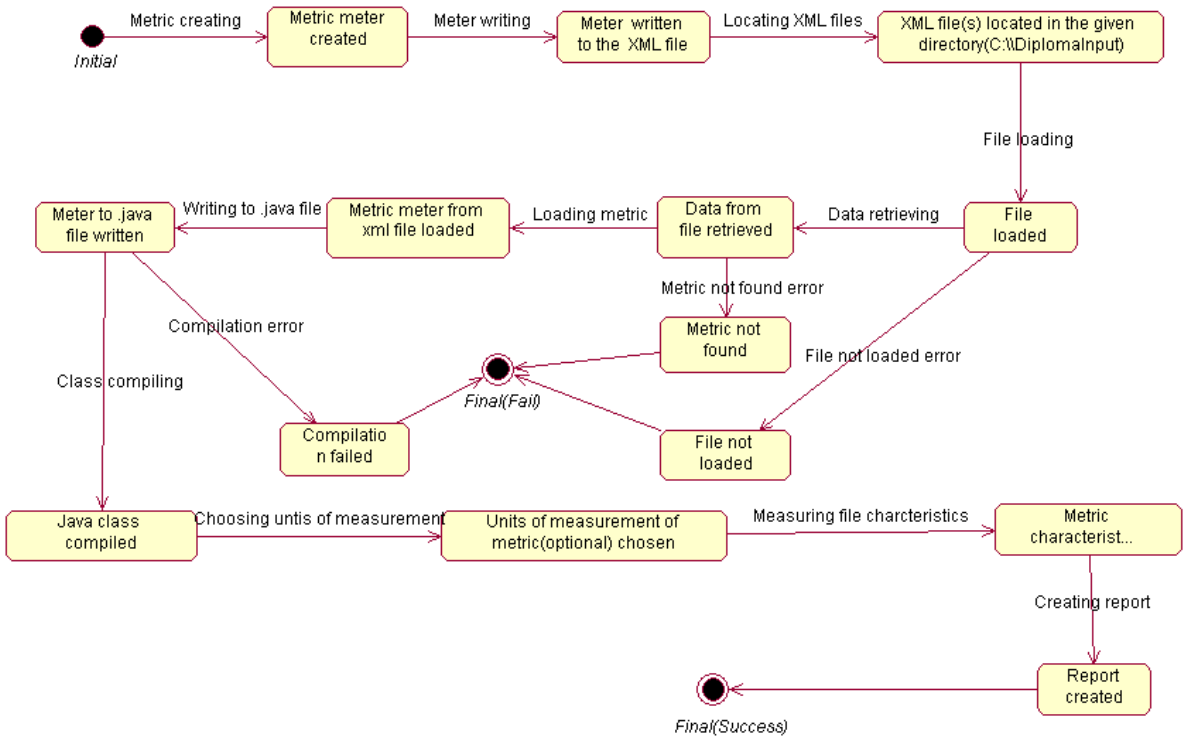

Рис. 1.4. Діаграма станів системи

Ця діаграма зображує перехід стану даної системи. Успішний варіант – це коли переходи всіх станів виконуються без помилок.

1.3. 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 Формальний опис метричних характеристик програмного забезпечення

Вимірювання завжди було фундаментальним для прогресу будь-якої інженерної дисципліни та програмного забезпечення. Показники програмного забезпечення використовувались для прийняття кількісних / якісних рішень, а також для оцінки ризиків та зменшення програмних проектів.

Вимірювання програмного забезпечення стало ключовим аспектом належної практики програмної інженерії. Діяльність з вимірювання додає вартості та забезпечує активну участь у всіх етапах процесу розвитку та інформування про них. Вимірювання може допомогти нам зробити конкретні характеристики наших процесів та продуктів більш помітними. Вимірювання охоплює кількісні оцінки, які зазвичай використовують метрики та міри, які можуть бути використані для безпосереднього визначення досягнення числових цілей якості. Незважаючи на всі ці досягнення та передбачувані переваги, вимірювання програмного забезпечення, схоже, не повністю проникло у виробничу практику. Що стосується використання програмного вимірювання для оцінки якості, ми помітили, що його застосування ще обмежене, оскільки більшість показників програмного забезпечення все ще використовуються в основному для оцінки витрат. Отже, зрозуміло, що все повинно бути вимірюваним. Показники програмного забезпечення використовуються для оцінки процесу розробки програмного забезпечення та якості отриманого продукту. Показники програмного забезпечення допомагають оцінити процес тестування та програмний продукт, надаючи об'єктивні критерії та вимірювання для прийняття управлінських рішень. Їх асоціація з ранніми виявленнями та виправленням проблем роблять їх важливими в програмному забезпеченні.

Метрики тестування програмного забезпечення виявили велику кількість застосувань під час тестування, такі як оцінка надійності, охоплення вихідного

коду, ефективність набору тестів тощо. Вони надають значущу та своєчасну інформацію, яка може допомогти нам вживати коригувальних заходів за необхідності. Ефективне впровадження показників може поліпшити якість програмного забезпечення та може допомогти нам доставити програмне забезпечення вчасно та в межах бюджету.

2.1.1. Вимірювання програмного забезпечення

Вимірювання – це відображення від емпіричного світу до формального, реляційного світу. Отже, міра – це число або символ, присвоєний сутності цим відображенням для характеристики атрибута. Це присвоєння цифр або символів будь-якій сутності здійснюється за однозначним правилом. Правилом призначення може бути будь-яке послідовне правило, за винятком випадкового призначення. Суб'єкт може бути об'єктом, таким як особа або специфікація програмного забезпечення, або подією, наприклад, організацією або кодованою програмою. Атрихют – це ознака чи властивість сутності, наприклад розмір (організації) або кількість рядків (кодованої програми).

Характеристиками хорошого вимірювання є:

– надійність – результат процесу вимірювання відтворюваний. (Подібні результати отримуються з часом та в різних ситуаціях.)

– дійсність – процес вимірювання фактично вимірює те, що передбачається виміряти.

– чутливість – процес вимірювання показує мінливість у відповідях, коли вона існує в стимулі чи ситуації.

Щодо програмного забезпечення, *Measurement* – це постійний процес визначення, збору та аналізу даних про процес розробки програмного забезпечення та його продуктів з метою розуміння та контролю процесу та його продуктів, а також надання значимої інформації для вдосконалення цього процесу та його продуктів. Відсутня можливість створити якісне програмне забезпечення або вдосконалити наш процес без вимірювань. Вимірювання є важливим для досягнення основних управлінських цілей прогнозування, прогресу та вдосконалення процесу.

Зазвичай програмне забезпечення вимірюється для таких цілей:

– характеристика, тобто збір інформації про якусь характеристику програмних процесів та продуктів, з метою отримання кращого уявлення про `` що відбувається ".

– оцінка, тобто оцінка деяких характеристик програмного процесу або продукту, наприклад, на основі історичних даних в тому ж середовищі розробки або даних, доступних із зовнішніх джерел.

– удосконалення, тобто використання причинно-наслідкового зв’язку для ідентифікації частин процесу або продукту, які можна змінити, щоб отримати позитивні наслідки для певної характеристики, що цікавить, та збору даних після того, як зміни було зроблено, щоб підтвердити чи не підтвердити, чи був ефект позитивним, та оцінити його ступінь.

– прогнозування, тобто виявлення причинно-наслідкових зв’язків між характеристиками продукту та процесу.

– відстеження, тобто (можливо постійне та регулярне) отримання інформації про деякі характеристики програмних процесів та продуктів з часом, щоб зрозуміти, чи контролюються ці характеристики в процесі проектів.

– перевірка, тобто перевірка визначених найкращих практик експериментально

2.1.2. Процес вимірювання програмного забезпечення

Процес вимірювання програмним забезпеченням повинен бути об'єктивним, упорядкованим методом кількісної оцінки, оцінки, коригування та остаточного вдосконалення процесу розробки. Дані збираються на основі відомих або передбачуваних проблем розвитку, проблем та питань. Потім вони аналізуються щодо процесу розробки програмного забезпечення та продуктів. Процес вимірювання використовується для оцінки якості, прогресу та ефективності на всіх фазах життєвого циклу.

Ключовими компонентами ефективного процесу вимірювання є:

– Чітко визначені проблеми розробки програмного забезпечення та міра (елементи даних), необхідні для забезпечення розуміння цих проблем;

– Обробка зібраних даних у графічні або табличні звіти (показники) для допомоги в аналізі проблем;

– Аналіз показників для забезпечення розуміння питань розвитку; і,

– Використання результатів аналізу для вдосконалення процесу та виявлення нових проблем та проблем (рис. 2.1).

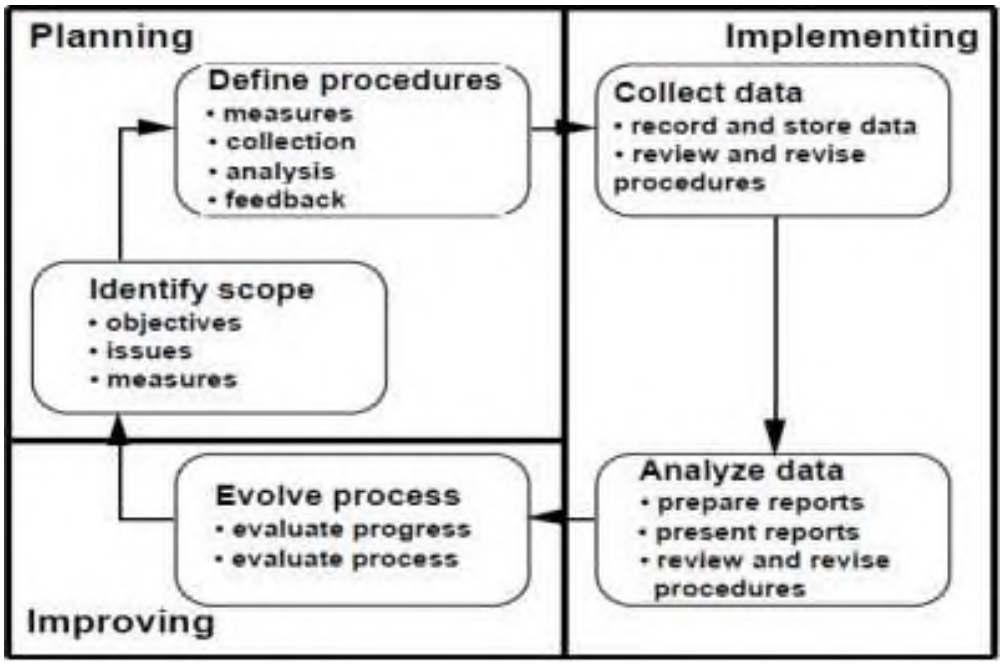

Рис. 2.1. Процес вимірювання програмного забезпечення

На рисунку 2.1 показано процес вимірювання програмного забезпечення. Діяльністю, необхідною для проектування процесу вимірювання з використанням цієї архітектури, є:

– Розробка процесу вимірювання, який повинен бути доступний як частина стандартного програмного процесу організації;

– Планування процесу за проектами та документування процедур шляхом адаптації та адаптації активу процесу;

– Здійснення процесу на проектах шляхом виконання планів та процедур;

– Удосконалення процесу шляхом вироблення планів та процедур у міру дозрівання проектів та зміни їх вимірювальних потреб.

2.1.3. Шкала вимірювання програмного забезпечення

Як правило, у теорії вимірювань використовують п’ять рівнів вимірювань, які застосовуються також до програмного забезпечення. Зазвичай, чим вужче розташування прийнятних змін, тим менша кількість шкал і тим інформативніша шкала. Вимірювальні шкали є ієрархічними, і кожна шкала рівня має всі властивості нижчих шкал, як показано нижче. Ми можемо перетворити шкали вищого рівня в нижчі (тобто відношення до інтервалу або порядкове чи номінальне; або інтервал до порядкового або номінального; або порядкове до номінального). Потужний аналіз може бути застосований до даних з більш інформативною шкалою вимірювань.

– Номінальна шкала .Це найбільш примітивна форма вимірювання. Шкала є номінальною, якщо вона ділить набір сутностей на категорії, без особливого впорядкування серед них.

– Порядкова шкала. Шкала є порядковою, якщо вона ділить набір сутностей на категорії, упорядковані за певним порядком. Кількісного порівняння немає.

– Інтервальна шкала. Ця шкала фіксує інформацію про розмір інтервалів, що розділяють класи. У інтервальній шкалі точна різниця між двома значеннями є основою для змістовних тверджень.

– Шкала співвідношення. Вимірювання, що зберігає порядок, розмір інтервалів між сутностями та співвідношення між сутностями. У шкалі співвідношень, на відміну від шкали інтервалів, існує абсолютна і не довільна нульова точка.

– Абсолютна шкала. Абсолютна шкала є найбільш інформативною в ієрархії шкали вимірювань. В абсолютному масштабі вимірювання проводиться методом підрахунку.

2.1.4. Проблеми з вимірюванням програмного забезпечення

Однозначне вимірювання є життєво важливим у життєвому циклі розробки програмного забезпечення. Повинні бути стандартизовані заходи, які можуть:

– Використовуватись як загальна базова лінія для вимірювання;

– Запропонуйте орієнтир для програмних вимірювачів для перевірки результатів їх вимірювання та здатності вимірювати той самий контрольний матеріал;

– Дозвольте вимірювачам використовувати відповідне контрольне поняття і, таким чином, говорити на одному рівні.

Однак вимірювання програмного забезпечення є складним за своєю суттю з наступних причин:

– Програмне забезпечення – це нематеріальний продукт. Це нетиповий продукт у порівнянні з іншими промисловими продуктами, оскільки він сильно варіюється щодо розміру, складності, техніки проектування, методів випробувань, застосовності.

– Існує мало консенсусу щодо конкретних показників програмних атрибутів, про що свідчить брак міжнародних стандартних заходів щодо програмних атрибутів, таких як складність та якість програмного забезпечення.

– Розробка програмного забезпечення настільки складна, що всі моделі є слабкими наближеннями, оскільки їх важко перевірити.

– Вимірювання властиві не всім проектам, організаціям. Заходи, які працюють для одного проекту, можуть не застосовуватися до іншого.

2.1.5. Метрика програмного забезпечення

Метрика – це кількісно вимірюваний програмний продукт, процес або проект, який безпосередньо спостерігається, обчислюється або передбачається. Метрики (або показники) обчислюються на основі вимірювань. Метрики полегшують кількісне визначення певної характеристики. Метрики – це вимірювання різних аспектів зусиль, які допомагають нам визначити, чи рухаємось ми до досягнення мети цієї діяльності. Метрики, як правило, спеціалізуються за предметною областю, і в цьому випадку вони діють лише в межах певного домену і не можуть бути безпосередньо порівняно оцінені або інтерпретовані поза ним. "По суті, показники програмного забезпечення займаються вимірюванням програмного продукту та процесом, за допомогою

якого він розробляється. Вони є кількісними показниками, що використовуються для порівняння програмних продуктів, процесів або проектів або для прогнозування їх результатів. Метрики програмного забезпечення повинні бути чітко визначені перед їх використанням; у таблиці 2.1 визначено елементи, які повинні бути визначені належним чином.

За допомогою програмних показників ми можемо:

– вимоги до монітора,

– прогнозувати ресурси розвитку,

– відстежувати прогрес у розвитку та

– визначення витрат на обслуговування.

2.1.6. Категорії метрик

Метрики програмного забезпечення можна класифікувати за різними категоріями, хоча однакові метрики можуть належати до кількох категорій. З комерційної точки зору, показники можна класифікувати на п’ять класів для вимірювання кількості та якості програмного забезпечення.

– технічні показникивикористовуються для визначення того, чи добре структурований код, адекватні посібники для апаратного та програмного забезпечення, що документація є повною, правильною та актуальною. Технічні показники також описують зовнішні характеристики впровадження системи.

– метрики дефектів використовуються для того, щоб визначити, що система помилково не обробляє дані, не аномально припиняє роботу та не робить багатьох інших речей, пов’язаних з несправністю програмної системи.

– показники задоволеності кінцевих користувачів використовуються для опису значення, отриманого від використання системи.

– показники гарантії відображають конкретні доходи та витрати, пов'язані з виправленням дефектів програмного забезпечення для кожного конкретного випадку. На ці показники впливає рівень дефектів, готовність користувачів висувати скарги та готовність та здатність розробника програмного забезпечення пристосувати користувача.

– метрики репутації використовуються для оцінки сприйняття задоволеності користувачів програмним забезпеченням і можуть генерувати найбільшу цінність, оскільки вони можуть сильно впливати на придбане програмне забезпечення. Репутація може суттєво відрізнятися від фактичного задоволення:

1) оскільки окремі користувачі можуть використовувати лише незначну частину функцій, передбачених будь-яким програмним пакетом; і

2) оскільки маркетинг та реклама часто впливають на сприйняття покупцем якості програмного забезпечення більше, ніж на його реальне використання.

З точки зору значимості Метрики можна згрупувати у два класи:

– основна метрика – це необхідна метрика, яка є важливою для підтримки управління тестами доставки рішень у проектах з розробки систем. Приклад: Відсоток виконаних вимог.

– неосновна метрика – необов'язкова метрика, яка може допомогти створити більш збалансовану картину якості та ефективності тестових зусиль. Приклад: Загальна кількість дефектів за фазою випробування.

З точки зору спостереження, Метрики також можна класифікувати як:

– Примітивні метрики – це ті, які можна спостерігати безпосередньо, наприклад, розмір програми (у *LOC*), кількість дефектів, що спостерігаються під час модульного тестування, або загальний час розробки проекту.

– Обчислювані метрики – це ті, які неможливо спостерігати безпосередньо, але певним чином обчислюються з інших метрик. Прикладами обчислюваних показників є ті, які зазвичай використовуються для підвищення продуктивності, такі як *LOC*, вироблений за людину-місяць (*LOC* / людина-місяць), або для якості продукції, наприклад, кількість дефектів на тисячу рядків коду (дефекти / *KLOC*). Обчислювані метрики – це комбінації інших метричних значень і, отже, часто є більш цінними для розуміння або оцінки програмного процесу, ніж прості метрики.

З точки зору вимірювання Метрики можна класифікувати як:

– пряме вимірювання атрибута суб'єкта господарювання не передбачає жодного іншого атрибута чи сутності. Пряме вимірювання – це оцінка чогось

існуючого. Наприклад, кількість рядків коду. Непряме / похідне вимірювання означає обчислення за участю інших атрибутів або сутностей за допомогою якоїсь математичної моделі (завжди містить обчислення принаймні двох показників). Наприклад, щільність дефекту = ні. дефектів програмного продукту / загальний розмір продукту;

– система прогнозування складається з математичної моделі разом із набором процедур прогнозування для визначення невідомих параметрів та інтерпретації результатів. Наприклад, якість програмного забезпечення.

Частіше показники програмного забезпечення класифікуються у набагато ширшому розумінні як:

– показники процесу – це показники процесу розробки програмного забезпечення, такі як загальний час розробки, тип використовуваної методології або середній рівень досвіду програмістів. Їх можна класифікувати як емпіричні, статистичні, теоретичні та композитні моделі.

– показники продукту – це показники програмного продукту на будь-якому етапі його розвитку, від вимог до встановленої системи. Метрики продукту можуть вимірювати складність дизайну програмного забезпечення, розмір кінцевої програми (вихідний або об’єктний код) або кількість сторінок документації, що виготовляється. Вони часто класифікуються відповідно до розміру, складності, якості та залежності даних.

Показники програмного забезпечення також можна класифікувати як:

– об'єктивні метрики завжди повинні призводити до однакових значень для даної метрики, виміряних двома або більше кваліфікованими спостерігачами.

– суб'єктивні метрики можуть вимірювати різні значення для даної метрики, оскільки їх суб'єктивне судження бере участь у досягненні виміряного значення. Що стосується показників продукту, розмір товару, виміряний у рядках коду (*LOC*), є об'єктивним показником, для якого будь-який поінформований спостерігач, який працює за однаковим визначенням *LOC*, має однакову виміряну величину для даної програми. Прикладом суб'єктивної метрики продукту є класифікація програмного забезпечення як "органічне",

"напіввід'єднане" або "вбудоване", як це вимагається в моделі оцінки витрат *COCOMO*.

Незважаючи на те, що програмні показники можна акуратно класифікувати як примітивні об'єктивні показники товару, примітивні суб'єктивні показники продукту тощо, цей модуль не суворо дотримується цієї організації.

2.1.7. Характеристика корисних метрик

Метрики слід збирати не тому, що вони прописані в літературі або тому, що вони визнані популярними в деяких компаніях, а тому, що вони корисні для прийняття рішень щодо конкретного проекту або в межах певної організації. Корисна метрика є точно визначеною (тобто вимірюваною чи кількісною), вона також допомагає вказати, чи досягає організація цілей програмного забезпечення. Існує кілька основних характеристик, пов’язаних із корисними показниками програмного забезпечення.

– простий і зрозумілий

– вимірюваний;

– економічний;

– показники повинні бути своєчасними;

– міцний;

– надійний та дійсний;

– послідовні та використовувані з часом;

– ненав’язливо зібрані;

– незалежний;

– підзвітний;

– точний.

Корисні показники повинні супроводжуватися даними, які є правильними (правильними згідно з правилами визначення метрики), точними, точними та послідовними (велика різниця у величині не виникає, навіть якщо особа або вимірювальний прилад змінюються). Процедура вимірювання повинна бути чітко описана досить чітко, щоб хтось інший міг повторити вимірювання.

2.1.8. Метрика тестування програмного забезпечення

Термін, що використовується для опису вимірювання певного атрибута програмного проекту, – це програмна метрика. Метрики програмного забезпечення, які виробляє команда контролю якості, стосуються тестової діяльності, яка є частиною фази тестування, і тому офіційно відома як Метрика тестування програмного забезпечення.

Процес випробуванняметрики надають інформацію про підготовку до тестування, виконання тесту та хід тесту. Вони використовуються для моніторингу прогресу тестування, статусу розробки та розробки тестових кейсів та результатів тестових кейсів після виконання. Вони не надають інформації про стан випробування продукту і в основному використовуються для вимірювання прогресу фази випробування. Показники процесів описують ефективність та якість процесів, що виробляють програмний продукт. Прикладами є зусилля, необхідні в процесі, час на виготовлення продукту, ефективність усунення дефектів під час розробки, кількість дефектів, виявлених під час випробувань, зрілість процесу. Деякі показники процесу тестування:

– кількість розроблених тестових кейсів;

– кількість виконаних тестових справ;

– % виконаних тестових справ;

– % пройдених тестових випадків;

– % тестів не вдалося;

– загальний фактичний час виконання / загальний передбачуваний час виконання;

– середній час виконання тесту.

Випробувальний продуктметрики надають інформацію про стан тестування та стан тестування програмного продукту та генеруються під час виконання тесту та виправлення коду або відстрочки. Дані для таких показників також генеруються під час тестування і можуть допомогти нам дізнатися якість продукту. Використовуючи ці показники, ми можемо виміряти стан випробувань продукції та орієнтовний рівень якості, корисний для прийняття рішень про

випуск продукції. Метрики товару описують такі характеристики товару, як розмір, складність, конструктивні особливості, продуктивність, ефективність, надійність, портативність тощо. Деякі тестові метрики товару:

– розрахунковий час для тестування;

– фактичний час тестування;

– % витраченого часу = (Фактичний витрачений час / передбачуваний час тестування) · 100;

– середній інтервал часу між несправностями;

– максимальна та мінімальна кількість відмов, що виникають за будь-який проміжок часу;

– середня кількість несправностей, що виникають через проміжки часу;

– залишився час для завершення тестування.

2.1.9. Важливість метрики тестування програмного забезпечення

Показники програмного забезпечення застосовуються до всього життєвого циклу розробки від початку, коли слід оцінювати витрати до контролю надійності кінцевого продукту на місцях, і того, як продукт змінюється з часом із вдосконаленням. Показники програмного забезпечення мають важливе значення на етапі тестування, оскільки показники тестування програмного забезпечення виступають як показники якості програмного забезпечення та схильності до несправностей.

Метрики тестування демонструють тенденції та характеристики з часом, що свідчить про стабільність процесу. Основним етапом встановлення тестових показників є визначення ключових процесів тестування програмного забезпечення, які можна об’єктивно виміряти. Вимірювання проектів розробки програмного забезпечення та тестування є складною, але важливою складовою професійної організації.

Проект програмного забезпечення може працювати з часом і надмірно, але все ще має велику кількість дефектів. Або, це може бути вчасно і за бюджетом і мати ще більшу кількість дефектів. Вимірювання дозволяє кількісно визначити ваш графік, розробку та тестування. Коли ви вимірюєте поточну ефективність

проекту, ви стаєте краще підготовленим для планування та складання бюджету для майбутніх проектів. Основна частка програмних проектів страждає від проблем якості, що, в свою чергу, вимагає нових показників тестування для ефективного вимірювання тестових процесів. Тестові показники – це потужний інструмент управління ризиками, який допомагає нам виміряти поточну ефективність. Тестові показники є ключовими «фактами» і слугують наступним цілям: Основна частка програмних проектів страждає від проблем якості, що, в свою чергу, вимагає нових показників тестування для ефективного вимірювання тестових процесів. Тестові показники – це потужний інструмент управління ризиками, який допомагає нам виміряти поточну ефективність. Тестові показники є ключовими «фактами» і слугують наступним цілям: Основна частка програмних проектів страждає від проблем якості, що, в свою чергу, вимагає нових показників тестування для ефективного вимірювання тестових процесів.

Тестові показники – це потужний інструмент управління ризиками, який допомагає нам виміряти поточну ефективність. Тестові показники є ключовими «фактами» і слугують наступним цілям:

– допомагає зрозуміти поточну позицію проекту.

– допомагає визначити пріоритети діяльності, щоб зменшити ризик перевитрат розкладу на випуски програмного забезпечення

– забезпечує основу для оцінки та полегшує планування подолання розриву в ефективності.

– забезпечує засоби для контролю / звітування про стан.

– визначте зони ризику, які потребують додаткового тестування.

– швидко визначає та допомагає вирішити потенційні проблеми та визначає напрямки вдосконалення.

– тестові показники забезпечують об’єктивний показник ефективності та результативності тестування.

– встановлення еталонів якості для кількох завдань та процесів, що беруть участь у розробці.

2.1.10. Помітна метрика тестування програмного забезпечення

Показники тестування можуть допомогти нам виміряти поточну ефективність будь-якого проекту. Зібрані дані можуть стати історичними даними для майбутніх проектів. Ці дані дуже важливі, оскільки за відсутності історичних даних усі оцінки – це лише здогадки. Отже, важливо записати ключову інформацію про поточні проекти. Тестові показники можуть стати важливим показником ефективності та результативності процесу тестування програмного забезпечення, а також можуть виявити ризикові сфери, які можуть потребувати додаткового тестування. Показники, пов'язані з тестуванням програмного забезпечення, розбиті на три категорії:

– висвітлення: Значущі параметри для вимірювання обсягу тесту та успішності;

– прогрес: параметри, які допомагають визначити прогрес тесту, який слід узгодити з критеріями успіху. Показники прогресу збираються ітеративно з часом. Вони можуть бути використані для графіку самого процесу (наприклад, час виправлення дефектів, час тестування тощо);

– якість: значущі показники досконалості, вартості, вартості тощо продукту, що тестується. Важко безпосередньо виміряти якість; однак виміряти ефекти якості простіше і можливо.

2.2 Оцінка метрики вихідного коду означає функціональні вимоги

При розробці програми слід дотримуватися наступних вимог:

1. Назва метрики. Кожен показник повинен мати унікальну і зрозумілу назву;

2. Опис. Кожен показник повинен мати короткий опис;

3. Граничні умови. Числа, які порівнюються з результатом, якщо аналізувати вихідний код;

4. Виділення. Результати вимірювання порівнюються з межами, а високі – з червоним або зеленим кольором, що означає, що підходить до меж чи ні відповідно;

5. Вибір мови. Програма мусить бути багатомовною;

6. Розширюваність. Програма повинна розширюватися, що означає можливість завантаження метрик до існуючої програми.

Як показано на діаграмі використання (Рис. 2.2). Користувач може створювати метрики та додавати їх до програми. Під час запуску програми користувач може виконати деякі дії, маніпулюючи нею, щоб отримати розраховані показники.

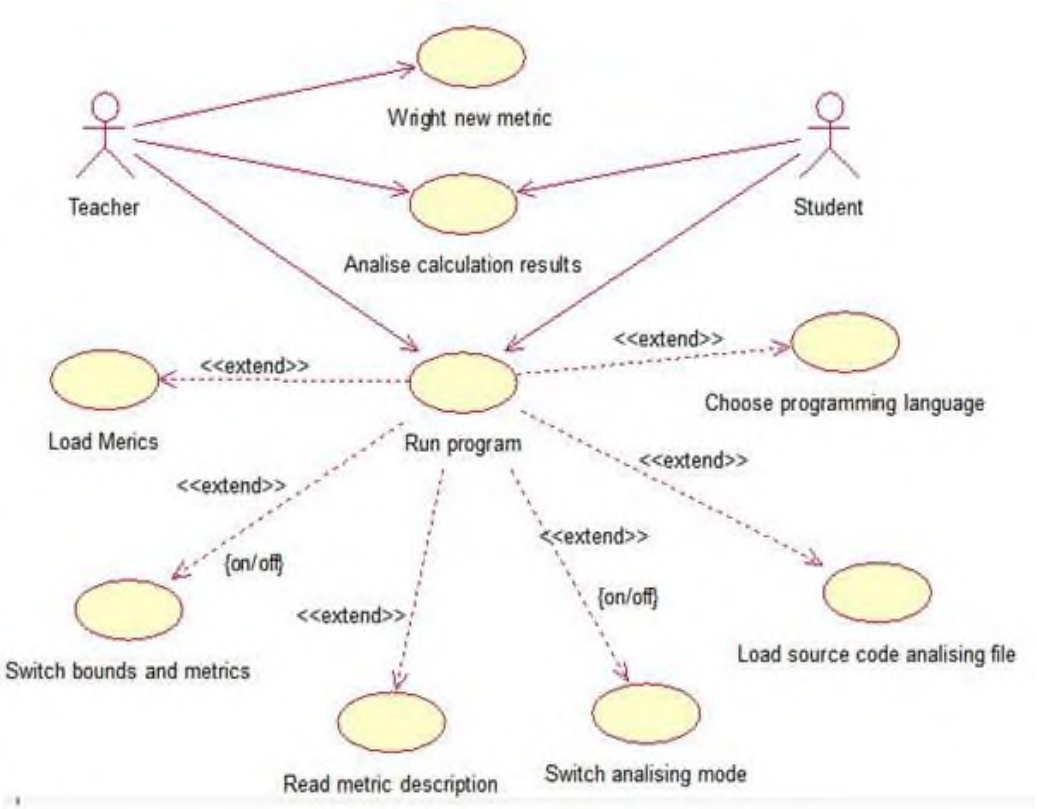

Рис. 2.2. Діаграма випадків використання програми.

2.3. Вимірювання надійності загальної архітектури ПЗ

Розроблена програма повинна вимірювати метричні характеристики вихідного коду. Програма повинна розширюватися, це означає, що кожен може писати метрику та додавати її в інтерфейс програм. Ось чому в програму буде

вбудовано мало метрик, а решту з них можна завантажувати під час роботи програми.

Мета програм – використовуватись у навчанні. Буде можливість встановити межі елементів метрик, а також покаже, чи відповідає вихідний код необхідним межам, виділивши червоним та зеленим кольорами. Також буде можливість вимкнути режим навчання та певні показники.

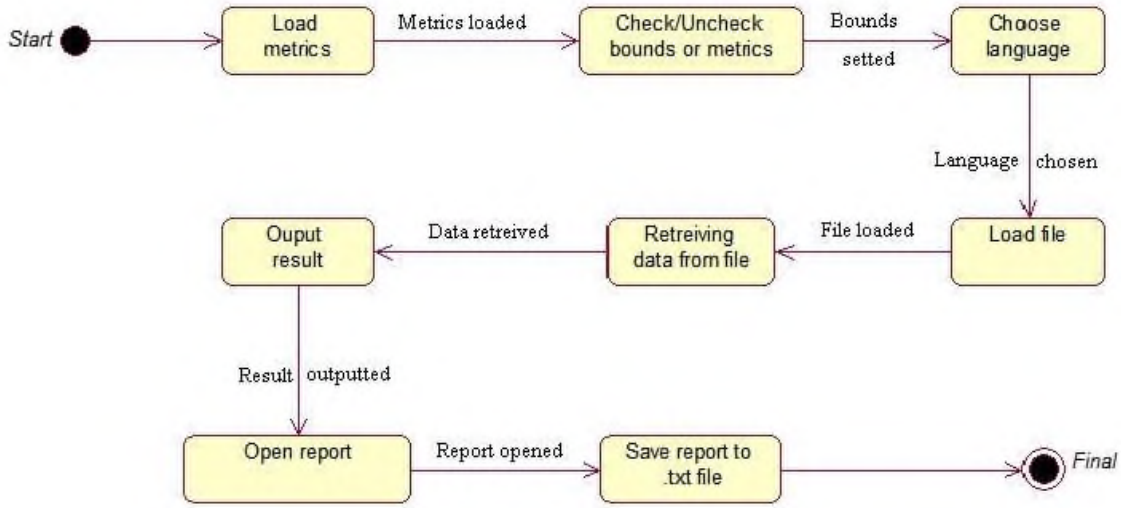

Рис. 2.3. Схема діаграми стану програми

Діаграма стану показує етапи дій, які маніпулюють програмою. Першим етапом є завантаження метрик, потім вибрати межі або зняти їх, якщо немає потреби. Виберіть мову проаналізованого вихідного коду, а потім завантажте файл. Після вибору файл буде проаналізований, і програма видасть результат, який можна переглянути у звіті та зберегти у текстовому файлі.

2.4. Загальна архітектура вимірювального інструменту

В результаті проектування отримана схема класу:

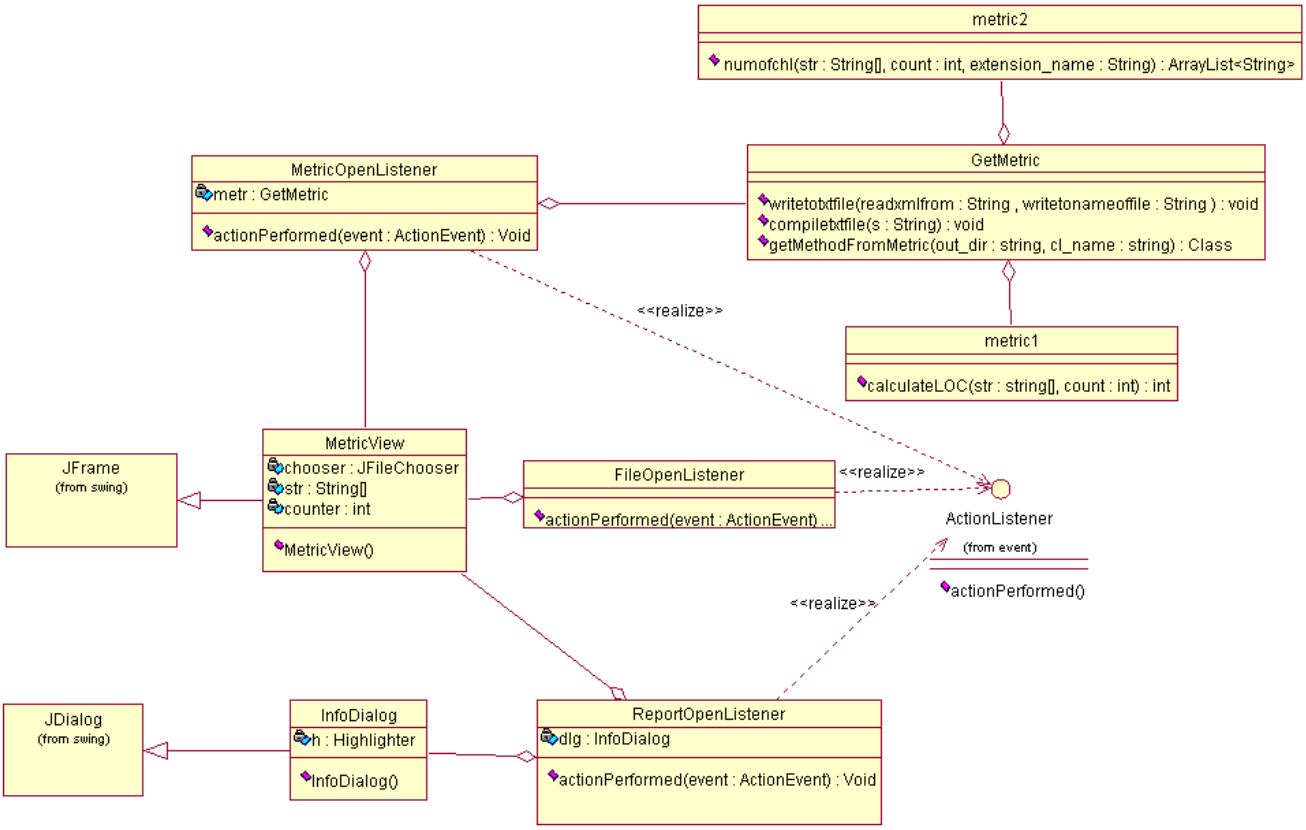

Рис. 2.4. Діаграм класів системи

У таблицях 2.1-2.10 будуть наведені короткі характеристики класів дизайну:

Таблиця 2.1

Властивості дизайнерського класу *JFrame*

Властивості дизайнерського класу	Опис
Ім'я	*JFrame*
Короткий опис	Батьківський клас *MetricView*.
Обов'язки	Розширена версія *java.awt.Frame*, яка додає підтримку архітектури компонента *JFC / Swing*.
Відносини	Узагальнення, де *JFrame* – суперклас, а *MetricView* – підклас.
Операції	захищена порожнеча *addImpl* (*Component comp*, *Object constraints*, *int index*) – Додає вказаний дочірній компонент. захищений *JRootPane createRootPane* () – Викликається методами конструктора для створення *rootPane* за замовчуванням. тощо

Таблиця 2.2

Властивості дизайнерського класу *MetricView*

Властивості дизайнерського класу	Опис
Ім'я	*MetricView*
Короткий опис	Основна рамка програми.
Обов'язки	Представляє основний фрейм, до якого додаються всі інші компоненти.
Відносини	Узагальнення, де *JFrame* – суперклас, а *MetricView* – підклас. Агрегація, де *MetricOpenListener* є сукупністю, а *MetricView* – частиною. Агрегація, де *FileOpenListener* є сукупністю, а *MetricView* – частиною. Агрегація, де *ReportOpenListener* є сукупністю, а *MetricView* – частиною.
Операції	*MetricView* () – конструктор, який додає всі необхідні компоненти.
Атрибути	Вибір *JFilechooser* – *JFileChooser* забезпечує простий механізм вибору користувачем файлу; *string* [] *str* – масив, де зберігаються дані з файлу; *Int* лічильник – підраховує кількість значень масиву; Ім'я рядка – ім'я файлу зберігається.

Таблиця 2.3

Властивості дизайнерського класу *JDialog*

Властивості дизайнерського класу	Опис
Ім'я	*JDialog*
Короткий опис	Батьківський клас *InfoDialog*.
Обов'язки	Основний клас для створення діалогового вікна. Цей клас можна використовувати для створення власного діалогового вікна або викликати безліч методів класу в *JOptionPane* для створення різноманітних стандартних діалогових вікон.
Відносини	Узагальнення, де *JDialog* є суперкласом, а *InfoDialog* – підкласом.
Операції	захищена порожнеча *dialogInit* () – Викликаний конструкторами належним чином ініціювати *JDialog*. *AccessibleContext getAccessibleContext* () – Отримує *AccessibleContext*, пов'язаний з цим *JDialog*.

Таблиця 2.4

Властивості дизайнерського класу *InfoDialog*

Властивості дизайнерського класу	Опис
Ім'я	*InfoDialog*
Короткий опис	Діалогове вікно, відповідальне за створення звіту.
Обов'язки	Коли в основному кадрі натискається кнопка “Показати звіт”, з’являється *InfoDialog*, що представляє результати оцінки показників та звіту.
Відносини	Узагальнення, де *JDialog* є суперкласом, а *InfoDialog* – підкласом. Агрегація, де *ReportOpenListener* є сукупністю, а *InfoDialog* – частиною.
Операції	*InfoDialog* () – конструктор, який використовується для створення діалогового вікна.
Атрибути	Хайлайтер *h* – інтерфейс для об’єкта, який дозволяє розмітити фон кольоровими областями.

Таблиця 2.5

Властивості дизайнерського класу *MetricOpenListener*

Властивості дизайнерського класу	Опис
Ім'я	*MetricOpenListener*
Короткий опис	Клас реагує на кнопку "Завантажити показники"
Обов'язки	Відповідає за отримання об’єкта метричного класу за допомогою методів класу *GetMetric*.
Відносини	Агрегація, де *MetricOpenListener* є сукупністю, а *MetricView* – частиною. Агрегація, де *MetricOpenListener* є сукупністю, а *GetMetric* – частиною. “Реалізувати” залежність там, де *MetricOpenListener* реалізує інтерфейс *ActionListener*
Операції	Публічна порожнеча *ActionPerformed* (*a*подія *ctionEvent*) – Метод, оголошений в інтерфейсі *ActionListener*.
Атрибути	*GetMetric* метр – Об'єкт класу *GetMetric*, необхідний для виклику функцій цього класу.

Таблиця 2.6

Властивості дизайнерського класу *ReportOpenListener*

Властивості дизайнерського класу	Опис
Ім'я	*ReportOpenListener*
Короткий опис	Клас реагує на кнопку "Показати звіт"
Обов'язки	Відповідає за створення діалогового вікна з необхідною інформацією.
Відносини	Агрегація, де *ReportOpenListener* є сукупністю, а *MetricView* – частиною. Агрегація, де *ReportOpenListener* є сукупністю, а *InfoDialog* – частиною. “Реалізувати” залежність, де *ReportOpenListener* реалізує інтерфейс *ActionListener*
Операції	Публічна порожнеча *ActionPerformed* (*a*подія *ctionEvent*) Метод, оголошений в інтерфейсі *ActionListener*.
Атрибути	*InfoDialog dlg* – Об'єкт класу *InfoDialog*.

Таблиця 2.7

Властивості дизайнерського класу *FileOpenListener*

Властивості дизайнерського класу	Опис
Ім'я	*FileOpenListener*
Короткий опис	Клас реагує на кнопку «Завантажити файл»
Обов'язки	Він відповідає за завантаження файлу в систему та отримання даних, знайдених у ній.
Відносини	Агрегація, де *FileOpenListener* є сукупністю, а *MetricView* – частиною. “Реалізувати” залежність, де *FileOpenListener* реалізує інтерфейс *ActionListener*
Операції	*ActionPerformed* (*a*подія *ctionEvent*) – Метод, оголошений в інтерфейсі *ActionListener*.

Таблиця 2.8

Властивості дизайнерського класу *GetMetric*

Властивості дизайнерського класу	Опис
Ім'я	*GetMetric*
Короткий опис	Клас, який допомагає завантажувати метрики в систему.
Обов'язки	Він відповідає за маніпулювання файлами *xml*, в яких зберігаються метрики.
Відносини	Агрегація, де *GetMetric* – це сукупність, а *metric*1 – частина. Агрегація, де *GetMetric* – це сукупність, а *metric*2 – частина. Агрегація, де *MetricOpenListener* є сукупністю, а *GetMetric* – частиною.
Операції	громадськості *void writeetotxtfile* (рядок *readxmlfrom*, рядок *writeetonameoffile*) – метод, відповідальний за читання коду з файлу *xml* та запис його у файл *txt*. *void compiletxtfile* (*String s*) – метод, який динамічно компілює клас *Java*, що зберігається у файлі *txt*. Клас *getMethodFromMetric* (*String outputdirectory, String classname*) – допомагає отримати методи з метричного класу.

Таблиця 2.9

Властивості дизайнерського класу *Metric*1

Властивості дизайнерського класу	Опис
Ім'я	*Metric*1
Короткий опис	Клас, що містить перший метричний код.
Обов'язки	Відповідає за метричні вимірювання.
Відносини	Агрегація, де *GetMetric* – це сукупність, а *metric*1 – частина.
Операції	*public int calcuLOC* (*String* [] *str, int i*) – імовірно метод, що використовується для обчислення метрики *LOC*.

Таблиця 2.10

Властивості дизайнерського класу *Metric*2

Властивості дизайнерського класу	Опис
Ім'я	*Metric*2
Короткий опис	Клас, що містить другий метричний код.
Обов'язки	Відповідає за метричні вимірювання.
Відносини	Агрегація, де *GetMetric* – це сукупність, а *metric*2 – частина.
Операції	*upublic ArrayList <String> numofchl* (*String s* [], *int count, String extension_name*) – імовірно метод, що використовується для обчислення метрики *NOC*.

Єдиний інтерфейс, який однозначно буде використаний у процесі створення системи, представлений у таблицы 2.11.

Таблиця 2.11

Властивості створеного інтерфейсу метрики *ActionListener*

Властивості інтерфейсу	Опис
Ім'я	*ActionListener*
Короткий опис	Інтерфейс, що містить метод, що виконується.
Обов'язки	Інтерфейс слухача для отримання подій дій. Клас, який зацікавлений в обробці події дії, реалізує цей інтерфейс, і об'єкт, створений за допомогою цього класу, реєструється в компоненті, використовуючи метод *addActionListener* компонента. Коли відбувається подія дії, викликається метод *actionPerformed* цього об'єкта.
Відносини	“Реалізувати” залежність, де *FileOpenListener* реалізує інтерфейс *ActionListener* “Реалізувати” залежність там, де *MetricOpenListener* реалізує інтерфейс *ActionListener* “Реалізувати” залежність, де *ReportOpenListener* реалізує інтерфейс *ActionListener*
Операції	*void actionPerformed* (*ActionEvent e*) – викликається, коли відбувається дія.

Під якісним програмним забезпеченням розуміється програмне забезпечення, яке не містить помилок або дефектів, доставлено вчасно і в рамках встановленого бюджету, відповідає вимогам і / або очікуванням і з можливістю коригування. У контексті розробки програмного забезпечення якість програмного забезпечення відображає як функціональне якість, так і структурний якість.

Функціональна якість програмного забезпечення відображає, наскільки добре воно відповідає заданому дизайну, на підставі функціональних вимог або специфікацій.

Структурна якість програмного забезпечення стосується обробки не функціональних вимог, які підтримують виконання функціональних вимог, таких як надійність або ремонтопридатність, і ступеня, в якій програмне забезпечення було вироблено правильно.

Software Quality Assurance (*SQA*) – це комплекс заходів, спрямованих на забезпечення якості процесів розробки програмного забезпечення, які в кінцевому підсумку призводять до якісних програмних продуктів. Діяльність встановлює і оцінює процеси, які виробляють продукти. Це включає процесно-орієнтовані дії.

Контроль якості програмного забезпечення- Контроль якості програмного забезпечення (*SQC*) – це комплекс заходів щодо забезпечення якості програмних продуктів. Ці дії спрямовані на виявлення дефектів у фактичній продукції. Це включає в себе дії, орієнтовані на продукт.

Проблема якості програмного забезпечення

В індустрії програмного забезпечення розробники ніколи не заявлять, що програмне забезпечення не має дефектів, на відміну від інших виробників промислової продукції. Ця різниця обумовлена наступними причинами.

Складність продукту

Це кількість режимів роботи, які допускає продукт. Зазвичай промисловий продукт допускає тільки кілька тисяч режимів роботи з різними комбінаціями настроювань машини. Проте, програмні пакети надають мільйони операційних

можливостей. Отже, забезпечення всіх цих операційних можливостей є серйозною проблемою для індустрії програмного забезпечення.

Модель Фактора Маккола

Ця модель класифікує всі вимоги до програмного забезпечення по 11 показникам якості програмного забезпечення. Ці 11 факторів згруповані в три категорії – експлуатація продукту, перегляд продукту і фактори переходу продукту.

Фактори роботи продукту- правильність, надійність, ефективність, цілісність, зручність використання.

Фактори перегляду продукту- ремонтопридатність, гнучкість, тестованих.

Фактори переходу продукту – переносимость, можливість повторного використання, сумісність.

Фактори якості програмного забезпечення роботи продукту

Згідно з моделлю Макколла, категорія роботи продукту включає п'ять факторів якості програмного забезпечення, які відповідають вимогам, які безпосередньо впливають на щоденну роботу програмного забезпечення. Вони полягають в наступному:

Правильність.

Ці вимоги стосуються правильності виведення програмного забезпечення системи. Вони включають в себе:

– вихідну місію;

– необхідна точність виведення, на яку можуть негативно вплинути неточні дані або неточні розрахунки;

– повнота виведеної інформації, на яку можуть вплинути неповні дані;

– актуальність інформації, яка визначається як час між подією і відповіддю системи програмного забезпечення;

– доступність інформації;

– стандарти кодування і документування програмного забезпечення системи;

– надійність.

Вимоги до надійності пов'язані з відмовою обслуговування. Вони визначають максимально допустиму частоту відмов програмної системи і можуть ставитися до всієї системи або до однієї або декількох її окремих функцій.

ККД Стосується апаратних ресурсів, необхідних для виконання різних функцій системи програмного забезпечення. Він включає в себе можливості обробки (в *МГц*), ємність зберігання (в *МБ* або *ГБ*) і можливість передачі даних (в МБПС або ГБПС).

Він також стосується часу між підзарядкою портативних блоків системи, таких як блоки інформаційної системи, розташовані в портативних комп'ютерах, або метеорологічні блоки, розміщені на відкритому повітрі.

Цілісність. Цей фактор пов'язаний з безпекою системи програмного забезпечення, тобто для запобігання доступу сторонніх осіб, а також для розмежування групи людей, яким даються дозволу на читання і запис.

Юзабіліті. Вимоги юзабіліті пов'язані з кадровими ресурсами, необхідними для навчання нового співробітника і експлуатації системи програмного забезпечення.

Фактори якості редакції продукту. Згідно з моделлю Маккола, три категорії якості програмного забезпечення включені в категорію редакції продукту. Ці чинники полягають в наступному.

Ремонтопридатність. Цей фактор враховує зусилля, які будуть потрібні користувачам і обслуговуючому персоналу для визначення причин збоїв програмного забезпечення, виправлення збоїв і перевірки успішності виправлень.

Гнучкість. Цей фактор пов'язаний з можливостями і зусиллями, необхідними для підтримки адаптивного обслуговування програмного забезпечення. Це включає в себе адаптацію поточного програмного забезпечення до додаткових обставин і клієнтам без зміни програмного забезпечення. Вимоги цього фактора також підтримують вчинені дії з обслуговування, такі як зміни і доповнення програмного забезпечення, щоб поліпшити його обслуговування і адаптувати його до змін в технічній або комерційної середовищі фірми.

Виявлення несправностей. Вимоги до тестованості відносяться як до тестування програмної системи, так і до її роботи. Він включає попередньо певні проміжні результати, файли журналу, а також автоматичну діагностику, виконувану програмної системою перед запуском системи, щоб з'ясувати, чи всі компоненти системи знаходяться в робочому стані, і отримати звіт про виявлені несправності. Інший тип цих вимог стосується автоматичних діагностичних перевірок, які застосовуються фахівцями з технічного обслуговування для виявлення причин збоїв програмного забезпечення.

Фактор якості програмного забезпечення для переходу продукту. Згідно з моделлю Макколла, три категорії якості програмного забезпечення включені в категорію переходу продукту, яка стосується адаптації програмного забезпечення до інших середах і його взаємодії з іншими програмними системами. Ці чинники полягають в наступному

Портативність. Вимоги переносимості мають тенденцію до адаптації програмної системи до інших середовищ, що складається з іншого обладнання, різних операційних систем і так далі. Програмне забезпечення повинно мати можливість продовжувати використовувати один і той же базове програмне забезпечення в різних ситуаціях.

Повторне використання. Цей фактор пов'язаний з використанням програмних модулів, спочатку розроблених для одного проекту, в новому проекті Програми, який розробляється в даний час. Вони можуть також дозволити майбутнім проектам використовувати даний модуль або групу модулів розробленого в даний час програмного забезпечення. Очікується, що повторне використання програмного забезпечення дозволить заощадити ресурси розробки, скоротити період розробки і забезпечити більш якісні модулі.

Сумісність. Вимоги до функціональної сумісності спрямовані на створення інтерфейсів з іншими програмними системами або з іншими апаратними засобами обладнання. Наприклад, вбудоване програмне забезпечення виробничого обладнання та випробувального обладнання взаємодіє з програмним забезпеченням управління виробництвом.

SQA Компоненти. *Software Quality Assurance*(*SQA*) – гарантує, що розроблене програмне забезпечення відповідає і відповідає певним або стандартизованим специфікаціям якості. *SQA* – це безперервний процес в рамках життєвого циклу розробки програмного забезпечення (*SDLC*), який регулярно перевіряє розроблене програмне забезпечення, щоб переконатися, що воно відповідає необхідним показникам якості.

Практики *SQA* застосовуються в більшості типів розробки програмного забезпечення незалежно від використовуваної моделі розробки програмного забезпечення. *SQA* включає і впроваджує методології тестування програмного забезпечення для тестування програмного забезпечення. Замість перевірки якості після завершення, *SQA* обробляє тест на якість на кожному етапі розробки, поки програмне забезпечення не буде завершено. З *SQA* процес розробки програмного забезпечення переходить в наступну фазу тільки тоді, коли поточна / попередня фаза відповідає необхідним стандартам якості. *SQA* зазвичай працює над одним або декількома галузевими стандартами, які допомагають в розробці рекомендацій по якості програмного забезпечення і стратегій реалізації.

Можна показати лінійки програмних продуктів, як найбільш вдалий підхід до внутрішньоорганізаційного повторного використання програмного забезпечення. Численні компанії, які значно вдосконалили свою ефективність науково-дослідних робіт (досліджень і розробок), збільшивши портфель своїх продуктів на замовлення, забезпечують стабільний досвід користувачів за допомогою представлення власних продуктів та пропонованих рівнів налаштованості своїх продуктів, яких можна досягти лише за допомогою спільних компоненти програмного забезпечення з відповідною мінливістю програмного забезпечення. Через ці переваги лінійка програмних рішень має основний вплив на бізнес компаній, які намагаються успішно застосовувати технології, тобто, з точки зору бізнесу, успішна лінійка продуктів забезпечує наступну "криву *S*" зростання для компанії.

Є, принаймні, дві причини, чому компанія була б зацікавлена у переході до програмних екосистем. Спочатку компанія може зрозуміти, що рівень

функціональності, який слід розробляти для задоволення потреб своїх клієнтів, набагато більше, ніж той, який можна створити за розумний проміжок часу та за допомогою інвестицій у НДДКР, які дають відчутний дохід.

Для компаній, що надають веб-послуги так само, як і для галузі програмного забезпечення в цілому, ринок часто працює за принципом «переможець бере всіх». Отже, створення на базі великого клієнта настільки швидко, наскільки це можливо, є ключовою стратегією довгострокового успіху.

По-друге, масова тенденція до регулювання контролю потребує необхідних інвестицій у НДДКР для успішного застосування. Користувачі вимагають необхідного рівня налаштування або навіть послідовності, що дозволяє кожному користувачеві створювати потенційно унікальну конфігурацію, що задовольняє його певні потреби та вимоги в мережі, а також в інших сферах, таких як мобільні пристрої. Розширюючи продукт (який включає платформу) за допомогою розроблених зовні компонентів або додатків, забезпечується ефективний механізм спрощення регулювання маси. Вищезазначені тенденції є деякими рушійними силами для появи екосистем програмного забезпечення. Наприклад, компаніям, які спочатку досягли успіху за допомогою свого веб-додатка, потрібно перехресне платформу свого додатка та звернення до непрямих розробників. Розробники можуть надати функціональність, яка відповідає потребам сегментів користувачів, які компанія, що надає платформу, не зможе розробити самостійно. Хоча компанії мають різні причини для переходу до програмних екосистем, можна визначити багато переконливих факторів, що пояснюють сучасну тенденцію:

– збільшення значення основних речень для існуючих користувачів

– збільшення привабливості для нових користувачів

– збільшення "статичного характеру" платформи додатків, тобто складна зміна платформи додатків

– прискорення процесу інновації через відкриту інновацію в екосистемі

– співпраця з партнерами в екосистемах спільно для розподілу інноваційних витрат

– міжплатформна функціональність, розроблена партнерами в екосистемі (як тільки успіх буде доведений),

– зниження ССВ (сукупна вартість володіння) для адаптивних функціональних можливостей, розділяючи послуги з партнерами в екосистемі

Новою тенденцією, що існує особливо в галузі компаній *Web* 2.0, але також і в інших сферах, є прийняття стратегії екосистеми програмного забезпечення. Визначення екосистеми програмного забезпечення полягає в наступному:

Екосистема програмного забезпечення складається з програмної платформи, деяких внутрішніх та зовнішніх розробників, спільноти експертів у цій галузі для обслуговування спільноти користувачів, які приймають відповідні елементи рішення для задоволення потреб.

Уважно вивчаючи детермінацію, можна точно простежити шлях від лінії програмного рішення до програмних екосистем, тобто є платформа і ряд внутрішніх розробників, що розробляють рішення на основі платформи. Однак, крім вищезазначеного, існують комутовані комутатори або спільнота зовнішніх розробників, які розробляються на одній платформі, розширюючи, таким чином, продукти, випущені компанією.

Оглядаючи екосистеми, можна вибрати такі об'єкти та предмети:

– операційна система – ключовий об'єкт екосистеми на базі ОС, дає основу, вимоги та обмеження для програмного забезпечення, яке розгортається на ній;

– постачальник операційної системи – компанія, що розробляє операційну систему. Він може взаємодіяти з постачальниками пристроїв та постачальниками *OEM* спеціально для забезпечення стабільності та сумісності з пристроями, які, в кінцевому результаті, задовольняють потреби клієнта більш ефективним методом.

– постачальник пристрою – компанія, що виробляє пристрої, включаючи процесори, системні плати, периферію тощо;

– пристрій – власне пристрій, створений Постачальником Пристрою. Взаємодіє з операційною системою і, зокрема, також може працювати з програмним забезпеченням;

– драйвер пристрою – це артефакт більш низького рівня абстракції, який необхідний для розгляду як частини екосистеми, оскільки він є компонентом взаємодії, що забезпечує взаємодію з Операційною системою;

– програмне забезпечення – власне програмне забезпечення, розроблене компанією;

– розробник програмного забезпечення – компанія, яка ініціювала розробку програмного забезпечення і має на неї права, такі ж, як і на підтримку;

– зовнішній розробник – суб'єкт, який може бути партнером розробника програмного забезпечення у створенні програмного забезпечення або може створювати окремі блоки, які розширюють основні функціональні можливості програмного забезпечення.

– Постачальник *OEM* – компанія, яка приймає рішення для *OEM*. Для досягнення цієї мети необхідно мати для цього договори з Постачальником ОС та Пристроєм, оскільки вони надають компоненти для прийняття рішень *OEM*;

– Постачальник *OEM* – компанія, яка продає рішення *OEM*;

– підприємство – клієнт, який має власну промислову базу, персонал і є покупцем апаратних і програмних рішень;

– машина – набір комп'ютерів, персональних помічників, електронного обладнання, на якому можна встановити програмне забезпечення;

– користувач / персонал – кінцевий користувач програмного забезпечення. У випадку перевірки підприємства – користувач є частиною підприємства, але також це може бути окрема приватна особа.

Розглянути екосистему можна за такою формулою:

Нехай екосистема буде *S*, набір компонентів буде *C*, а набір відносин – *R*. Тоді $S = (C, R)$, де E, R – непусті множини. Таким чином: C = {операційна система, постачальник ОС, постачальник пристрою, пристрій, драйвер пристрою, розробник програмного забезпечення, зовнішній розробник,

постачальник *OEM*, постачальник, постачальник, підприємство, користувач / персонал}, $C \subset C$.

R = {Використання, створення, розширення, співпраця, сумісність, виготовлення, експлуатація, вміст, надання, підтримка, придбання, налаштування, продаж}, $R \subset$ C.

Створюючи модель, можна розділити окремі частини, які можна розглядати окремо від інших частин.

Екосистема програмного забезпечення, організованого навколо програми, можна в деякому сенсі розглядати як зворотну функціонування екосистеми. Ця категорія орієнтована на сферу застосування та часто починається з додатка, який має успіх на ринку без підтримки екосистеми навколо нього. Ця перша хвиля успіху залучає в основному велику кількість клієнтів та чудову фінансову основу для будь-якої компанії, яка досягає успіху в цій галузі.

Успіх програми на ринку, як правило, створює значну кількість певних запитів, які компанія не в змозі виконати через обмежені ресурси НДДКР, а отже, така бізнес-модель зазвичай не враховує функції, які використовуються підмножиною клієнтів . У відповіді компанії зазвичай дають *API* (програмний інтерфейс програми), щоб спочатку клієнти могли найняти розробників програмного забезпечення для розширення програми за допомогою функціональних можливостей, специфічних для клієнта. Як тільки компанія починає відкривати додаток за допомогою надання *API*, додаток переходить до предметно-орієнтованої платформи, за допомогою якої інженери-розробники сторонніх розробників можуть розширюватися до розширених збірок або мостів до інших додатків, які мають велике значення для підмножина клієнтів.

Успіхдодаток на ринку, як правило, генерує велику кількість конкретних запитів, які компанія не може задовольнити через обмежені ресурси досліджень та розробок, а також тому, що бізнес-модель, як правило, не включає функції, які використовуються підмножиною клієнтів. У відповідь на це компанії зазвичай пропонують *API* (Інтерфейс програмування програм), щоб спочатку замовники могли найняти розробників програмного забезпечення для вдосконалення додатків із функціональністю, що відповідає замовникам. Як тільки компанія

починає відкривати додаток, надаючи *API*, додаток стає предметно-орієнтованою платформою, розробниками якої є треті сторони, вона може поширюватися на розширені збірки або мости до інших додатків, які є більш важливими для підгрупи клієнтів. Успішно переходячи від підходу, керованого компанією, до підходу, орієнтованого на застосування, до платформи,

Третя категорія екосистем програмного забезпечення менш також має менше значення в порівнянні з попередньою. З іншого боку, протягом тривалого часу це був священний Грааль програмних платформ, наприклад, при підтримці конфігурації, яка настільки інтуїтивно зрозуміла (іншими словами, змодельована для легкого розуміння кінцевого користувача), шанс, що кінцевий користувач може необхідні програми самостійно.

Більшість екосистем для кінцевого користувача набуває форми мови програмування (*DSL*), графіки чи тексту. Через великий обсяг операцій, необхідних для створення *DSL*, і всіх інструментів для нього, контекст *DSL* і, отже, поле програми, в якому обмежуються кінцеві користувачі, які можуть створювати програми, має тенденцію бути обмеженим. Унікальна перспектива полягає в тому, що ця категорія екосистем програмного забезпечення є вужчою, ніж орієнтовані на програми екосистеми програмного забезпечення (які, в свою чергу, вже є більш сфокусованими, а не екосистемами ОС).

Явні приклади екосистем для кінцевого користувача – *Microsoft Excel, Lego Mindstorms* та засоби створення, такі як *Channels Yahoo*! І *Microsoft PopFly*. Важливо зазначити, що в області мобільних пристроїв не існує успішних екосистем кінцевого користувача [7].

Такі екосистеми відрізняються такими особливостями:

– Фактор темпу успіху – це важливість програми, створеної кінцевим користувачем. Якщо результат подібної операції сприймається недостатньо успішно – екосистема стикається з багатьма труднощами.

– Оскільки фактичне створення додатків вироблятиметься групою клієнтів, існує складність спільного використання системи з іншими клієнтами. Через низьку фінансову підтримку спонукання кінцевих користувачів до участі у створенні додатків часто не має вагомих причин [7].

Для більшості компаній виробників програмних систем розробка програмного забезпечення великих додатків – досить складний, дорогий, непередбачуваний і повільний процес. Чотири десятиліття досліджень розробки програмного забезпечення привели до появи широкого спектру методів управління складністю розробки програмного забезпечення системного програмного забезпечення. Однак через темпи зростання сучасних програмних систем та дослідницького базового рівня компаній (НДДКР) нам потрібні нові рішення врегулювання складності розробки програмного забезпечення.

Хоча зараз можна визначити три тенденції, що сприяють розвитку складності розробки програмного забезпечення програмного забезпечення, і нам слід проаналізувати і зрозуміти витік проблеми з нього ще краще, ніж раніше. Перша тенденція – широко поширене явище використання підходу лінійки програмних продуктів. За останнє десятиліття структурований підхід до внутрішньоорганізаційного повторного використання активів програмного забезпечення досяг нового рівня вдосконалення. Однак прийняття тенденції лінійки програмного продукту також приносить новий рівень залежності в організації, відсутній у підході орієнтації на продукт, додаючи складності. Друга тенденція – широка глобалізація розробки програмного забезпечення програмного забезпечення у багатьох організаціях. Хоча, в особливо міжнародних компаніях,

2.5. Висновки до розділу

У цьому розділі розглянуті питання щодо вимог, які слід впровадити в розробляється програмний продукт. Також розглядається огляд середовища, що розвивається, і опис метрик та їх значення в програмному продукті.

Тому у другій частині диплому було проаналізовано:

– Формальний опис метричних характеристик програмного забезпечення

– Оцінка метрики вихідного коду означає функціональні вимоги

– Вимірювання означає загальну архітектуру

Також у розділі було розглянуто архітектуру для вилучення метрик разом із їх формальним описом за допомогою таких інструментів, як *OCL* та метамодель *UML*. Більше того, бібліотека заходів *FLAME* служить вхідним матеріалом для формалізації метрик. Мета проекту *FLAME* – перетворити розвиток щільних бібліотек лінійної алгебри з мистецтва, зарезервованого для експертів, у науку, зрозумілу як новачкам, так і експертам. Замість того, щоб бути лише бібліотекою, проект охоплює нову нотацію для вираження алгоритмів, методологію для систематичного виведення алгоритмів, інтерфейси прикладних програм (*API*) для представлення алгоритмів у коді та інструменти для механічного виведення, реалізації та аналізу алгоритмів та реалізації.

РОЗДІЛ 3
РОЗРОБКА СИСТЕМИ ДЛЯ ВИЗНАЧЕННЯ РІВНЯ НАДІЙНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Розробка метричних лічильників

Програма написання висловлюється з питання розширюваності. Там було багато ідей, як це реалізувати, через Базу даних або файл *XML*. Але Мова *Java* – чудова мова, яка включає багато вирішених питань Існує механізм *Java Scripting*, який використовувався для розширення програми.

3.1.1. Сценарії для платформи *Java*

Платформа *Java* забезпечує багаті ресурси як для настільних ПК, так і для веб-додатків розвитку. Однак використання цих ресурсів поза платформою має було непрактично, якщо ви не вдаєтесь до власних програмних рішень.

Запит на специфікацію (*JSR*), який допомагає розробникам інтегрувати *Java* технології та мови сценаріїв, визначивши стандартну структуру та інтерфейс прикладного програмування (*API*), щоб зробити наступне:

1. Доступ та керування об’єктами, заснованими на технологіях *Java*, із середовища сценаріїв;

2. Створення веб-вмісту за допомогою мов сценаріїв;

3. Вбудовування середовища сценаріїв у додатки, засновані на технологіях *Java*.

Більшість мов сценаріїв набираються динамічно. Зазвичай ви можете створювати нові змінні без попереднього визначення типу змінної, і ви можете повторно використовувати змінні для зберігання значень різних типів. Крім того, мови сценаріїв, як правило, виконують багато перетворень типів автоматично, наприклад, перетворюючи число 10 у текст "10" за необхідності. Хоча деякі мови сценаріїв компілюються, більшість мов інтерпретуються. Середовища сценаріїв зазвичай виконують компіляцію та виконання сценаріїв в рамках одного і того ж

процесу. Зазвичай ці середовища також аналізують та компілюють сценарії в проміжний код під час їх першого запуску.

Ці якості мов сценаріїв допомагають швидше писати програми, виконувати команди неодноразово та зв'язувати компоненти різних технологій. Спеціальні мови сценаріїв можуть виконувати певні завдання легше або швидше, ніж інші мови загального призначення. Наприклад, багато розробників вважають, щоМова сценаріїв *Perl*- чудовий спосіб обробити текст та створити звіти. Інші розробники використовують мови сценаріїв, доступні вбашабокшкомандні оболонки для управління командами та завданнями. Інші мови сценаріїв допомагають зручно визначати користувальницькі інтерфейси або веб-вміст. Розробники можуть використовувати мову програмування та платформу *Java* для будь-якого з цих завдань, але мови сценаріїв іноді виконують цю роботу також добре або краще. Цей факт не зменшує потужності та багатства платформи *Java*, а просто визнає, що мови сценаріїв посідають важливе місце в наборі інструментів розробника.

Наприклад, уявіть собі калькулятор із набором основних операцій. Хоча базовий калькулятор може мати лише чотири або п'ять основних операцій, ви можете забезпечують програмовані функціональні клавіші, які користувач може налаштувати. Клієнти можуть використовувати будь-яку мову сценаріїв, яку вони вважають за краще додавати іпотечні розрахунки, перетворення температури або навіть більш складна функціональність калькулятора.

Іншим прикладом цієї співпраці може бути текстовий процесор, який дозволяє клієнту створювати власні фільтри для генерації різних форматів файлів.

3.1.2. Описання використаних бібліотек

Короткий опис використовуваних бібліотек:

– *import java.awt. **;

– *import java.awt.event. **;

– *import javax.swing. **;

– *import javax.swing.filechooser.FileFilter*;

– *import java.io. **;

– *import java.util. **;

– *import java.script. **;

Пакет *java.awt* містить усі класи для створення користувальницьких інтерфейсів та для малювання графіки та зображень. Об'єкт користувальницького інтерфейсу, такий як кнопка або смуга прокрутки, за термінологією *AWT* називається компонентом. Клас *Component* є коренем усіх компонентів *AWT*. Детальний опис властивостей, якими мають усі компоненти *AWT*, див. У розділі Компонент.

Деякі компоненти запускають події, коли користувач взаємодіє з компонентами. Клас події *AWT* та його підкласи використовуються для представлення подій, які можуть запускати компоненти *AWT*. Опис моделі події *AWT* див. У *AWTEvent*. Контейнер – це компонент, який може містити компоненти та інші контейнери. Контейнер може також мати менеджер макета, який контролює візуальне розміщення компонентів у контейнері. Пакет *AWT* містить кілька інтерфейсів макета для створення власного менеджера макетів.

Пакет *java.awt.event* Надає інтерфейси та класи для роботи з різними типами подій, що запускаються компонентами *AWT*. Події запускаються джерелами подій. Прослуховувач подій реєструється з джерелом подій, щоб отримувати сповіщення про події певного типу. Цей пакет визначає події та прослуховувачі подій, а також адаптери прослуховувача подій, які є зручними класами для полегшення процесу написання прослуховувачів подій.

Пакет *javax.swing* Забезпечує набір "полегшених" (на мові *Java*) компонентів, які для максимально можливий ступінь, однаково працюйте на всіх платформах.

Загалом *Swing* не є безпечним для потоків. До всіх компонентів *Swing* та пов’язаних класів, якщо не задокументовано інше, необхідно отримати доступ до потоку диспетчеризації подій.

Типові програми *Swing* виконують обробку у відповідь на подію, породжену жестом користувача. Наприклад, натискання кнопки *JButton* повідомляє всі *ActionListeners*, додані до *JButton*. Як і всі події, створені користувачем, надсилаються в потоці диспетчеризації подій, на які впливають

обмеження.. Однак вплив полягає лише в побудові та показі гойдалки застосування. Виклики основного методу програми або методів в апплеті – це не викликається в потоці диспетчеризації подій. Таким чином, слід подбати про це передати керування потоком диспетчеризації подій під час побудови та показу додаток або аплет. Кращий спосіб передати управління та розпочати робота з *Swing* полягає у використанні *invokeLater*. Розклад методу *invokeLater*. Запуск для обробки в потоці диспетчеризації подій.

Пакета *javax.swing.filechooser* містить класи та інтерфейси, що використовуються*JFileChooser*компонент.

Пакета *java.io* забезпечує введення та виведення системи через потоки даних, серіалізацію та файлову систему. Якщо не зазначено інше, передача нульового аргументу конструктору або методу в будь-якому класі чи інтерфейсі в цьому пакеті спричинить *aNullPointerException*кинути.

Пакет *java.util* містить фреймворк колекцій, застарілі класи колекцій, модель подій, засоби дати та часу, інтернаціоналізацію та різні класи утиліти (маркер рядків, генератор випадкових чисел та бітовий масив).

Пакет *javax.script API* сценаріїв складається з інтерфейсів та класів, які визначають *Java*TM*Scripting Engines* і забезпечує основу для їх використання в додатках *Java*. Цей *API* призначений для використання програмістами, які бажають виконувати програми, написані мовами сценаріїв у своїх програмах *Java*. Програми на мовах сценаріїв, як правило, надаються кінцевими користувачами програм.

3.2. Блок-схема реалізації програмного забезпечення

Програма створена таким чином, як показано на рисунку 3.1. Починаючи з цього, ми маємо можливість додавати показники, які потрібно обчислити. Наступним кроком є ввімкнення / вимкнення кнопок перевірки, які відповідають за режим вивчення, межі та управління підсвічуванням. Також ви можете ввімкнути / вимкнути всю метрику.

Крок після цього – додавання файлу вихідного коду для аналізу. І завершальний етап набуває

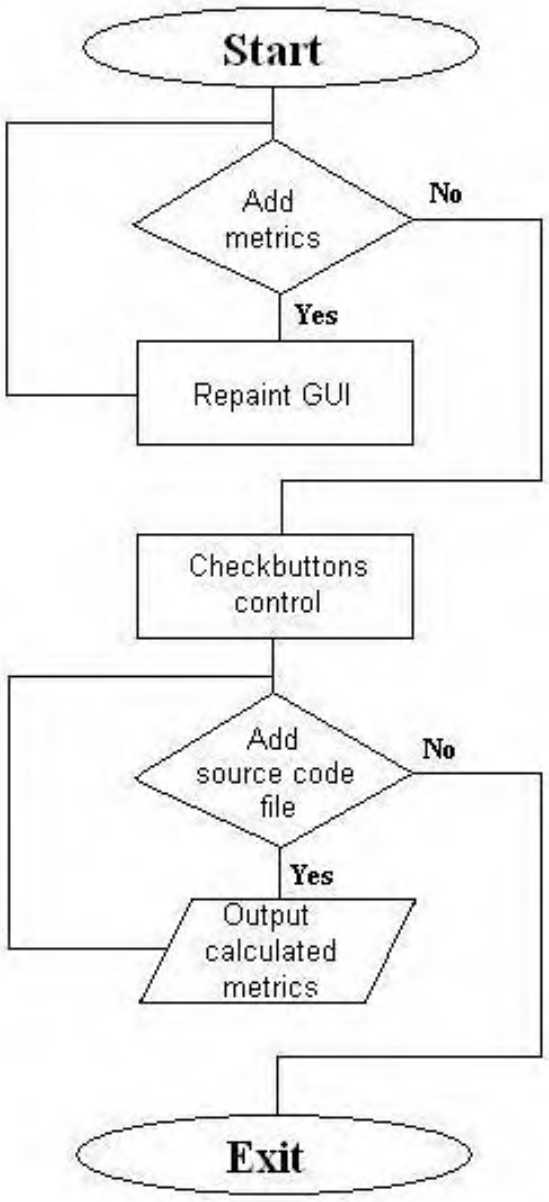

Рис. 3.1. Узагальнена схема алгоритму роботи створеної програмної системи

Реалізуємо це на прикладі створення метрики *LOC*.

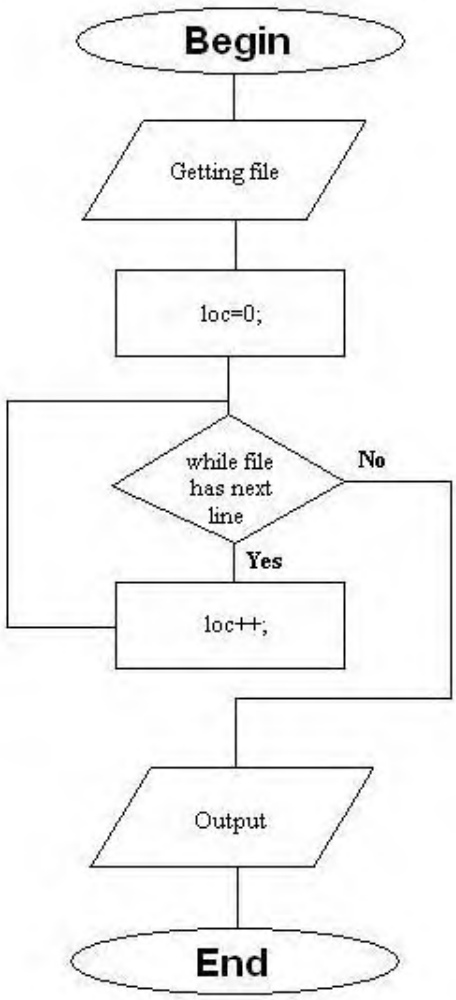

Рис. 3.2. Схема алгоритму розрахунку метрики *LOC*

Як це виглядає в коді *Java*:

```
public class LOC {
    private int CodeLinesCounter(File f) {
    int loc = 0;
        try {
        Scanner scanner = new Scanner(f);

        while (scanner.hasNextLine())
        {

            String lines = scanner.nextLine();
          loc++;
```

```
            }
            scanner.close();

        }  catch (FileNotFoundException fnfex) {
          // ... We just got the file from the JFileChooser,
          //     so it's hard to believe there's problem, but...
          JOptionPane.showMessageDialog(ChekalenkoDiploma.this,
                    fnfex.getMessage());
      }

       if(setBounds.isSelected()){
              if(boLOC.isSelected()){
        if(!(fromVal1<=loc && loc<=toVal1)){
              _loc.setBackground(Color.RED);
        }
        else _loc.setBackground(Color.GREEN);
       }}
       //System.out.println(fromVal1 + "  " + toVal1);
      return loc;

    }
}
```

3.3. Реалізація системи вимірювань

При запуску програми з'являється вікно (рис. 3.3). Спочатку він має лише два показники, що стосуються метрик *Lines Of Code* (*LOC*) та цикломатичної складності. Отже, згідно з завданням дипломної роботи ми можемо бачити, що програма є багатомовною і має можливість вибору мови програмування вихідного коду. Це впливає на розширення файлів, які можна вибрати за

допомогою кнопки "Завантажити файл". Програма *Aslo* обчислює ризик нестабільного коду відповідно до цикломатичної складності. А вбудований текст подає класи, що містяться в коді (якщо такий є). У всіх полях, крім Обмежувачів, регістри відсутні.

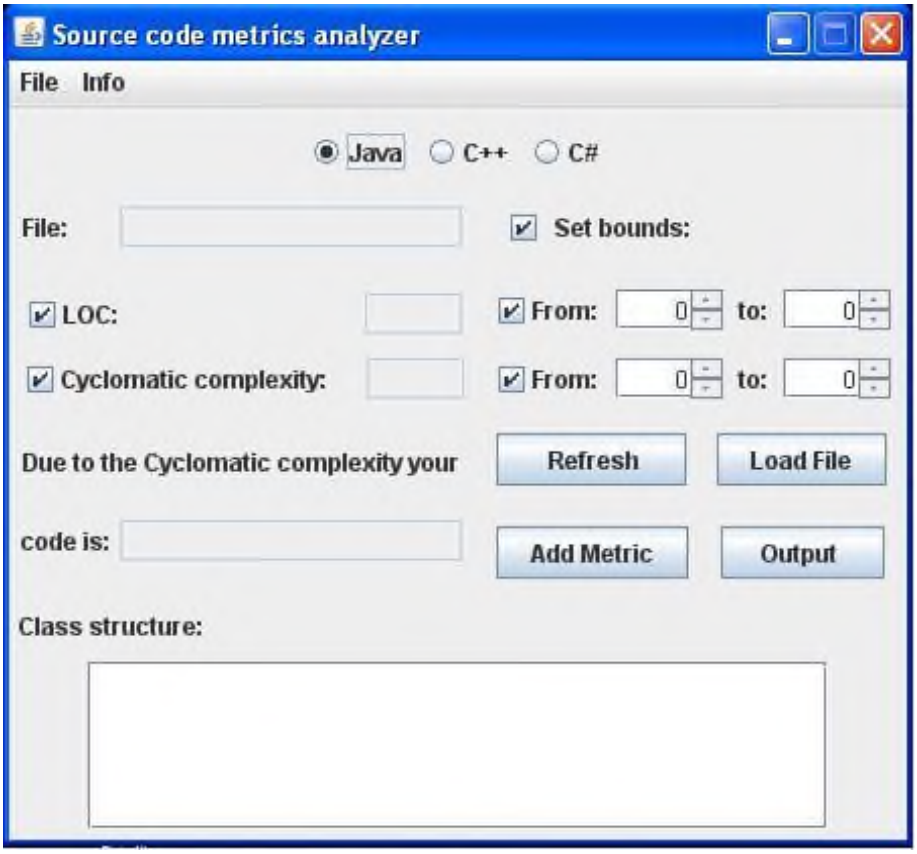

Рис. 3.3. Початковий перегляд програми

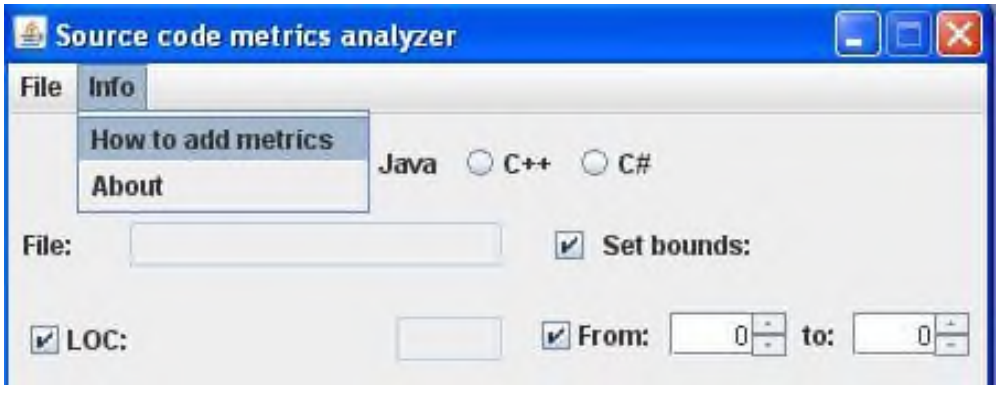

Рис. 3.4. Як додати опцію меню метрик

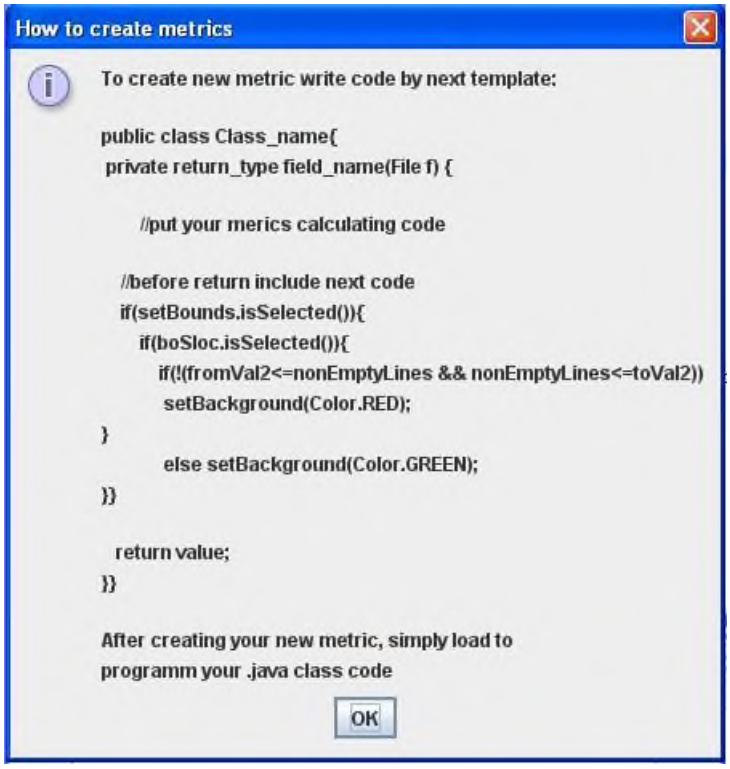

Рис. 3.5. Короткий опис створення та додавання нової метрики

Створення метрики максимально спрощено. Ви просто створюєте клас із приватним полем, кажучи, який тип він поверне (насправді це не має значення, тому що повернене значення все одно буде перетворено на рядок). Потім ви пишете код, який робить певний розрахунок завдяки метриці, яку ви пишете. І перед написанням того, що поверне ваше поле, додайте код для кнопок перевірки, про який буде сказано нижче. Після цього збережіть свій код у файлі *.java* та завантажте його кнопкою “Додати метрику” (рис. 3.6).

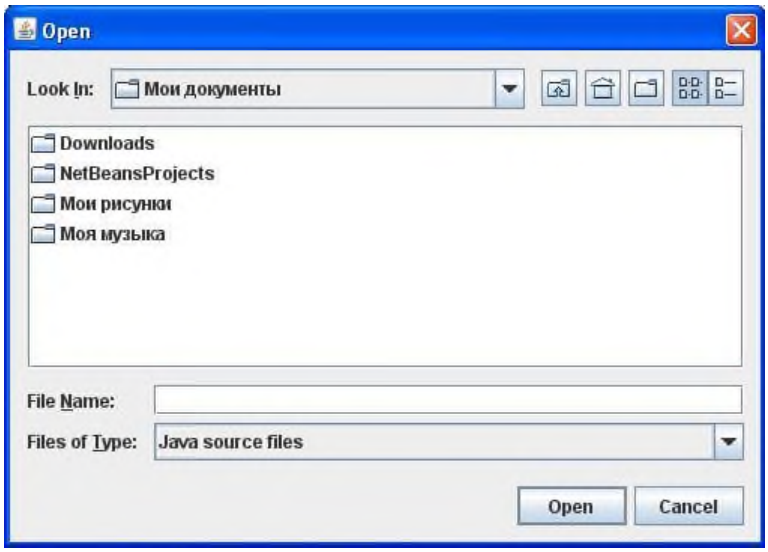

Рис. 3.6. Файл, вибраний для додавання показників

Завантаження файлу *.java* з метрикою (рис. 3.7 та рис. 3.8).

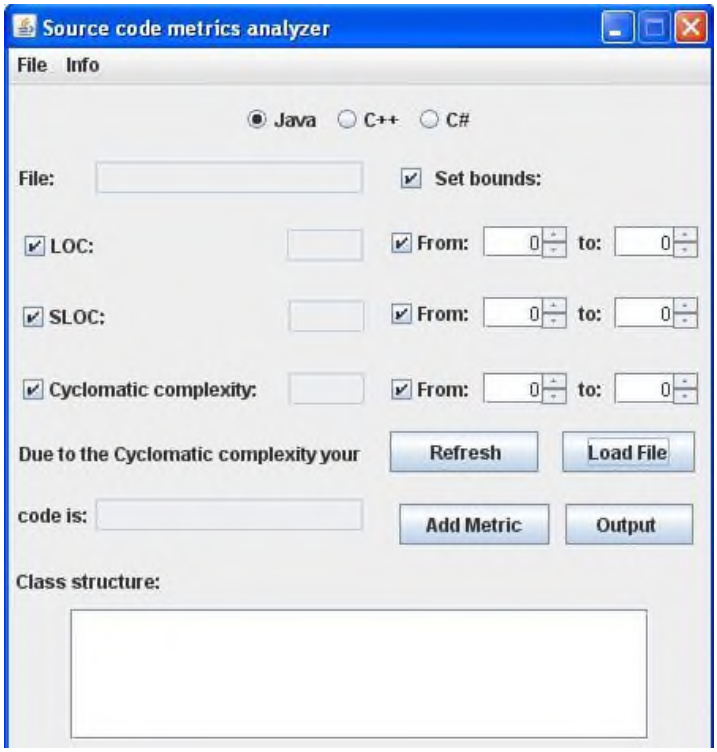

Рис. 3.7. До програми додано метрику *SLOC*

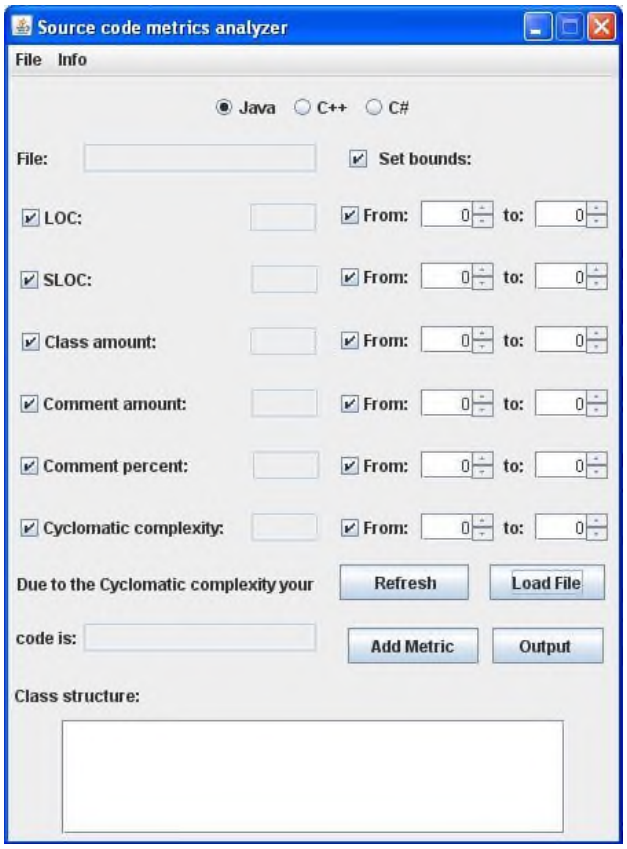

Рис. 3.8. Додано показники суми класу, кількості коментарів та відсотків коментарів

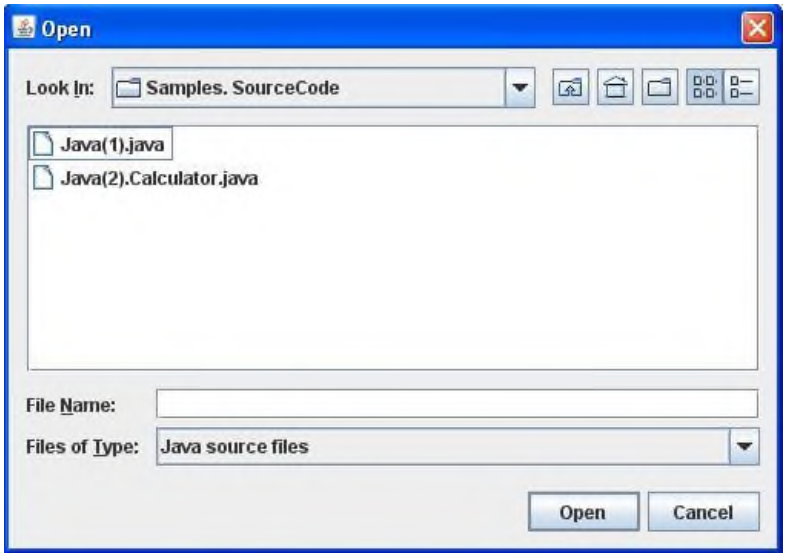

Рис. 3.9. Вибір файлу для аналізу

А тепер давайте подивимось, як працює програма. Отже, спочатку обрана мова *Java*, відповідно ми вибираємо файли *.java* (Рис. 3.9). Також ви можете завантажити файл коду *souce*, натиснувши *File-> Open*.

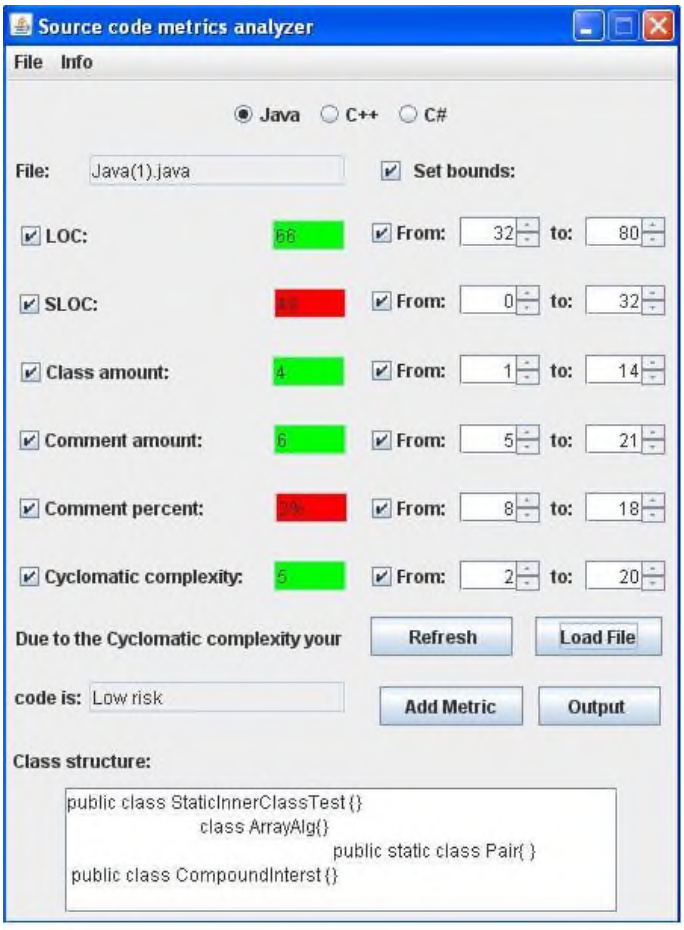

Рис. 3.9. Програма в дії

Після вибору файлу для аналізу програма негайно обчислює результати. Як показано на рисунку 3.10, можемо бачити результати в текстових областях. Синій або зелений індикатор відображається, якщо результат ми отримали в межах, заданих у розділі меж. Відповідно зелений означає "ОК", червоний означає, що ваш результат не в діапазоні.

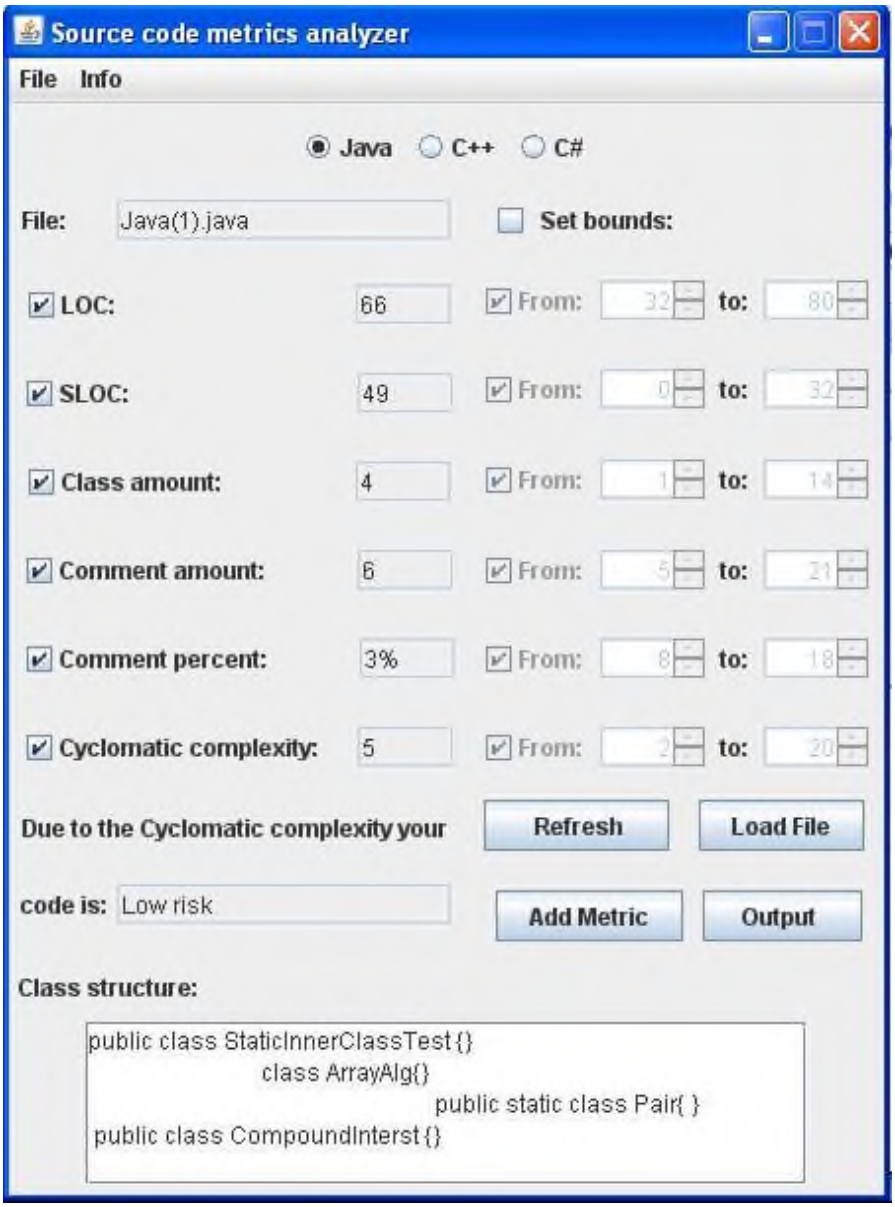

Рис. 3.10. Вимкнення меж

Якщо зніміть прапорець біля пункту "Встановити межі", діапазони вимкнено. Кольори не виділяться. Це робиться на випадок, якщо програма буде використана для не навчальних цілей (рис. 3.11).

Також є можливість зняти певні межі, якщо вони не вимагаються (рис. 3.12).

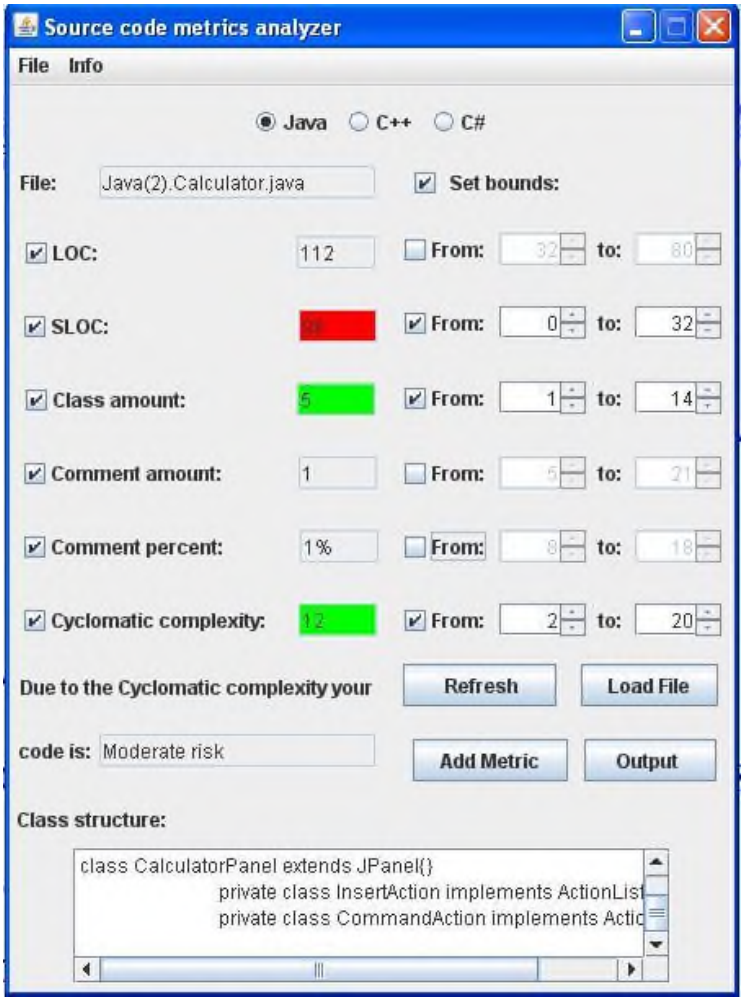

Рис. 3.11. Налаштування меж

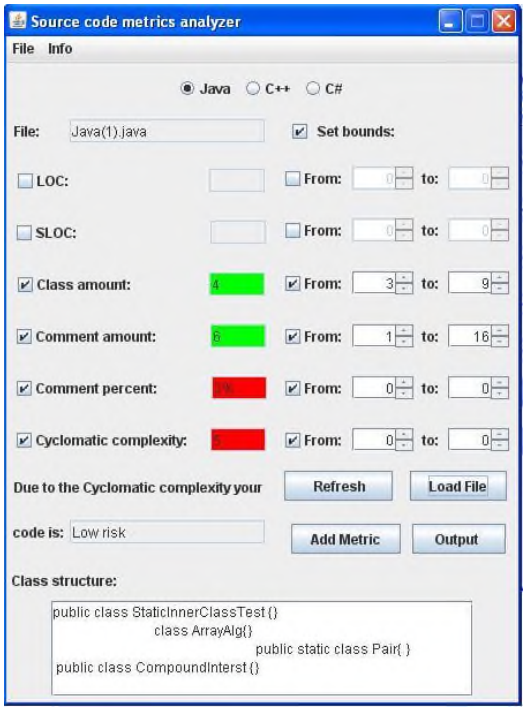

Рис. 3.12. Зніміть певні показники

Якщо певні метрики не вимагаються, їх можна відмінити. У цьому випадку ці показники стали вимкненими, тому як розділ меж, який з ними пов’язаний.

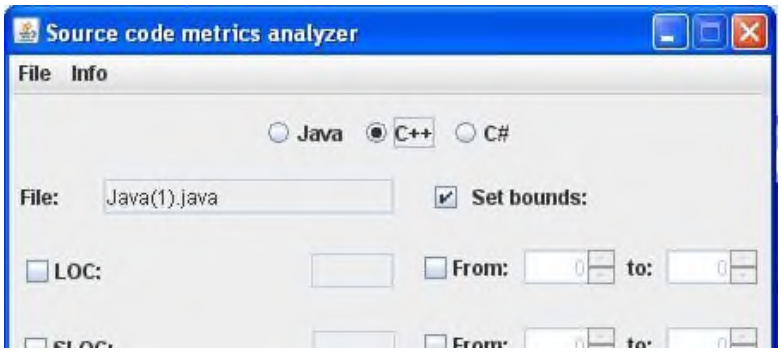

Рис. 3.13. Переключення мови на *C* ++

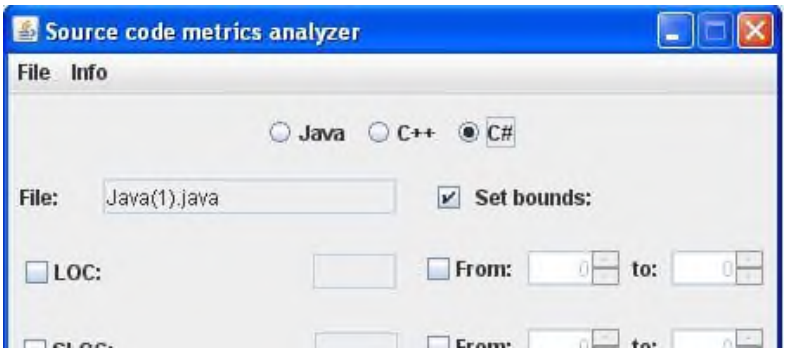

Рис. 3.14. Переключення мови на *C* #

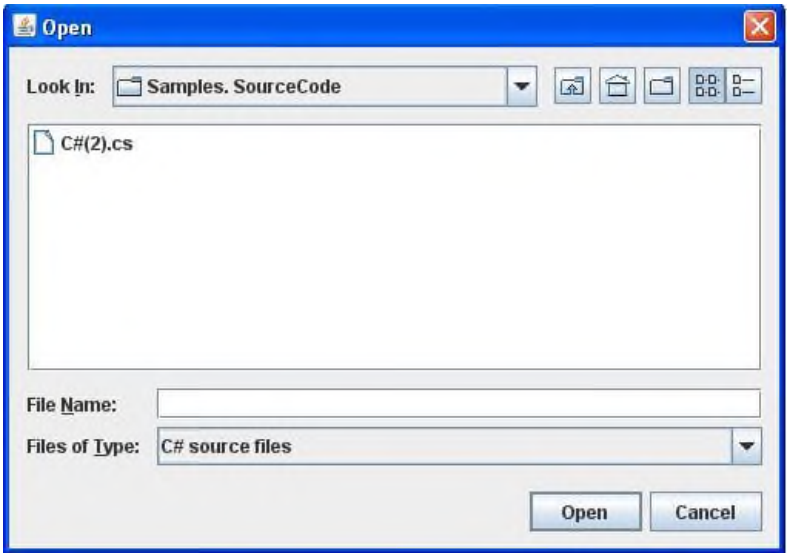

Рис. 3.15. Відкриття файлу коду *C* # джерела

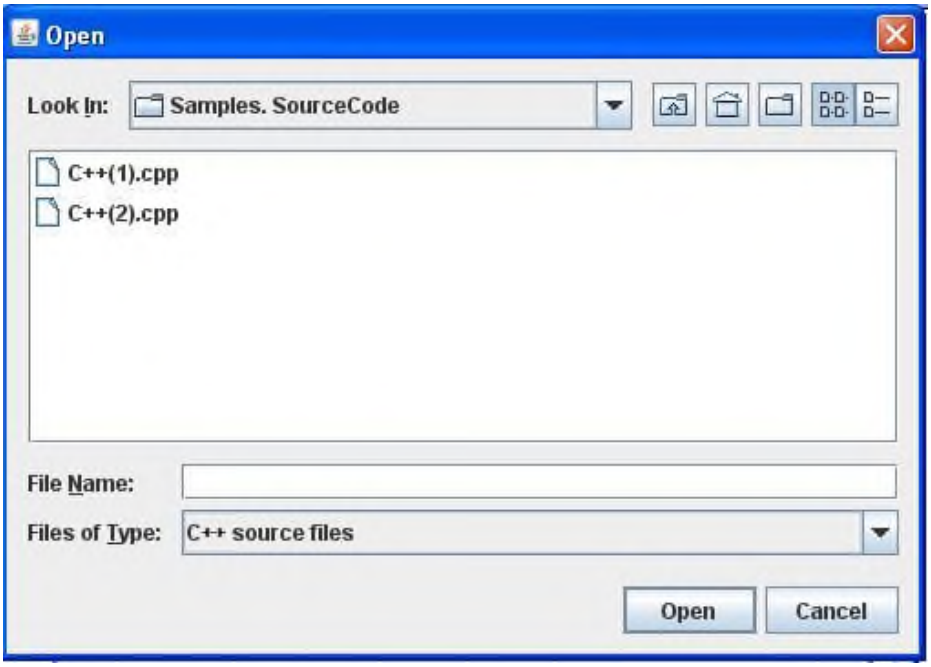

Рис. 3.16. Відкриття вихідного файлу *C ++*

Відповідно до мови файли з розширенням відображаються у завантажувачі файлів.

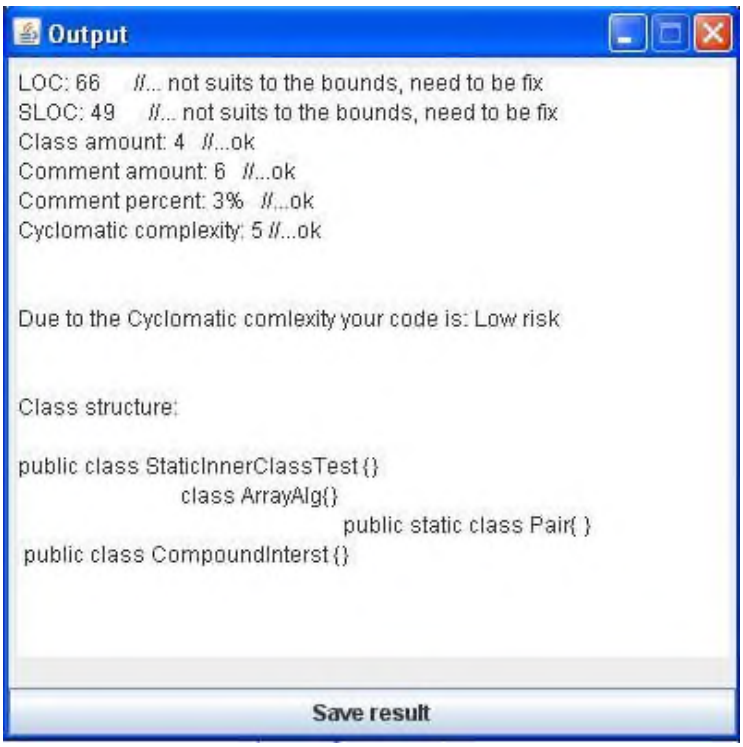

Рис. 3.17. Вихідний результат

Результат обчислення можна переглянути у вікні виводу, викликаному натисканням кнопки «Висновок». У цьому новому кадрі результат може бути збережений у текстовому файлі (рис. 3.18).

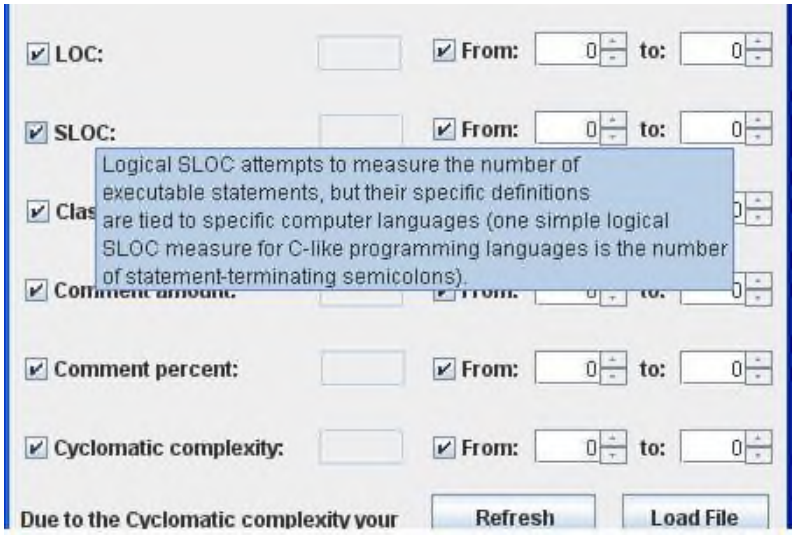

Рис. 3.18. Метричний опис *SLOC*

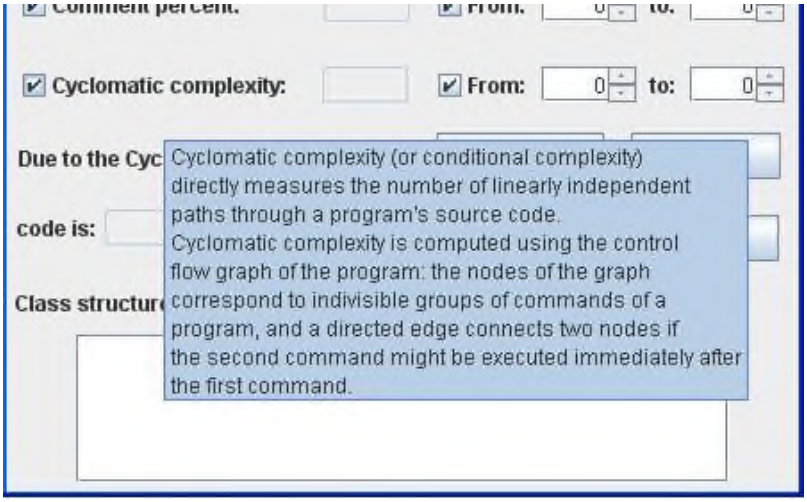

Рис. 3.19. Метричний опис цикломатичної складності

Щоб прочитати опис метрики, просто наведіть вказівник на метрику, яка вас цікавить, і з’явиться підказка.

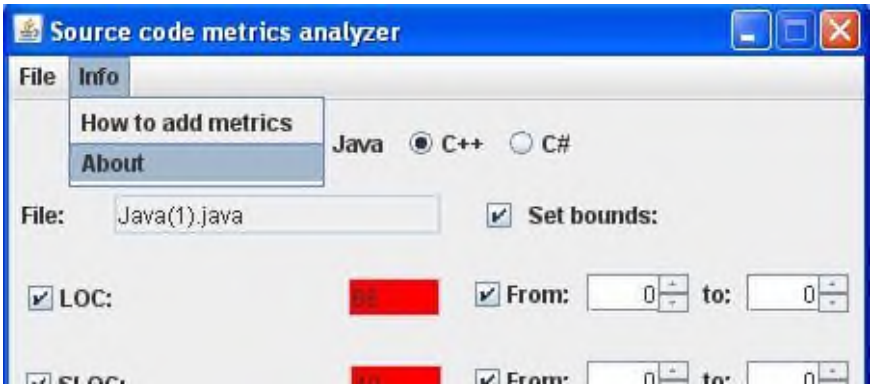

Рис. 3.20. Розкриття пункту меню «*About*»

3.4. Дослідження та конкурентний аналіз розробленої системи з існуючими аналогами

Давайте проаналізуємо м’який продукт, який був розроблений у цій роботі, з деякими існуючими аналогами. Обраний аналог називається Зрозуміти.

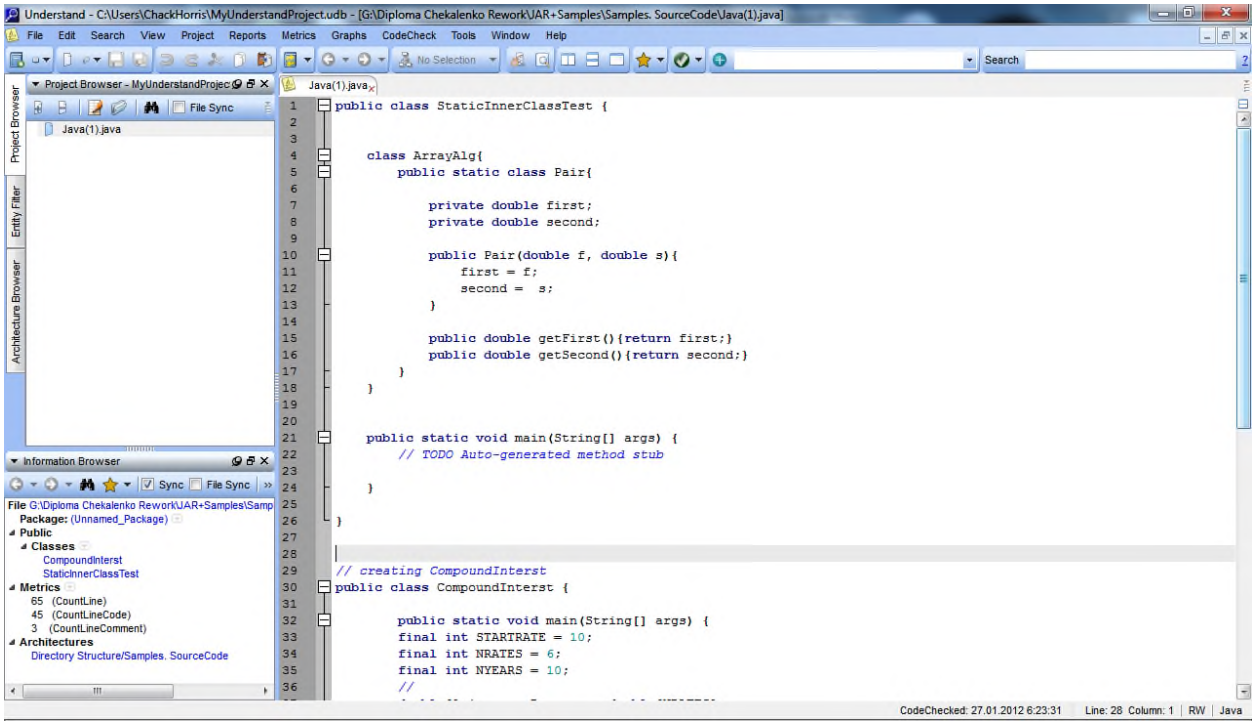

Рис. 3.22. Аналізатор коду

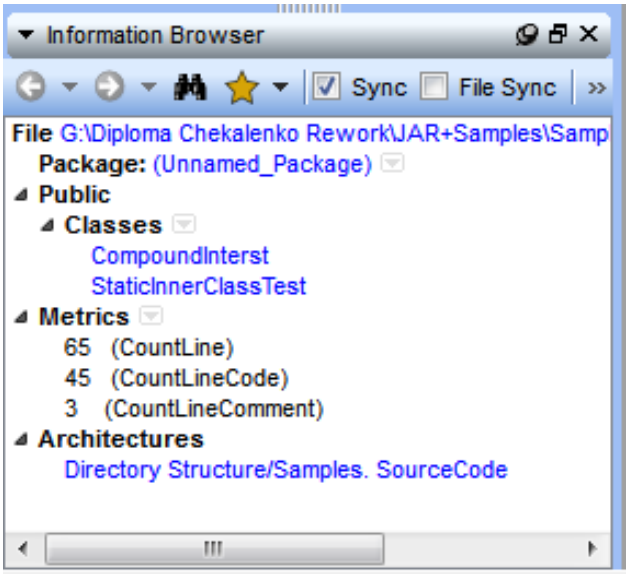

Рис. 3.23. Вікно метрик.

Як бачимо, кількість показників обмежена 4. Порівняно з розробленим м'яким продуктом він дає. Перш за все, розуміння не можна розширювати і має обмежену кількість метрик. У нього немає варіантів вибору.

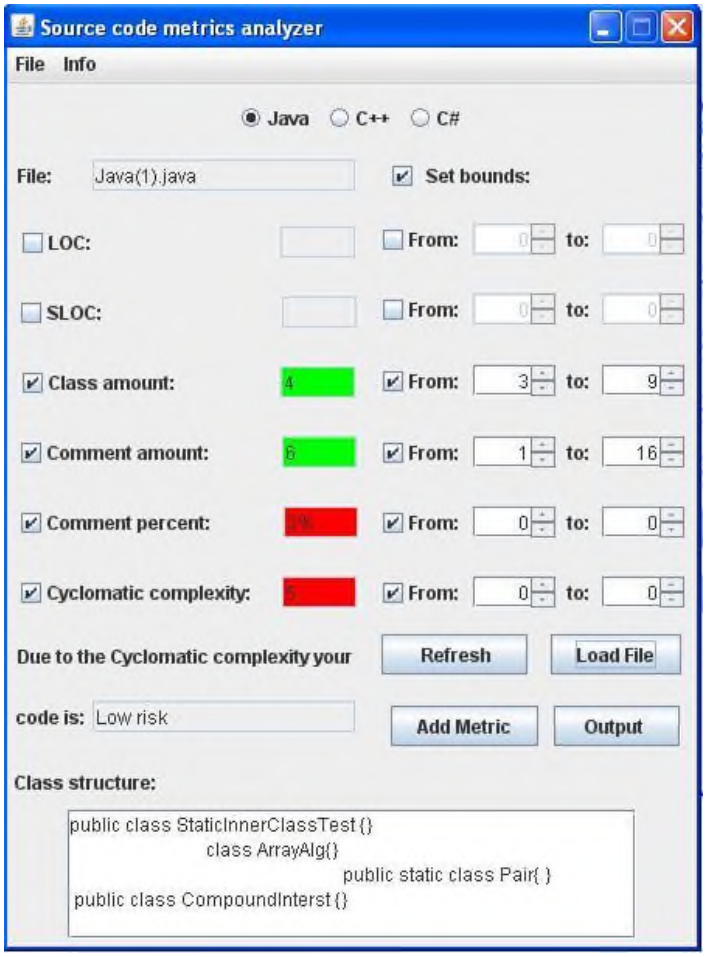

Рис. 3.24. Розроблений метричний аналізатор.

Окрім того, що стосується навчального процесу, він не може показати вам меж і має безліч варіантів, які не корисні для навчального процесу.

3.5. Висновки до розділу

У цьому розділі показаний результат роботи, програмний продукт, якій оцінює надійність роботи програми, що ілюструє її скріншотами. Також це має узагальнену блок-схему роботи програми, описує спосіб розвитку продукту, зокрема спосіб написання класи, які можна включити до запущеного програмного забезпечення.

Також розроблене програмне забезпечення було порівняно з існуючими аналогами програмного забезпечення.

В третій частині диплому було проаналізовано:

– програма вимірювання метричних вимірів

– розробка автоматизованої системи вимірювань

– дослідження та конкурентний аналіз розробленої системи з існуючими аналогами

ВИСНОВКИ

Важливість вивчення метрик очевидна. Це дає можливість зробити більш компактний код і, як результат, швидкі та ефективні програми. Легка програма для дослідження метричних характеристик вихідного коду – це саме рішення для полегшення процесу вивчення програмування.

Результати роботи спрямовані на використання у навчальному процесі та можуть застосовуватися навчальними закладами.

Як бачимо з перегляду існуючих програмних аналізаторів метрик, вони не потребують функцій, необхідних для навчального процесу. Програмне забезпечення, розроблене в цій роботі, враховує всі переваги, недоліки та перевищення. Серед них можна виділити такі особливості:

– легкість;

– відкритість джерела;

– розширюваність;

– простота інтерфейсу;

– багатомовність.

Метрики програмного забезпечення можна розділити на три категорії -

• метрики продукту- описує характеристики продукту, такі як розмір, складність, особливості дизайну, продуктивність і рівень якості.

• метрики процесу- ці характеристики можуть використовуватися для поліпшення діяльності по розробці і супроводу програмного забезпечення.

• метрики проекту- ці метрики описують характеристики і виконання проекту. Приклади включають число розробників програмного забезпечення, штатний розклад протягом життєвого циклу програмного забезпечення, вартість, графік і продуктивність.

метрики продукту- описує характеристики продукту, такі як розмір, складність, особливості дизайну, продуктивність і рівень якості.

метрики процесу- ці характеристики можуть використовуватися для поліпшення діяльності по розробці і супроводу програмного забезпечення.

метрики проекту- ці метрики описують характеристики і виконання проекту. Приклади включають число розробників програмного забезпечення, штатний розклад протягом життєвого циклу програмного забезпечення, вартість, графік і продуктивність.

Деякі показники відносяться до декількох категорій. Наприклад, показники якості процесу в проекті є як показниками процесу, так і показниками проекту.

Метрики якості програмного забезпеченняявляють собою підмножина метрик програмного забезпечення, які фокусуються на аспектах якості продукту, процесу і проекту. Вони більш тісно пов'язані з метриками процесу і продукту, ніж з метриками проекту.

Метрики якості програмного забезпечення можна розділити на три категорії:

- Метрики якості продукції
- Показники якості в процесі
- Метрики якості обслуговування

Метрики якості продукції

Ці показники включають в себе наступне -

- Середній час до відмови
- щільність дефектів
- Проблеми з клієнтами
- задоволеність клієнтів

Середній час до відмови

Це час між невдачами. Цей показник в основному використовується з критично важливими системами безпеки, такими як системи управління повітряним рухом, авіоніка і зброю.

щільність дефектів

Він вимірює дефекти щодо розміру програмного забезпечення, вираженого у вигляді рядків коду або функціональної точки і т. Д., Т. Є. Вимірює якість коду на одиницю. Цей показник використовується в багатьох комерційних системах програмного забезпечення.

Проблеми з клієнтами

Він вимірює проблеми, з якими стикаються клієнти при використанні продукту. Він містить погляд клієнта на проблемне простір програмного забезпечення, яке включає в себе проблеми, не пов'язані з дефектами, а також проблеми з дефектами.

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.

У третій частині диплому було проаналізовано:

– аналіз існуючих програмних показників.

– обмеження;

– прийняття та громадська думка;

– формальний опис метричних характеристик програмного забезпечення;

– оцінка метрики вихідного коду означає функціональні вимоги;

– вимірювання означає загальну архітектуру;

– програма вимірювання метричних вимірів;

– розробка автоматизованої системи вимірювань;

– дослідження та конкурентний аналіз розробленої системи з існуючими аналогами

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

Додаток А

Схема алгоритму визначення метрик рядків коду