ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ

ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інженерії програмного забезпечення

Пояснювальна записка

до бакалаврської роботи

на ступінь вищої освіти бакалавр

на тему: «**РОЗРОБКА WEB-ДОДАТКУ ДЛЯ СТАТИЧНОГО ДОСЛІДЖЕННЯ МЕТРИК ПРИКЛАДНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА МОВІ JAVA**»

Виконав: студент 5 курсу, групи ППЗ-52
Спеціальності
121 Інженерія програмного забезпечення
(шифр і назва спеціальності)
Кукла А.В.
(прізвище та ініціали)
Керівник Коваленко Д.С.
(прізвище та ініціали)
Рецензент_____
(прізвище та ініціали)

Київ – 2021

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ТЕЛЕКОМУНІКАЦІЙ

Кафедра інженерії програмного забезпечення
Ступінь вищої освіти бакалавр
Спеціальність 121 Інженерія програмного забезпечення
(шифр і назва)

Завідувач кафедри
Інженерія програмного забезпечення
_____ Негоденко О.В.
“____” _____2021 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТА

Куклі Артему Володимировичу

1. Тема роботи: Розробка web-додатку для статичного дослідження метрик прикладного програмного забезпечення на мові java.

2. Керівник роботи: Коваленко Даноло Сергійович асистент кафедри ІПЗ
3. Затверджені наказом вищого навчального закладу від 16.03.2021 року №65

4. Строк подання студентом роботи 01 . 06 . 2021 року
5. Вихідні дані до роботи: розробити web-додаток для сатичного дослідження

6. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити):
   1. Принцип статичного аналізу ПЗ;
   2. Три основних види дослідження вихідних текстів аналізу ПЗ;
   3. Веб-додатки і веб-сервіси на платформі java;
   4. Метрики середовища java;
   5. Науково-технічна література.

7. Графічна частина роботи представлена на перезнтації в кінці диплома.

8. Дата видачі завдання 19.04.2021

КАЛЕНДАРНИЙ ПЛАН ГРАФІК

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Аналіз та підбір літератури по обраній тематиці бакалаврської роботи, формування завдання	19.04.2021	Викон.
2	Аналіз статичного аналізу з допомогую готових web-додатків для порівняння	24.04.2021	Викон.
3	Дослідження методів статистичного аналізу	05.05.2021	Викон.
4	Аналіз та підбір метри для аналізу пз на java	15.05.2021	Викон.
5	Аналіз отриманих результатів дипломної роботи. Підсумки роботи	20.05.2021	Викон.
6	Розробка доповіді і презентації	01.06.2021	Викон.

Студент _____ Кукла А.В.

(підпис) (прізвище та ініціали)

Керівник роботи _____ Коваленко Д.С.

(підпис) (прізвище та ініціали)

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ
ПОДАННЯ
ГОЛОВІ ДЕРЖАВНОЇ ЕКЗАМЕНАЦІЙНОЇ КОМІСІЇ
ЩОДО ЗАХИСТУ БАКАЛАВРСЬКОЇ РОБОТИ

Направляється студент Кукла А.В. до захисту бакалаврської роботи
(прізвище та ініціали)
за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

на тему: Розробка web-додатку для статичного дослідження метрик прикладного програмного забезпечення на мові java

Бакалаврська робота і рецензія додаються.

Директор інституту _____ Ольховая І.О.
(підпис) (прізвище та ініціали)

Довідка про успішність

Кукла А.В. за період навчання в Науковому інституті телекомунікацій,
(прізвище та ініціали)
з 2019 року до 2021 року повністю виконав навчальний план за напрямом підготовки, спеціальністю з таким розподілом оцінок за:
національною шкалою: відмінно ____%, добре ____%, задовільно ____%;
шкалою ECTS: A ____%; B ____%; C ____%; D ____%; E ____%.

Провідний фахівець інституту _____ _____
(підпис) (прізвище та ініціали)

Висновок керівника бакалаврської роботи

Студент Кукла Артем Володимирович під час написання бакалаврської роботи показав гарну технічну та теоретичну підготовку, уміння користуватися науково-технічною літературою. Працюючи над завданнями, які були поставлені керівником, проявив сумлінність, ініціативність та хист до наукової роботи.

Бакалаврська робота виконана на високому рівні і заслуговує оцінку “добре”, а студенту Кукла Артем Володимирович присвоєння кваліфікації «фахівець з розробки та тестування програмного забезпечення».

Керівник роботи _____ Коваленко Д.С.
(підпис) (прізвище та ініціали)
« » _____ 2021 року

Висновок кафедри про бакалаврську роботу

Бакалаврську роботу розглянуто. Студент Кукла А.В.
(прізвище та ініціали)
допускається до захисту даної роботи в Державній екзаменаційній комісії.

Завідувач кафедри
Інженерії програмного забезпечення _____ Негоденко О.В.
(підпис) (прізвище та ініціали)
« » _____ 2021 року

ВІДГУК РЕЦЕНЗЕНТА
по бакалаврській роботі

студента Кукли Артема Володимировича
на тему: «РОЗРОБКА WEB-ДОДАТКУ ДЛЯ СТАТИЧНОГО ДОСЛІДЖЕННЯ МЕТРИК ПРИКЛАДНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА МОВІ JAVA»

Актуальність:

У сучасному світі розробки програмного забезпечення статичний аналіз є запорукою успіху всього проекту. Правильне і своєчасне вбудовування цього етапу в процес написання коду може значно заощадити як тимчасові, так і фінансові ресурси. Одним із факторів, що сприяють на неуспішність проектів є те, що сучасне програмне забезпечення майже ніколи не розробляється повністю з нуля, а досить розширене та модифікує існуючий код і часто включає сторонній вихідний код. Це може призвести до поганої загальної ремонтопридатності, складної розширюваності та великої складності програмного коду, тому в сфері розробки баз даних, де нерідко можна зустріти великі програмні комплекси, статичний аналіз є особливо актуальним.

Позитивні сторони:

1. Проведено аналіз проблем реалізації, проектування та впровадження статичного аналізу метрики прикладного програмного забезпечення на мові java. Виконаний аналіз метрик програмного забезпечення та оцінки зібраних даних.

2. Результат дослідження показав актуальну розробки web-додатку для статичного дослідження метрик прикладного програмного забезпечення, що дозволяє впроваджувати ефективні, економічно та технічні рішення.

3. Зміст дипломної роботи повністю відповідає завданню, а поставлені задачі виконані у повному обсязі.

Недоліки:

1. У дипломній роботі, недостатньо розкрито подальший розвиток програмного забезпечення на мові java для статичного доследження метрики ПЗ.

2. Слід було б виконати аналіз альтернативних технологій, та виконати порівняння показників якості.

Висновки:

Незважаючи на дрібні недоліки, бакалаврська робота заслуговує оцінку **добре**, а студент Кукла Вртем Володимирович - присвоєння кваліфікації «фахівець з розробки та тестування програмного забезпечення».

Якість проекту (роботи)	
Виконано на замовлення підприємства	
Виконано за тематикою НДР	
Виконано з макетом	
Виконано з застосуванням ЕОМ та МПТ	√
Має практичну цінність	√
Проект-частина комплексної теми	

Підпис рецензента _____
(_____)

РЕФЕРАТ

Текстова частина бакалаврської роботи: 56 стор., 20 рис., 4 табл., 18 дж.

Об'єкт дослідження – процес статичного дослідження прикладного програмного забезпечення.

Предмет дослідження – особливості дослідження метрик прикладного програмного забезпечення на мові java.

Мета роботи – дослідити статичний аназід метрики прикладного програмного забезпечення.

Методи дослідження. Для вирішення поставлених завдань були використані наступні методи: статичний, лексичний, синтаксичний і семантичний аналіз, метрики статичного дослідження в середовищі java.

Тенденції розвитку програмування наступного покоління ведуть до створення більш складніших в структурі і розміру ПЗ. Внаслідок чого необхідно дослідження і впровадження більш педантичного підходу до тесутвання нового продукуту та подальшого його супроводу і аналізу на помилки. І створення на їх основі, вихідних данних більш, досконалий продукт.

Вище сказане підводить нас до того, що потрібно розвиватися в напрямку статичного і динамічного аналізу ПЗ. Звичайному розробнику в наш час тяжко прослідкувати за правильність написання коду, правильність стилю, простою написання тому він користується спеціальними додатками, але вони поки що не досконалі.

На основі результатів виконаних досліджень розроблено web-додаток на мові java для статичного аналізу метрики програмного забезпечення.

СТАТИЧНИЙ АНАЛІЗ, СИНТАКСИЧНИЙ АНАЛІЗ, СЕМАНТИЧНИЙ АНЛІЗ, ЛЕКСИЧНИЙ АНАЛІЗ, ОПЕРАТОР, ОПЕРАНТ, БАЙТ-КОД,

МЕТРИКИ, КОМПІЛЯЦІЯ, ТОКЕН, ДИЗАСЕМБЛЕР, АССЕМБЛЕР, ФРЕЙМВОРК, JAR-ФАЙЛИ, РЕПОЗИТОРИЙ.

ЗМІСТ

Стор.

ВСТУП……..…10

1 СТАТИЧНИЙ АНАЛІЗ ВИХІДНОГО ТЕКСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ …….....11

2 СТАТИСТИЧНИЙ АНАЛІЗ МЕТРИК ПРОГРАМНОГО КОДУ…..…….....19

2.1 Вихідні дані для отримання метрик коду ……...……..20

2.2 Програма збору, аналізу і візуалізації метричних даних java-коду…….....22

2.3 Аналіз статистики байт-коду …….24

2.3.1 Метрики коду і 5 дані……..…….…….....24

2.3.2 Оцінка зібраних даних……...…...…….....26

2.3.3 Закони розподілу метрик……..…….....30

2.4 Оцінка стану проекту……..…….…….....33

2.4.1 Приклад аналізу. Бібліотека реактивного програмування ReCore……35

2.4.2 Приклад аналізу. Бібліотека середовища виконання програм в Android……..36

2.4.3 Приклад аналізу. стандартному середовищі реактивного програмування RxJava……...…37

3 ВЕБ-ДОДАТКИ І ВЕБ-СЕРВІСИ НА ПЛАТФОРМІ JAVA ……...39

3.1 Віртуальна машина Java …….…….....39

3.2 Інструменти і середовище Java……..…42

3.3 Конфігурація Eclipse і створення проекту ……...43

3.4 Багатосторінкові і односторінкові додатки……...44

3.5 JSP-сценарії……,……...……..46

3.6 Фреймворки для створення веб-додатків……..…….....47

3.7 Google Web Toolkit, HTML5, AJAX……..…….....48

3.8 Веб-додатки на платформі Java……..…….....49

4 ВИБІР МЕТРИК ДЛЯ WEB-ДОДАТКІВ В ПРОГРАМНОМУ СЕРЕДОВИЩІ JAVA……...…..51

ВИСНОВКИ……..54
ПЕРЕЛІК ПОСИЛАНЬ…….....55
Додаток А……...……....57
Додаток Б……...…….59
Додаток С……...……65
ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація)…….….68

ВСТУП

Показники програмного забезпечення відіграють важливу роль в управлінні професійними програмними проектами. Від їх якості залежать як характеристики програмного продукту на стадії програмування, так і перспективи його успішного супроводу. Багато експертів з наукових кіл, а також з промисловості погоджуються з тим, що більшість сучасних програмних продуктів та процес їх розробки мають порівняно низьку якість.

У звіті Standish Group-CHAOS за 2020 рік зазначено, що 24% проектів програмного забезпечення є неуспішними. Це означає, що вони або анулюються до завершення розробки проекту або завершуються, але ніколи не використовуються. Одним із факторів, що сприяють на неуспішність проектів є те, що сучасне програмне забезпечення майже ніколи не розробляється повністю з нуля, а досить розширене та модифікує існуючий код і часто включає сторонній вихідний код. Це може призвести до поганої загальної ремонтопридатності, складної розширюваності та великої складності програмного коду. Для кращого розуміння впливу змін коду та відстеження проблем зі складністю, в циклі розробки програмного забезпечення часто використовуються метрики якості програмного забезпечення. В ідеалі, метрики програмного забезпечення повинні постійно обчислюватися під час процесу розробки, щоб забезпечити найкраще можливе відстеження. Більш того, програмні показники повинні бути визначені командами розробників не лише для охоплення загальних факторів, а й для вимірювання конкретних цілей компанії, проекту чи групи. Метрики використовуються, наприклад, для відстеження прогресу розвитку, вимірювання впливу на реструктуризацію та оцінку якості. Вони є найбільш вигідними, якщо їх можна обчислювати безперервно під час розробки програмного продукту. У цій бакалаврській роботі представлено інтегрований та гнучкий підхід до метричних обчислень шляхом вбудовування його в статичний аналіз програм. Таким чином, показники можна обчислювати на вимогу для кожної компіляції навіть задовго до того, як програмне забезпечення буде повністю розроблено.

1 СТАТИЧНИЙ АНАЛІЗ ВИХІДНОГО ТЕКСТУ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Сутність статичного аналізу вихідного тексту програмного забезпечення, а також дизасембльованого тексту виконуваного коду полягає в перевірці семантичної і синтаксичної відповідності тексту програмного забезпечення певними правилами. Це перегляд вихідного коду розробником в тих місцях, де на погляд статичного аналізатора присутнє невірне оформлення або помилка.

Основним завданням статичного аналізу ПЗ, в загальному випадку, є відновлення алгоритму до початкового тексту програм. Надалі відновлений алгоритм використовується для верифікації програми і заявленим у програмній документації її специфікаціям (декларованим можливостям). На рис.1. представлено послідовність проведення статичного аналізу ПЗ.

Рисунок 1.1 – Послідовність проведення статичного аналізу ПЗ

Окремими завданнями статичного аналізу ПЗ є:

- перевірка модульної структури ПЗ, а також логічної структури окремих модулів і порівняння цих структур з наведеними в програмній документації;

- підготовка вихідних даних для проведення динамічного аналізу ПЗ і розробки плану тестування ПЗ;

- оцінка конструктивних характеристик програми, ступеня простоти модифікації і супроводу програми;

- визначення наявності недосконалостей в програмі, невикористовуваних ділянок програми, зайвих змінних;

- оцінка текстової складності програми, витрат на її розробку і освоєння;

- експертиза ідентичності програм для встановлення авторства і вирішення правових спорів;

- визначення кількісних характеристик при оцінці рівня якості програми.

Найпростіші форми статичного аналізу здійснюються компіляторами. Зазвичай набір виявлених ними помилок обмежений порушеннями синтаксису і правил сумісності типів. На іншому кінці спектру систем статичного аналізу знаходяться засоби автоматичного доведення правильності програм. Ці засоби вимагають детальної формальної специфікації властивостей програми, які треба довести, тому вони є вимогливими до ресурсів.

Проведення статичного аналізу «вручну» підготовленим експертом полягає в детальному перегляді вихідного тексту програм. З огляду на великі обсяги і складність побудови алгоритмічних конструкцій сучасних програмних засобів, застосування даного методу виконання статичного аналізу призводить до великих витрат часу і людських ресурсів і вимагає залучення експерта високої кваліфікації і як програміста, і як експерта якості. При цьому точність отриманих результатів, як правило, не виходить задовільною. У ряді випадків людські можливості просто не дозволяють провести статичний аналіз «вручну».

Автоматизовані інструментальні засоби здатні обчислити або хоча б допомогти у визначенні найбільш трудомістких характеристик, дозволяючи, таким чином, експерту зосередити увагу на інтерпретації отриманих характеристик в рівень якості програми.

Статичний аналіз програмного засобу передбачає отримання наступних характеристик (графічних і метричних):

- модульної структури програмного засобу (граф викликів, список шляхів викликів, матриця викликів і досяжності, точки викликів, метрики ієрархії викликів);

- логічної структури окремого програмного модуля (граф управління, шляхи тестування, метрики структури управління);

- характеристик тексту програми (статистичні дані про коментування програми, текстові метрики Холстеда).

Граф викликів відображає модульну структуру програмного засобу. Вершини графа зображені у вигляді прямокутників, що містять імена компонент (модулів), ребра - у вигляді стрілок.

Шлях виклику представляє послідовність дотичних ребер з графа викликів (кінцева вершина ребра є початковою вершиною наступного ребра), де початкова вершина першого ребра є корінь графа, а кінцева завжди являє собою компоненту, яка не міститься деревовидною структурою, тобто це лист дерева. У списку шляхів викликів представлені всі шляхи викликів, які підлягають тестуванню.

Матриця викликів і досяжності містить інформацію про два основних типах структур між довільними парами компонентів. Елементи в ієрархії викликів можуть перебувати в одній з наступних взаємозв'язків: одна з них безпосередньо викликає іншу або в графі викликів існує шлях, що починається на одному з елементів даної пари і закінчується на іншому, тобто елементи даної пари не можуть бути викликані одна з одної безпосередньо, а лише через ланцюжок послідовних викликів.

Матриця викликів і досяжності дозволяє відповісти на наступні питання:

- якщо змінити модуль А, то чи може це якось вплинути на модуль В;

- яке число модулів, що викликаються модулем А, і що це за модулі;

- яке число модулів, досяжних модулем А, але не викликаючих на нього, і що це за модулі;

- які модулі є недосяжними (ніколи не викликаються);

- які вершини графа є кінцевими (що не містять виклику).

Ця інформація використовується при тестуванні, супроводі та експлуатації програмних продуктів.

Таблиця точок викликів показує взаємозв'язок імен компонент, що викликаються, області дії виклику і координат точок викликів, що задаються номерами рядків і стовпців розташування операторів виклику. Область дії виклику може бути внутрішнім або зовнішнім залежно від того, чи була викликана компонента визначена в блоці компіляції чи ні.

Будь-яка програма може бути представлена у вигляді одного або декількох логічних блоків. Логічний блок представляє ділянку програми (послідовний набір операторів), що не має розгалужень, тобто при виконанні першого оператора блоку послідовно виконуються і інші оператори цього блоку. Граф управління програми представляє орієнтований граф, вершинами якого є логічні блоки, а ребрами - передача управління між блоками.

Оператори програми можуть виконуватися в різному порядку в залежності від обраного набору вхідних даних, тобто існує багато способів досягнення точки виходу з точки входу програми. Шлях тестування визначається як маршрут в графі управління, початкова вершина якого є вершиною графа, а кінцева вершина - вихідною вершиною графа. Програма може мати нескінченно велике число можливих шляхів тестування. В цьому випадку тестування зводиться до виконання деякого набору маршрутів, що покриває кожну гілку хоча б один раз. Перерахування шляхів тестування, які відповідають зазначеним вище вимогам, наводиться у вигляді граф-схеми, де вершини, що представляють виконані один за одним блоки, з'єднані стрілками, які позначають порядок виконання.

В цілому повний процес аналізу ПЗ включає три основних види дослідження вихідних текстів:

1. Лексичний аналіз, що полягає в пошуку лексем елементів НДВ (в тому числі в шістнадцятирічному поданні), званих сигнатурою елементів НДВ.

Лексичний аналіз поверхнево розглядає вихідний код програми і розбиває його на відповідні маркери (tokens). Токен - це значуща частина вихідного коду. Приклади токенів включають ключові слова, пунктуаційні знаки та літерали -

такі, як числа і рядки. Токени включають пробіли, які зазвичай ігноруються, але використовуються для поділу токенів, і коментарі.

При проведенні аналізу програмного засобу на відсутність НДВ здійснюється пошук сигнатур (токенів) наступних класів: сигнатур елементів НДВ, сигнатур «підозрілих функцій» і сигнатури штатних процедур використання системних ресурсів і зовнішніх пристроїв.

Пошук сигнатур реалізується за допомогою спеціальних програм-сканерів.

2. Синтаксичний аналіз, який передбачає пошук, розпізнавання і класифікацію синтаксичних структур НДВ, а також побудову структурно-алгоритмічної моделі самої програми.

Синтаксичний аналіз полягає в пошуку алгоритмічних образів основних і додаткових елементів НДВ відповідно до принципів концептуального уявлення НДВ. Під час синтаксичного аналізу витягується значення з вихідного коду для перевірки синтаксичної правильності програми (власне синтаксичний аналіз) і побудови її внутрішнього уявлення (синтаксично-структурний або структурний аналіз). Рішення задач пошуку і розпізнавання синтаксичних структур НДВ має самостійне значення для верифікаційного аналізу програм, оскільки дозволяє здійснювати пошук елементів НДВ, які не мають сигнатури.

Структурно-алгоритмічна модель програми необхідна для реалізації такого вигляду аналізу - семантичного.

3. Семантичний аналіз, що полягає в пошуку НДВ на основі знань способів і методів організації віртуальних середовищ, що зберігаються в базі даних.

Семантичний аналіз передбачає дослідження програми вивчення сенсу складових її функцій (процедур) в аспекті операційного середовища комп'ютерної системи. На відміну від попередніх видів аналізу, заснованих на статичному дослідженні, семантичний аналіз націлений на вивчення динаміки програми - її взаємодії з навколишнім середовищем. Процес дослідження здійснюється у віртуальному операційному середовищі з повним контролем дій програми і відстеженням алгоритму її роботи по структурно-алгоритмічній моделі. Семантичний аналіз є найбільш ефективним видом аналізу, але і найбільш

трудомістким. З цієї причини доцільно поєднувати три перерахованих вище види аналізу. Розроблені критерії дозволяють розумно поєднувати різні види аналізу, істотно скорочуючи час дослідження, не знижуючи його якості.

Кожен з видів аналізу представляє закінчене дослідження ПО відповідно до спеціалізації. Результати дослідження можуть мати як самостійне значення, так і корелюватися з результатами повного процесу аналізу.

Додатково до приватних методів статичного аналізу також відносяться аналіз покажчиків, пошук і усунення мертвого коду, мінімізація кількості змінних. Додаткове застосування метрик складності дозволяє проводити статичний аналіз програм за відсутності семантичного означення використовуваних функцій і процедур.

Статичний аналіз початкового програмного коду передбачає виконання таких основних процедур:

- попередній контроль якості розробки програми на основі кількісного оцінювання обраних метрик складності;
- контроль повноти і відсутності надмірності вихідних текстів ПО на рівні функціональних об'єктів (процедур);
- контроль відповідності вихідних текстів ПО його об'єктного (завантажувального) коду;
- контроль повноти і відсутності надмірності вихідних текстів ПО на рівні файлів;
- контроль зв'язків функціональних об'єктів (модулів, процедур, функцій) з управління;
- контроль зв'язків функціональних об'єктів (модулів, процедур, функцій) за інформацією;
- контроль об'єктів різних типів (глобальних, локальних, зовнішніх змінних і т.п.);
- формування переліку маршрутів виконання функціональних об'єктів (процедур, функцій).

Вихідними даними для такого аналізу є програмна документація, вихідні тексти програм, бібліотечне оточення програм, середовище розробки та середовище застосування ПЗ.

До основних етапів статичного аналізу відносяться:

- препроцесорна обробка програми стандартними засобами компілятора;
- лексичний і синтаксичний аналіз;
- формування бази даних про програмні об'єкти;
- формування керуючого графа програми;
- формування візуалізацій компонент ПЗ;
- формування запитів до бази даних програмних об'єктів і навігації по візуалізацій;
- формування звіту статичного аналізатора.

Додатково виконуються процедури, що дозволяють об'єднати якісні і кількісні показники статичного аналізу:

- аналіз виконуваних файлів і бібліотечного оточення на відповідність списку функцій в заголовку і наявних у файлі;
- формування переліку дефектів програмного коду, що дають судження про наявність програмних закладок;
- формування повного графа з управління, розпізнавання керуючих конструкцій;
- розрахунок метрик оцінки програмного засобу, отримання метричної оцінки керуючого графа і кожної функції (рівень мови, рівень програми, цикломатичне число і ін.);
- застосування процедури інтерпретації команд мови асемблер (якщо необхідно) для отримання графа за інформацією, що полягає у виділенні ділянок коду, що не беруть участь у формуванні графа за інформацією, і виділення нетипових ланцюжків команд мови асемблера, підозрілих на НДВ.

Алгоритм роботи системи проведення статичного аналізу вихідних текстів ПО на основі розглянутих методів і процедур представлений на рис. 1. В якості

потенційних недекларованих можливостей програми розглядаються виявлені при роботі системи статичного аналізу потенційно небезпечні фрагменти програми, що визначаються формальними засобами аналізу структурних компонентів керуючих графів модулів ПЗ. На рис.1.2 представлено схему алгоритму системи статичного аналізу.

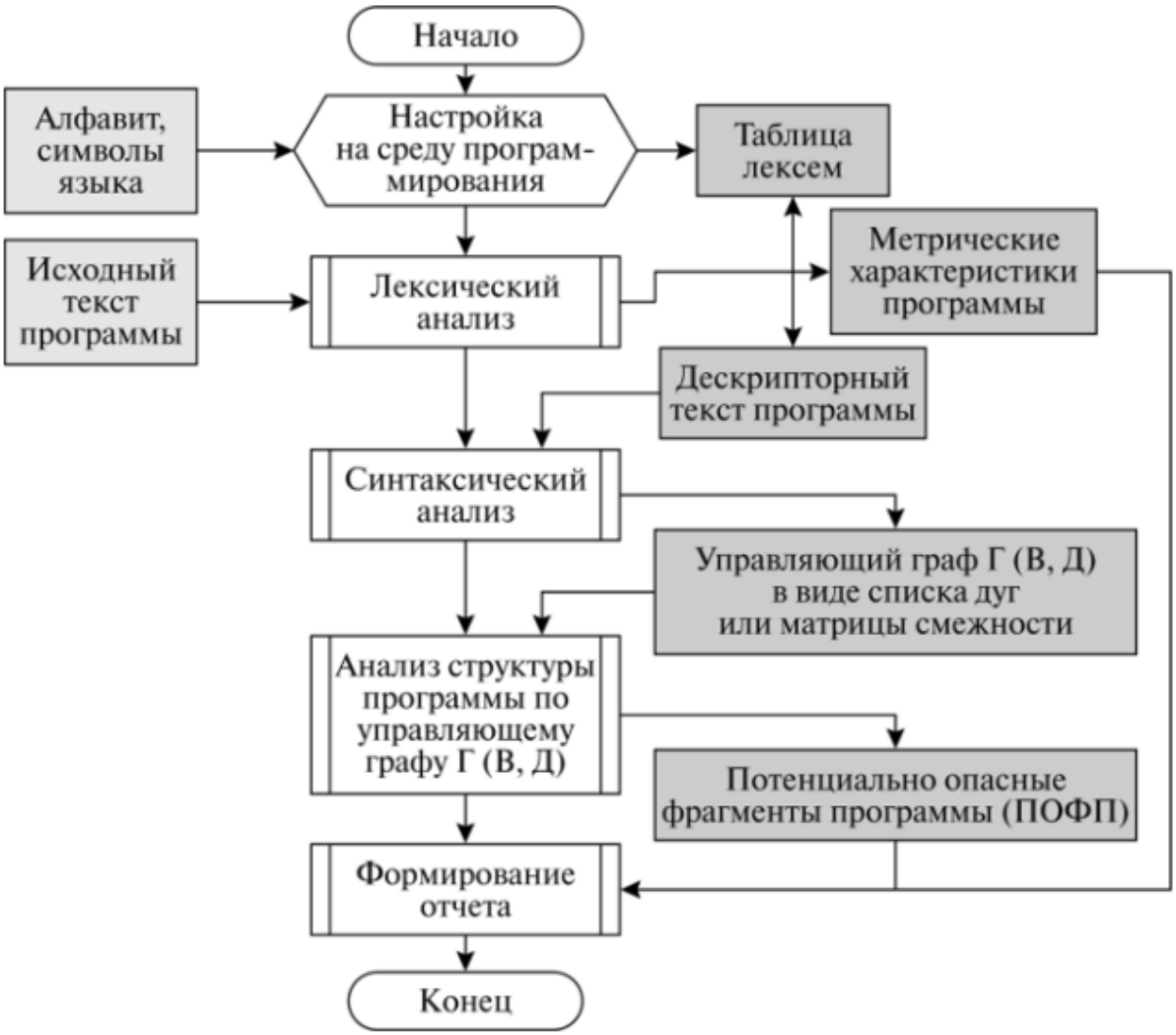


Рисунок 1.2 – Схема алгоритму системи статичного аналізу

2 СТАТИСТИЧНИЙ АНАЛІЗ МЕТРИК ПРОГРАМНОГО КОДУ

Загальновизнаний набір метрик в даний час «канонізований» і не зазнає істотних змін. Існують онлайн- і офлайн-системи документування, супроводження та отримання метричних характеристик проекту, а також численні плагіни в інтегрованих середовищах розробки. На самій простій кількісній метриці - кількості рядків коду вихідного тексту (SLOC) - заснована найбільш популярна методика оцінки трудомісткості розробки програмного забезпечення COCOMO .

Підтримка якості коду не гарантує його експлуатаційних властивостей - надійності, стійкості, ймовірності помилок: перераховані властивості забезпечуються тестуванням коду і артефактів проектування. Однак при інших рівних умовах якість коду також сприяє цього. Іншими словами, неякісний код може бути як завгодно «хорошим», але для розробки «хорошого» коду бажано підтримувати його якість.

Іншим фактором не на користь метрик є відсутність рамкових законів, що дозволяють оцінити або розрахувати фундаментальні властивості програмного продукту на основі його коду. Для порівняння: в мостобудуванні конструкція моста може бути як завгодно оригінальною, але вона розраховується за правилами і законами опору матеріалів.

При існуючій ситуації отримання метричних характеристик якості коду переслідує такі цілі:

- підтримка стандартів коду при колективному володінні кодом;
- виявлення потенційно небезпечного коду;
- виявлення вузьких місць в структурі коду;
- моніторинг розроблюваного проекту в системі контролю якості при наявності статистики, розгорнутої в часі.

На практиці в основному переважає прагматичний підхід: використовуються засоби підтримки стандартів кодування - однакова стилістика коду і прості кількісні метрики.

Самі по собі метричні характеристики нічого не дають. Їх необхідно якось інтерпретувати, як мінімум, в порівнянні з наявною статистикою. Тут в програмній інженерії склалася парадоксальна ситуація. З одного боку, нас оточує величезна кількість загальнодоступного програмного коду, як в вихідних текстах, так і в проміжних платформонезалежних форматах (байт-код віртуальної машини Java - JVM). З іншого боку, загальнодоступна статистика його метрик відсутня. Така статистика за основними показниками якості програмного коду, нехай навіть якості формальної, може бути корисна, перш за все, як основа галузевої статистики, щодо якої можна буде проводити дослідження і оцінки якості окремих проектів. Поки що така оцінка виглядає як «річ у собі» - отримані метрики немає з чим порівнювати, можна тільки аналізувати тимчасовий тренд отриманих показників.

Левова частка загальнодоступного проміжного коду представлена байт-кодом JVM, який є платформо-незалежним стандартом де-факто. Засоби розподіленої збірки проектів і доступу до артефактів дозволяють формалізувати процес отримання необхідного статистичного матеріалу. На сьогоднішній день один лише репозиторій maven зберігає порядку 107 посилань на артефакти (jar-файли). Основним недоліком тут є схильність результатів вимірювання метрик оптимізаціям компілятора.

2.1 Вихідні дані для отримання метрик коду

«Вихідний матеріал» для отримання метрик доступний в трьох формах: вихідні тексти програм, проміжний код і архітектурно-залежний машинний код.

Вихідні тексти програм дозволяють отримати найбільш точні показники метрик. Статистичне дослідження є репрезентативним в рамках конкретної мови програмування, а вибірка представляє собою безліч програм з відкритим

вихідним текстом, реалізованих на даному мовою. У разі аналізу різних мов програмування це - єдиний доступний підхід. Дослідження машинного коду неможливо без прив'язки до апаратного середовища. В силу свого низького рівня, воно обмежене в наборі метрик - так, неможливо провести аналіз метрик об'єктноорієнтованого коду (об'єкти є абстракцією високого рівня). Вибірка репрезентативна для програм, скомпільованих для певного набору процесорних інструкцій. У доповненні до традиційних метрик тут можна оцінити час виконання програм і обсяг споживання апаратних ресурсів.

В байт-коді JVM зберігається інформація про сутність і відношення високого рівня – таких як «Клас», «Метод», «Поле», «Виклик методу», і інших, що дозволяє виміряти показники метрик об'єктно-орієнтованого коду. Більша частина оптимізації виконується під час виконання програми, що дозволяє уникнути перекручування показників метрик. Вибірка є платформо-незалежною, репрезентативною для сімейства мов, компільованих в байт-код JVM (понад 200 мов), а також володіє найбільшим об'ємом в порівнянні з двома іншими підходами. На нижньому рівні системи збору та аналізу статистики необхідні засоби, що дозволяють аналізувати байт-код і отримувати необхідні метрики. В таблиці 2.1 представлений список таких програмних засобів.

Таблиця 2.1 − ПЗ аналізу метрик байт-коду

Назва	Ліцензія	Рік	Масовий аналіз
Dependency Finder	Вільна (BSD License 2.0)	2010	Так
JDepend	Вільна (BSD License)	2005	Так
CyVis	Вільна (GNU General Public License)	2006	Ні
Ckjm	Вільна (Apache License)	2014	Ні
Jqassistant	Вільна (GPLv3 License)	2018	Ні

Продовження таблиці 2.1 − ПЗ аналізу метрик байт-коду

JArchitect	пропріетарна	2018	Так
Jtest	пропріетарна	2018	Так
Sonargraph	пропріетарна	2018	Так

Представлені в списку рішення не задовольняють необхідного набору вимог: наявність вільної ліцензії, актуальна підтримка, можливість масового дослідження коду програми - одночасний аналіз групи класів пакету або архіву. Тому була проведена розробка оригінальної бібліотеки збору метрик байт-коду на основі фреймворку ASM, який здійснює розбір class-файлів через програмний інтерфейс, який реалізує патерн «відвідувач».

2.2 Програма збору, аналізу і візуалізації метричних даних java-коду

Програма статистичного аналізу метрик байт-коду (рис.2.1-2.2) дає необхідний мінімум засобів для збору, статистичної обробки та розвідувального аналізу даних - метрик програмного коду:

- збір основних метричних характеристик байт-коду з проектів і бібліотек, представлених в вигляді jar-файлів;
- підтримка вибірок jar-файлів в однойменних каталогах;
- сканування і завантаження jar-файлів з Maven Репозіторія за категоріями, які вводяться в вигляді ключових слів пошуку на сайті репозіторія;
- збір вихідних даних метрик як по окремим проектам/бібліотекам, так і по категорії в цілому;
- генерація представницьких вибірок по вихідній множині значень метрики;
- експорт основних характеристик законів розподілу і даних гістограм зібраних метрик в текстові файли;
- візуалізація гістограм розподілів метрик.

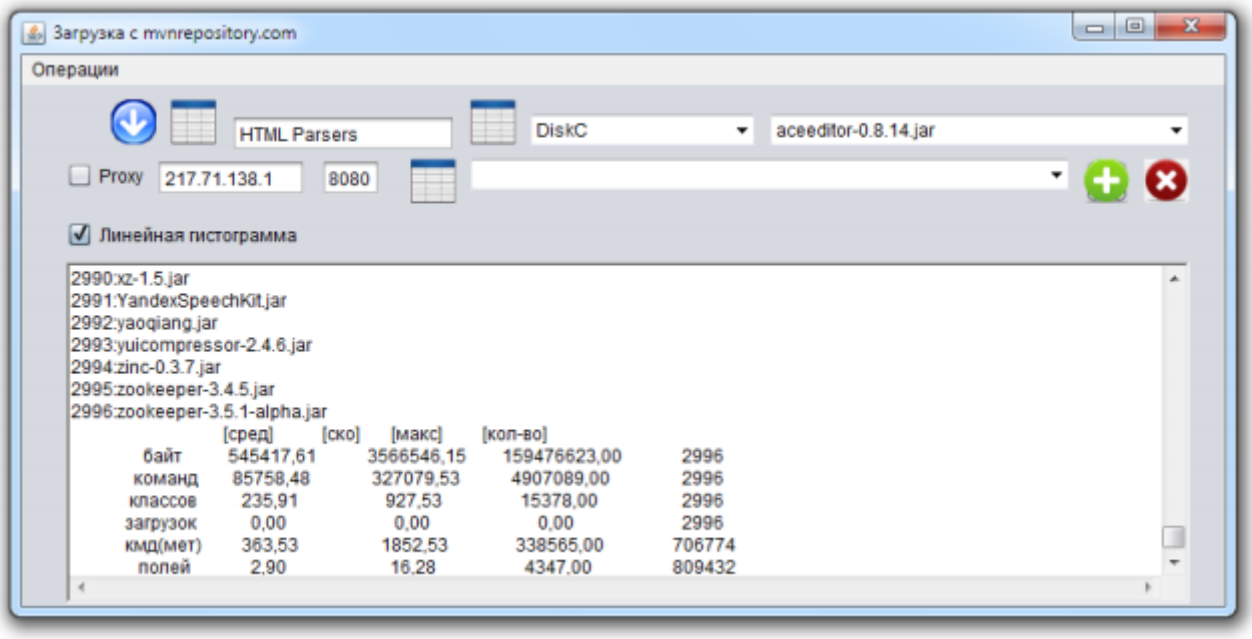


Рисунок 2.1 – Зовнішній вигляд програми

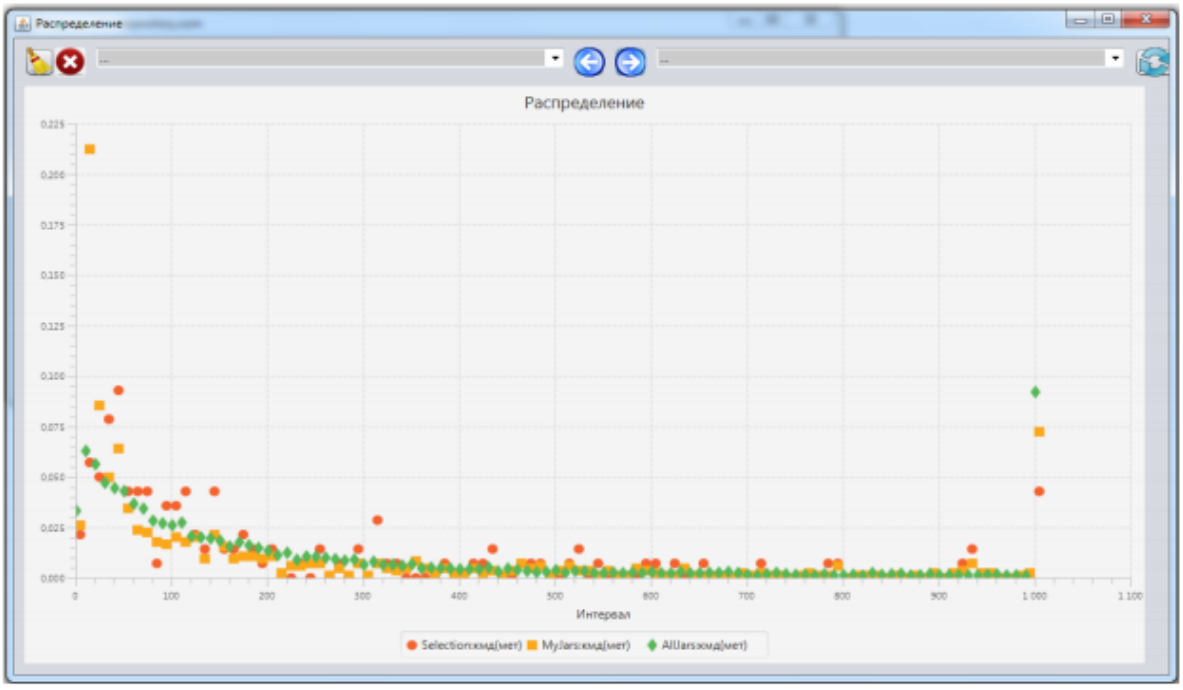


Рисунок 2.2 – Гістограми статистик метрики для різних категорій

2.3 Аналіз статистики байт-коду

Для практичного використання статистики даних по метриках байт-коду необхідно вирішити наступні завдання:

- визначити, наскільки обсяг вихідних даних дозволяє судити про достовірність імовірнісних характеристик метрик – законів розподілу і їх основних параметрів;

- провести детальний аналіз кожної метрики. Визначити види законів розподілу даних метрики;

- розробити методику оцінки стану програмного проекту на основі порівняння метрик проекту зі статистикою метрик.

2.3.1 Метрики коду і вихідні дані

Оскільки байт-код є об'єктно орієнтованим, статистичною одиницею вимірювання метрики як випадкової величини є клас.

Дослідження проводилося для метрик двох видів. До першого належать метрики байт-коду і проекту, що не належать до канонічно наборам. Деякі з них мають відношення до проекту (jar-файлу) в цілому. Всі вони мають оригінальні абревіатури:

- розмірність jar-файлу проекту в байтах – JFS (JarFile Size);
- кількість команд байт-коду в jar-файлі (проекті) - JCS (Jar Code Size);
- кількість завантажень проекту зі репозиторію - NoDL (Number of DownLoads);
- кількість класів - JNoC (Jar - Number of Classes);
- кількість полів в класі - NoF (Number of Filelds);
- кількість методів в класі - NoM (Number of Methods);
- загальна кількість полів і методів класу - NoFM (Number of Fields and Methods);
- довжина байт-коду методу, кількість команд в методі - MCS (Method Code Size).

Наступні метрики з груп Чідамбера / Кемерера і Мартіна є стандартними і мають загальноприйняті позначення:

- доцентрове зчеплення Ca (Afferent Couplings) - кількість класів поза категорії (пакета), що залежать від класів цієї категорії;
- відцентрове зчеплення Ce (Efferent Couplings) - кількість класів всередині категорії (пакета), які залежать від класів поза цієї категорії:
- нестабільність I (Instability), I = Ce / (Ca + Ce);
- висота дерева наслідування - DIT (Depth of Inheritance Tree) - максимальна довжина шляху (кількість вершин - класів) по дереву наслідування;
- кількість нащадків класу - NOC (Number of Children) - середня кількість прямих наслідувань класу;
- зчеплення між класами - CBO (Coupling Between Object Classes) - загальна кількість викликів методів і використання властивостей об'єктів інших класів в коді класу;
- зважена сумарна вага методів в класі - WMC (Weighted Methods per Class) – сума значень цикломатичної складності методів в класі.

В якості вихідних даних використовувалися набори jar-файлів, розбиті на групи залежно від мети аналізу і «походження»:

Група наборів великого обсягу (група 1):

- всі доступні файли, викачані з репозиторіїв, а також ті, що знаходяться на системному диску в різних середовищах розробки (Android, Java EE, Java SE, JDK, Scala, NetBeans, IntelJJ IDEA) - DiskC (2996 проектів, 806525 класів);
- jar-файли великого розміру (більше 6.8 Мб) з першого набору;
- jar-файли малого розміру (менше 1 Мб) з першого набору.

Група наборів по предметній області програмування, викачаних з репозиторію Maven [5] - HTTP (категорії HTTP Clients, HTTP Parser), JDBC, JVM (категорії JVM Libraries, ByteCode Analysers), JSON, Audio (група 2).

Окремі jar-файли великого розміру поширених засобів розробки (Android, RxJava), набір jar-файлів власних проектів кафедри (MyJars), окремий файл власного проекту (BRSCore) (група 3).

2.3.2 Оцінка зібраних даних

Вже зовнішній вигляд гістограм розподілу значень метрики MCS, обраної в якості зразка, дозволяє судити про якісну різницю зібраних даних для наборів різних груп (рис.2.3-2.5)

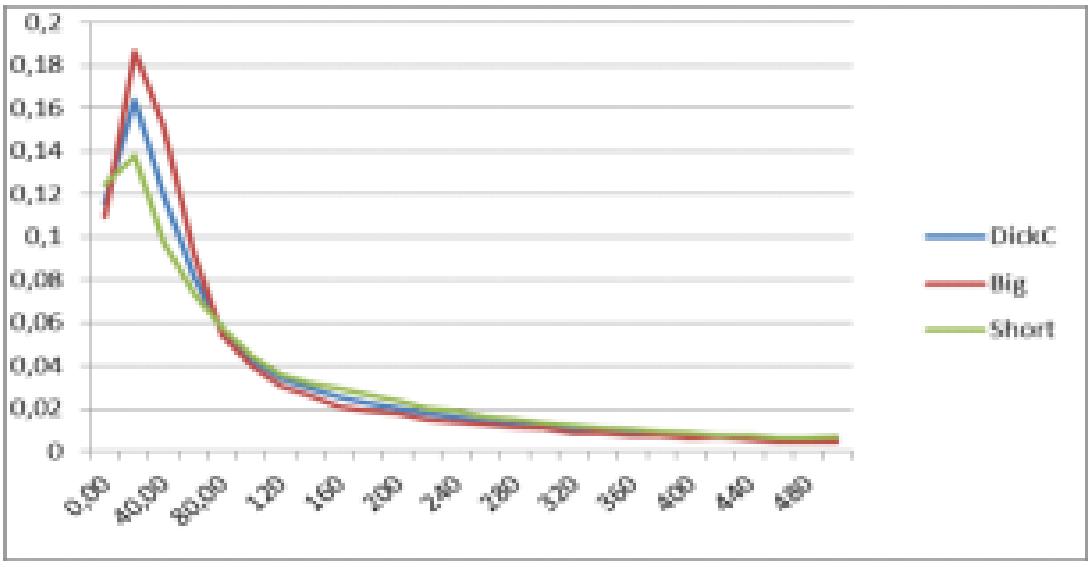


Рисунок 2.3 – Розподіл метрики MCS для наборів групи 1

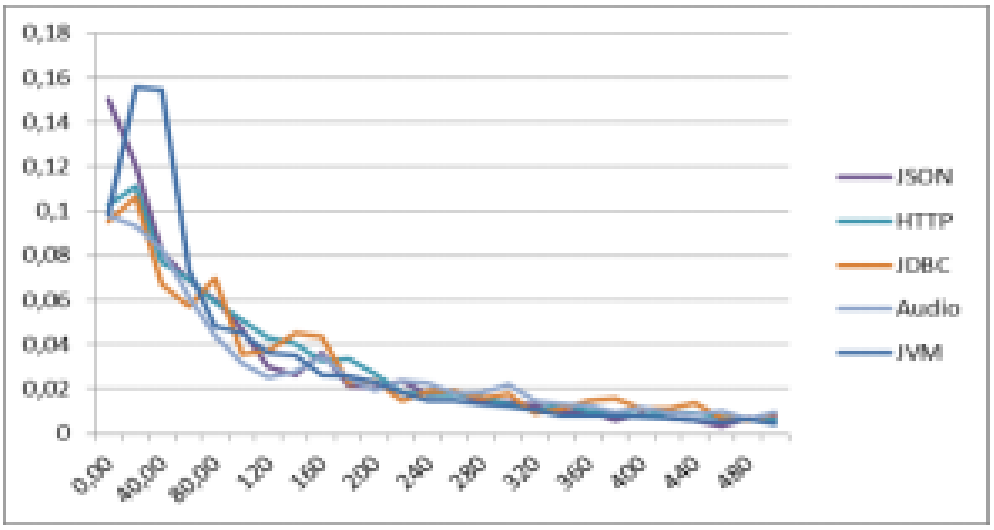


Рисунок 2.4 – Розподіл метрики MCS для наборів групи 2

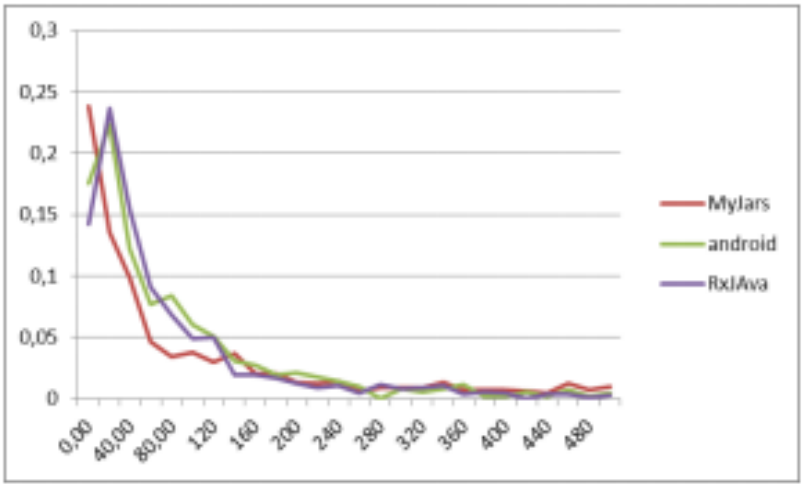


Рисунок 2.5– Розподіл метрики MCS для наборів групи 3

Визначення відповідності вибірки закону розподілу є завданням статистичної перевірки гіпотези про її приналежності цього розподілу. Для вирішення цієї та подібних завдань використовувався пакет статистичного аналізу EasyFit.

Оскільки вид більшості розподілів виглядає як зворотньо експонентний, «хвіст» розподілу створює статистичний «шум» на малих частотах появи відповідних значень метрики. Чинний діапазон значень метрики (рис.2.6) скорочується до розміру, при якому значення критерію оцінки достовірності перевищує теоретичне значення для обраного рівня значущості (УЗ) гіпотези α, тобто закон стійко визначається. У нашому випадку використовується найбільш поширений критерій Колмогорова-Смирнова.

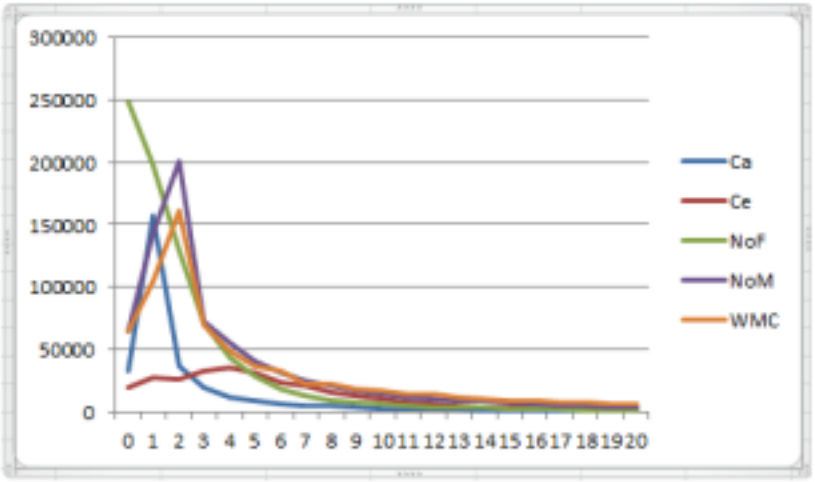


Рисунок 2.6 – Розподіл метрик в діапазоні 0…20

У той же час скорочення діапазону збільшує похибку визначення параметрів розподілу. У таблиці 2.2 для метрики NoM показано, як параметр експоненціального розподілу λ залежить від зміни діапазону.

Таблиця 2.2 − Оцінка діапазону значень метрики NoM, що аналізуються

λ	0,342	0,251	0,217	0,199	0,186
Критерій *	0,196	0,148	0,018	0,193	0,204
P-Value	0,756	0,695	0,243	0,077	0,02
Відкинуто	-	-	-	-	+
Критерій **	0,391	0,287	0,234	0,205	0,187
Діапазон	11	21	31	41	51

Критерій * - розрахункове значення критичної статистики для критерію Колмогорова-Смирнова

Критерій ** - теоретичне значення критичної статистики при УЗ = 0,05

Має місце природна межа діапазону значень метрики, що враховуються в законі розподілу. Вона пов'язана з умовою прийняття гіпотези - значення критерію не перевищує критичне значення, встановлене для обраного рівня значущості.

Хоча більшість метрик мають цілочисельні значення, закони розподілу визначаються для безперервних випадкових величин з огляду на різноманітності останніх.

На рис.2.7 зображено залежність значення критерію від розмірності вибірки-значень метрики в наборі (кількості класів, для яких визначено MCS). Ось X - десятичний порядок розмірності вибірки.

У таблиці 2.3 наведені результати визначення пакетом EasyFit законів розподілу для різних наборів для статистичних даних метрики MCS - кількість команд байт-коду в методі.

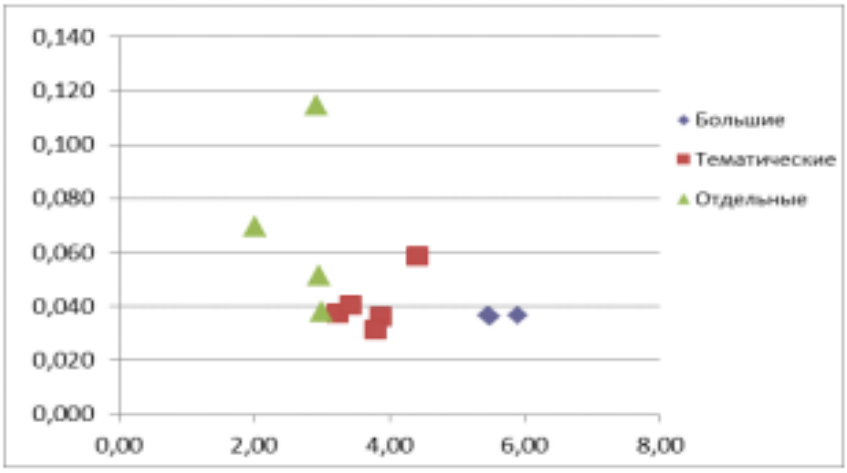


Рисунок 2.7 – Значення критерію Колмогорова-Смирнова залежно від порядку вибірки

Таблиця 2.3 − Закони розподілу для метрики MCS

Метрика MCS	DiskC	Big	Short	JSON	JVM	HTTP	Audio	JDBC	MyJars	Android	RxJava	BRSCore
μ статистики	364	337	364	389	443	427	**840**	548	327	**106**	142	546
σ статистики	1856	2035	1333	1170	2932	1570	**3541**	1385	856	**166**	342	1340
Классов (NC)	806525	313208	286362	7532	26380	6171	1723	2666	843	985	931	103
Log10(NC)	5,91	5,50	5,46	3,88	4,42	3,79	3,24	3,43	2,93	2,99	2,97	2,01
Критерий*	0,037	0,036	0,036	0,036	0,059	0,032	0,037	0,041	0,115	0,038	0,051	0,070
P-Value	0,993	0,995	0,994	0,995	0,752	0,999	0,993	0,986	0,067	0,997	0,846	0,904
Критерий**	0,121	0,121	0,121	0,121	0,121	0,121	0,121	0,121	0,121	0,134	0,116	0,170
Распределение	1	1	2	2	1	3	2	3	4	2	1	4
σ	1,14	1,08	1,20	1,17	1,06	1,15	**1,16**	1,13	1,13	**1,15**	1,12	1,14
μ	4,15	4,09	4,21	4,26	4,22	4,36	**4,5**	4,49	4,26	**3,97**	4,01	4,28
Jar-файлов	3356	53	3082	132	64	51	**46**	291	3	**1**	1	1

Розподіл: 1-логнормальний, 2-Джонсона, 3-експоненціальне, 4-Парето

Більшість наборів, в тому числі що складаються з окремих jar-файлів, дозволяють «встановити» закон розподілу з урахуванням імовірнісного характеру проведеної процедури. Значення критерію лежать в вузькому діапазоні для всіх наборів груп 1 і 2.

У той же час основні параметри статистики (середнє і середньоквадратичне відхилення) змінюються в широких межах на різних наборах (рис. 2.8). Одночасно змінюються і параметри закону розподілу. (σ і μ для логнормального розподілу - в табл.2.3). Між тими і іншими спостерігається природна кореляція.

«Викиди» параметрів для тематичних вибірок і окремих jar-файлів цілком можуть бути пояснені специфікою предметної області.

Наприклад, мале середнє MCS для Android і RxJava - переважанням коротких методів при високому рівні модульності і абстрагування в коді, а значне середнє в Audio - наявністю об'ємних методів обробки аудіо-даних.

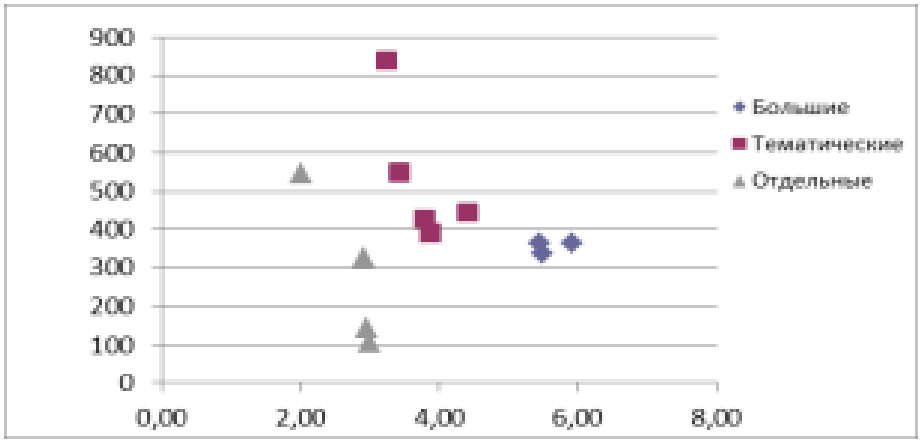


Рисунок 2.8 – Середнє значення метрики MCS на різних наборах

Таким чином, тільки набори групи 1 можуть бути використані в якості статистичного матеріалу для отримання статистичних параметрів і законів розподілу метрик байт-коду. Розмірність вихідних даних повинна бути не менше 10^5 класів.

2.3.3 Закони розподілу метрик

Як було показано, статистичний аналіз метрик проводиться на вибірці групи 1 (DiskC). Дані аналізу метрик наведені в таблиці 2.4.

Таблиця 2.4 − Підсумкові параметри оцінки законів розподілу для різних метрик

Метрика	MCS	WMC	CBO	DIT	NoC	NoF	NoM	I	NoFM	Ce	Ca	JNoC	JCS	JFS
Критерий*	0,037	0,128	0,07	0,262	0,069	0,240	0,148	0,120	0,084	0,081	0,024	0,047	0,069	0,041
P-Value	0,993	0,839	0,94	0,215	0,999	0,151	0,695	0,418	0,996	0,997	0,130	0,973	0,984	0,999
Критерий**	0,121	0,287	0,19	0,338	0,287	0,287	0,287	0,187	0,287	0,287	0,28	0,134	0,21	0,154
Распределение	4	2	4	3	7	3	6	1	2	5	3	6	5	2
Ячеек гистограммы	126	21	51	15	21	21	21	51	21	21	22	100	40	75
Шаг выборки	4	1	1	1	1	1	1	0	1	1	1	1	400	2000
Диапазон	504	21	51	15	21	21	21	1,02	21	21	22	100	16000	2E+05
Ранг распределения	1	1	1	1	1	1	4	2	1	1	1	2	1	3

Розподіл: 1-бета, 2-гамма, 3-Гумбеля, 4-логнормальний, 5-Джонсона, 6-експоненціальне, 7- статичне.

Результати можна згрупувати наступним чином. До першої групи належить велика частина метрик (MCS, WMC, CBO, NoM, NoFM, Ce, Ca, JFS), яка має характерний вигляд кривої розподілу (рис. 2.9)

Для більшості метрик характер такого розподілу можна легко пояснити. Наприклад, для сумарної цикломатичної складності (рис.2.10) перший відлік відповідає часто зустрічаючих класів (інтерфейсів) з порожніми методами, класи з лінійним кодом в методах також зустрічаються часто і мають малі значення метрики.

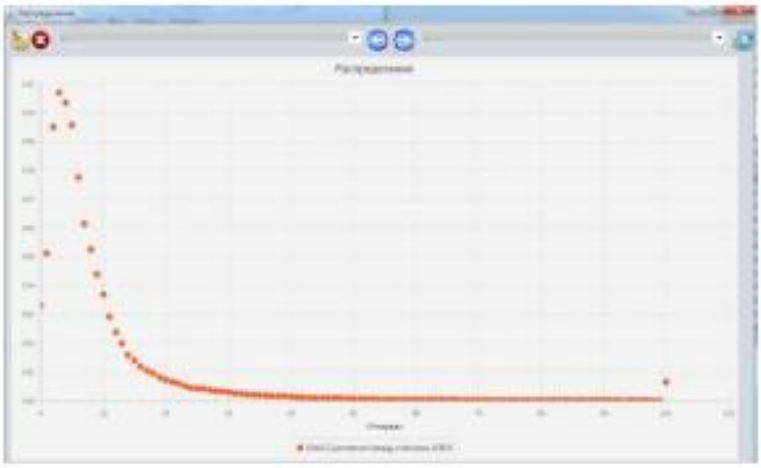


Рисунок 2.9 – Вид емпіричної кривої розподілу першої групи (CBO)

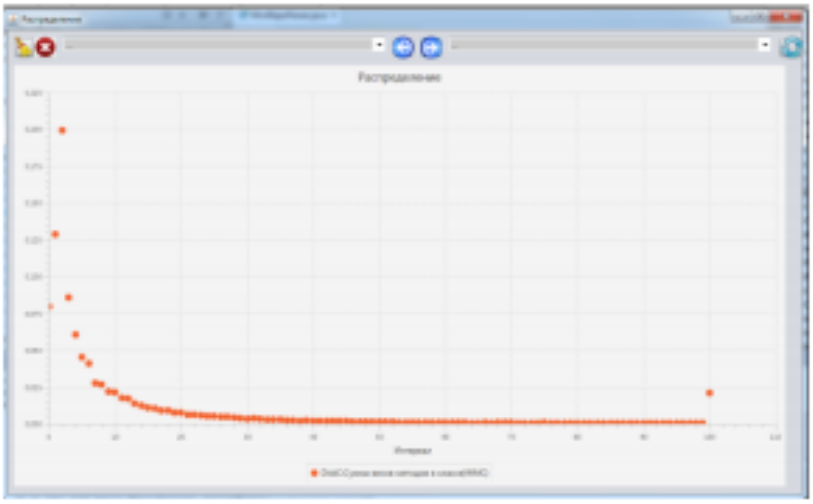


Рисунок 2.10 – Статистика метрики - сумарна складність коду класу (WMC)

Характерний регресивний вигляд графіка відповідає загальним принципам модульного програмування і ООП – переважне використання «коротких» і простих сутностей (класів, методів) перед складними і об'ємними. В даному випадку статистика підтверджує, що розробники в своїй основній масі слідують цьому правилу.

Друга група метрик (DIT, JCS, JNoC, NoC, NoF) відрізняється відсутністю підйому в початковій точці кривої розподілу (NoF на рис.2.11). Для зазначеної метрики це обумовлено великою кількістю класів без даних - інтерфейсів, класів зі статичними методами, а також класів з мінімальною кількістю даних (полів).

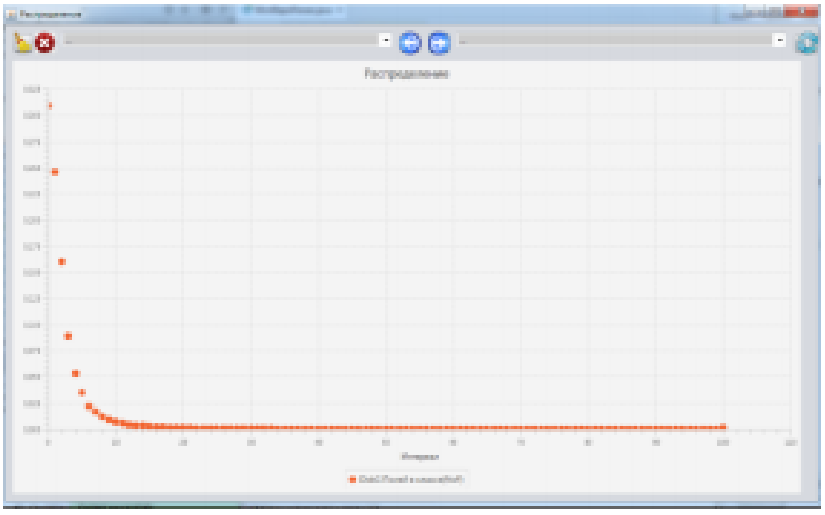


Рисунок 2.11 – Статистика метрики - кількість полів в класі (NoF)

Статистика метрики «нестабільність» має особливий вид розподілу (рис.2.12) пояснюється тим, що метрика обчислюється як I = Ce / (Ca + Ce), що при великій частоті малих значень Ca і Ce дає викиди на значеннях 0.25, 0.33, 0.4, 0.5 тощо.

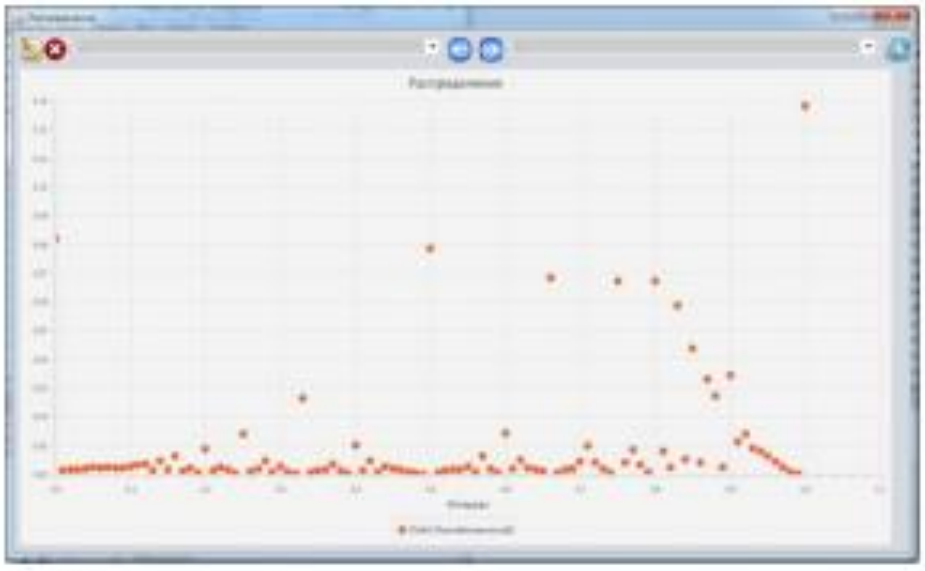


Рисунок 2.12 – Статистика метрики - нестабільність (I)

2.4 Оцінка стану проекту

Статистичні дані для оцінки стану проекту можна використовувати формально, необхідний змістовний аналіз специфіки проекту. Це пояснюється самою природою метрик програмного коду: самі по собі значення метрик не є показником якості проекту. Наявність достовірної статистики дає додаткові підстави для інтерпретації і оцінки результатів вимірювань. Можливо кілька типових варіантів оцінки розбіжності метрик і статистики:

- відхилення окремих метрик може пояснюватися специфікою проекту, це пояснення знайдено і правдоподібно;

- масове відхилення метрик може свідчити про низьку якість проекту, відхилення окремих метрик можуть бути оцінені як відступ від принципів технології ООП;

- відхилення окремих метрик обумовлено самою природою статистичного аналізу і не є показником якості проекту.

Розглянемо кілька варіантів аналізу конкретних проектів (jar-файлів). Порівняння будемо проводити за основними параметрами статистик (МО, СКО) метрик проекту і вибірки великого обсягу DiscC. Значення метрик аналізованого проекту нормовані до відповідних значень метрик в вибірці. Для нормованих значень використовуються ті ж самі абревіатури, що і для вихідних метрик.

Приклад аналізу. Бібліотека ядра системи обліку рейтингу успішності BRSCore

Бібліотека (власний проект кафедри) містить загальний програмний код клієнт-серверних додатків для системи обліку рейтингу успішності - рівні доступу до даних, бізнес-рівень, комунікації, контролер уявлень (екранних форм). Нормовані метрики проекту (рис.2.13) мають МО в межах 0.5-1.5 від статистичних, що вимагає пояснення.

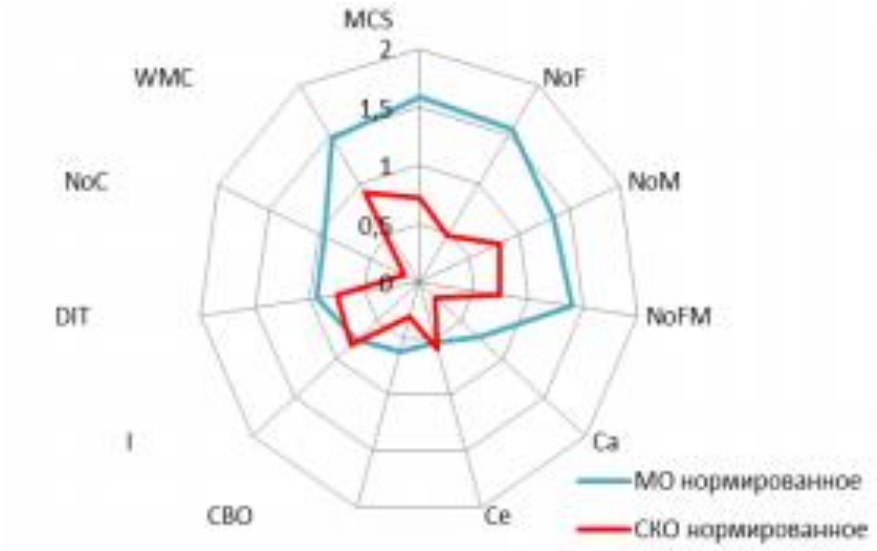


Рисунок 2.13 – Нормовані метрики проекту бібліотеки BRSCore

Значні перевищення нормованих значень довжини коду методу (MCS = 1.58), сумарної складності методів класів (WMC = 1.46), кількості полів (NoF = 1.56) і методів (NoM = 1.33) свідчать про нестрогому дотриманні принципів ООП («роздуті» класи, відсутність необхідних абстракцій). Цим же можуть

пояснюватися занижені значення зчеплення (Ca, Ce, CBO) - методи класів «самодостатні», кожен використовує власні засоби.

2.4.1 Приклад аналізу. Бібліотека реактивного програмування ReCore

Бібліотека середовища реактивного програмування ReCore не містить елементів інтерфейсу користувача, реалізує великий набір внутрішніх абстракцій (реактивні типи даних, класи), кожна абстракція має досить велику кількість реалізацій-наслідувачів. Бібліотека має розмірність 0.4 від середньостатистичної. Нормовані метрики проекту (рис. 2.14) мають МО в межах 0.8-1.2 від статистичних, що свідчить про відсутність якихось явних вад.

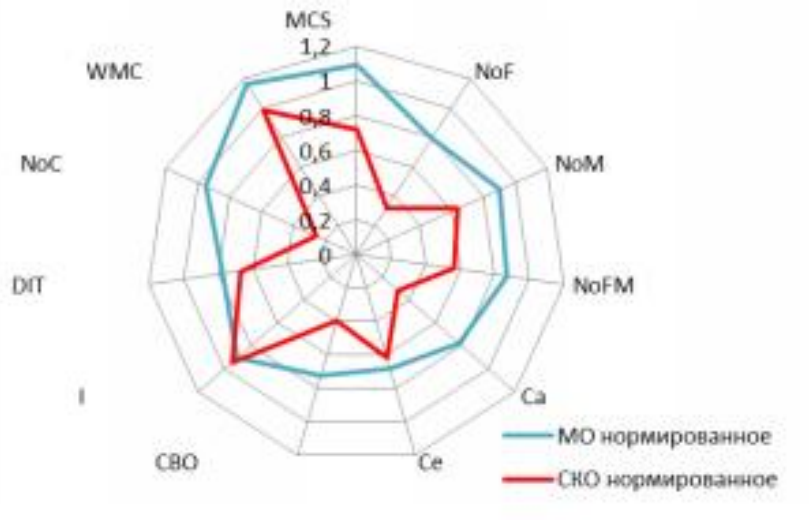


Рисунок 2.14 – Нормовані метрики проекту ReCore

Для відхилень нормованого СКО проекту можна знайти розумні пояснення, пов'язані з особливостями проекту:

- NoF = 0.3 - однорідність класів по кількості полів;
- NoC = 0.25 - однорідність за кількістю наслідувачів, однотипність варіантів наслідування;
- CBO = 0.4 - однорідність по зв'язках між класами;
- Ca = 0.3 - однорідність по використанню різних пакетів проекту сторонніми
- класами.

Можливо, в проекті має місце повторюваність ідентичних схем або груп класів, які знижують розкид статистичних даних.

2.4.2 Приклад аналізу. Бібліотека середовища виконання програм в Android

Бібліотека Android представляє собою бібліотеку оточення (середовища виконання), в якої працює Java-програма в ОС Android. Фактично вона реалізує всю специфіку середовища функціонування програм в Android, нижче її розташовані бібліотеки Linux і сам Linux. Бібліотека має розмірність коду, яку можна порівняти із середньостатистичною, але значна кількість класів (730 проти 235 в середньому). Вид нормованих метрик (рис. 2.15) показує, що вони сильно відрізняються від статистичних.

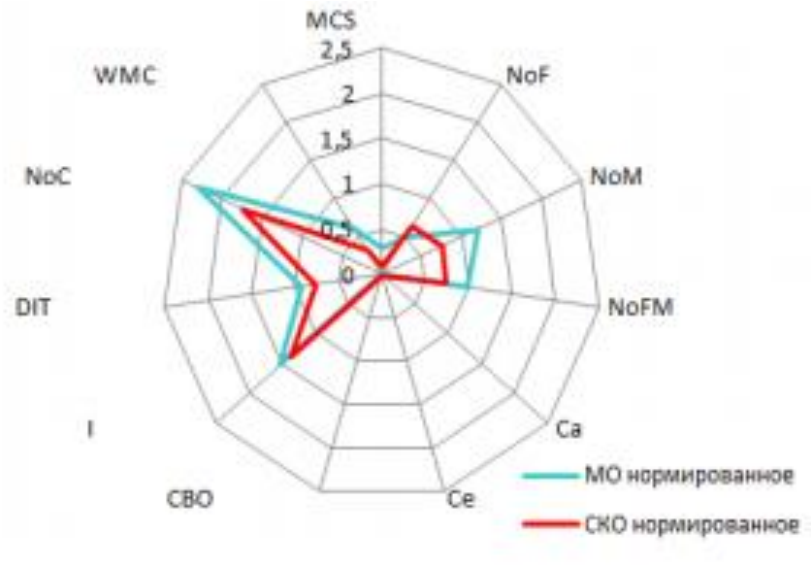


Рисунок 2.15 – Нормовані метрики бібліотеки Android

Бібліотека має дуже низькі нормовані значення зчеплення (Ca = 0.006, Ce = 0.004, CBO = 0.005), що пояснюється високою автономністю компонент і відсутністю прямих зв'язків між класами з різних пакетів. Кожен пакет (категорія) являє собою окремий сервіс і самою бібліотекою не використовується.

Значна кількість наслідувчів (NoC = 1,75) вказує на «популярність» використання базових класів і абстракцій, а малі значення нормованої довжини

методу (MCS = 0.08) і зваженої складності класу (WMC = 0.3) - про виняткове використання коротких методів з простою логікою.

2.4.3 Приклад аналізу. стандартному середовищі реактивного програмування RxJava

Середа реактивного програмування пропонує засоби для організації подієвого управління на основі передплати на події та широко використовується як засіб організації програми на Java, в тому числі взаємодії зовнішніх форм (графічного інтерфейсу). Бібліотека середнього розміру - близько 1 Мб. Вид нормованих метрик (Рис. 2.16) показує, що деякі з них значно відрізняються від статистичних.

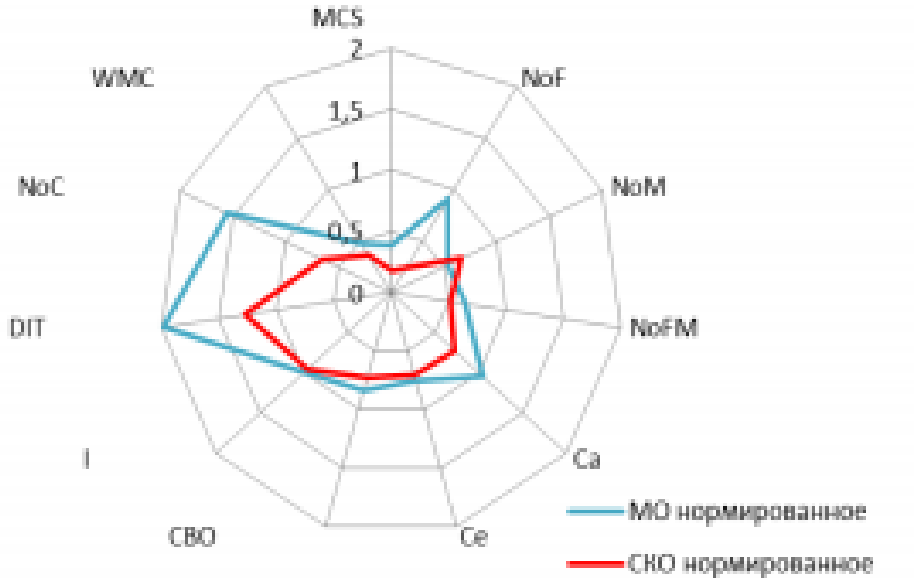


Рисунок 2.16 – Нормовані метрики бібліотеки RxJava

Значна глибина дерева наслідування (DIT = 2), кількість наслідувачів (NoC = 1.55), мінімальна довжина методів (MCS = 0.4) і низька сумарна складність (WMC = 0.47) характеризують розробку як виключно модульну, з високим рівнем наслідування. Це узгоджується з тим, бібліотека підтримує специфічну парадигму програмування, для якої необхідно створювати значну кількість абстракцій і розвивати їх.

3 ВЕБ-ДОДАТКИ І ВЕБ-СЕРВІСИ НА ПЛАТФОРМІ JAVA

3.1 Віртуальна машина Java

Мова Java є скомпільованою і інтерпретованою мовою. Вихідний код Java перетворений в прості бінарні інструкції, що більше схоже на машинний код мікропроцесора. Як би там не було, якщо джерело C або C++ зменшений до рідних інструкцій для певної моделі або процесора, то код Java скомпільовано в універсальний формат - інструкції для віртуальної машини.

Скомпільований байт-код Java виконується інтерпретатором часу виконання. Система підтримки виконання виконує всі звичайні дії процесора апаратного забезпечення, але робить це в безпечному віртуальному середовищі. Вона виконує набір стекових інструкцій і управляє пам'яттю як операційна система. Вона створює прості типи даних і управляє ними, завантажує і викликає знову посилаючі блоки коду. Що найголовніше, вона робить це відповідно до чітко визначеної відкритої специфікації, яку може застосувати будь-який, хто хоче створити віртуальну машину Java. Разом віртуальна машина і визначення мови представляють повну специфікацію. Немає властивостей базового мови Java, які б залишилися невизначеними або залежними від впровадження. Наприклад, мова Java специфікує розміри і математичні властивості всіх своїх простих типів даних, а не залишає їх реалізації платформи.

Інтерпретатор Java досить легкий і компактний, він може бути реалізований в будь-якій формі, придатної для певної платформи. Інтерпретатор може працювати як окремий додаток або може бути впроваджений в інше програмне забезпечення, таке як веб-браузер. Все сказане означає, що код Java повністю портативний. Один і той же байт-код програми Java може працювати на будь-якій платформі, що забезпечує середу виконання Java, як показано на рис. 3.1. Вам не потрібно робити альтернативні версії вашого додатку для різних платформ і не потрібно роздавати вихідний код кінцевим користувачам.

Фундаментальна одиниця коду Java - клас. Як і в інших об'єктно-орієнтованих мовах, класи є компонентами додатка, які містять виконуваний код і дані. Скомпільовані класи Java поширюються в універсальному бінарному форматі, що містить байт-код Java і іншу інформацію класу. Класи можуть зберігатися окремо і зберігатися в файлах або архівах локально або на мережевому сервері. Класи розташовуються і завантажуються динамічно під час виконання, коли вони необхідні додатку.

На додаток до платформо-залежним системам виконання Java має ряд фундаментальних класів, які містять архітектурнозалежні методи. Ці «рідні» методи служать воротами між віртуальною машиною Java і реальним світом. Вони реалізовані в скомпільованій мові на платформі хоста і надають доступ нижнього рівня до ресурсів, таким як мережа, система управління вікнами і файлова система хоста. Переважна частина Java написана на самій мові Java, завантаженій з цих базових елементів, і тому портативна. Це базові інструменти Java, такі як компілятор Java, мережеві бібліотеки і бібліотеки GUI, які також написані на Java і доступні на всіх платформах Java точно таким же чином без портування.

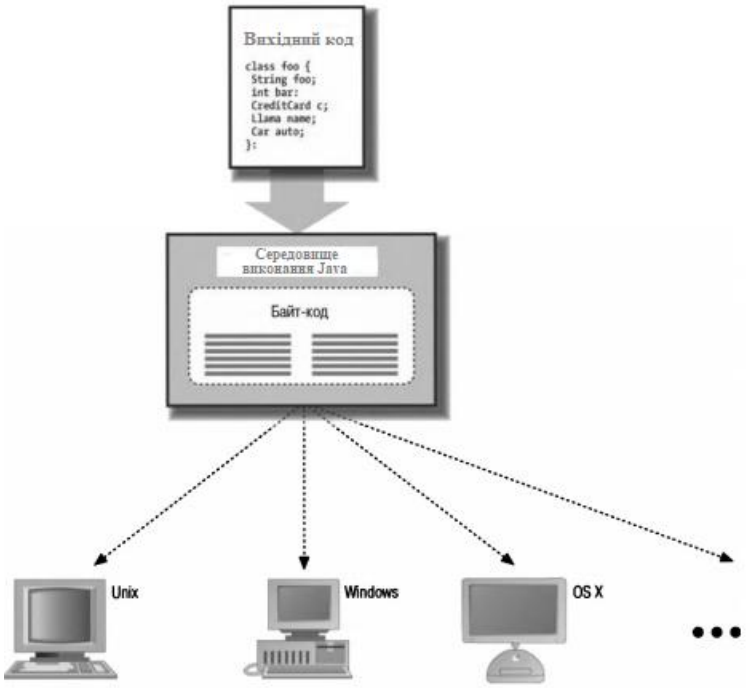


Рисунок 3.1 - Середовище виконання Java

Історично інтерпретатори вважалися повільними, але Java не є традиційною інтерпретованою мовою. На додаток до компіляції вихідного коду в портативний байт-код Java була ретельно розроблена таким чином, що програмна реалізація системи виконання може далі оптимізувати свою продуктивність, компілюючи байт-код в оригінальний машинний код на льоту. Це називається компіляція «точно в строк», або динамічна компіляція. З динамічної компіляцією код Java може виконуватися так само швидко, як оригінальний код, і підтримувати свою транспортабельність і безпеку.

Цей пункт часто незрозумілий тим, хто хоче порівняти продуктивність мов. Існує тільки одна серйозна проблема, від якої страждає компільований Java-код під час виконання заради безпеки і архітектури віртуальної машини, - перевірка меж масиву. Все інше може бути оптимізовано в оригінальний код так само, як при статичній компіляції мови. Крім цього, мова Java містить більше структурної інформації, ніж багато інших мов, передбачаючи більше типів оптимізації. Також ці оптимізації можуть здійснюватися під час виконання, приймаючи до уваги реальну поведінку і характеристики додатків.

Проблема традиційної динамічної компіляції - це те, що оптимізація коду займає час. Отже, динамічний компілятор може давати гідні результати, але може страждати від значного часу очікування під час запуску програми. В основному це не має великого значення для довго працюють додатків серверної сторони, але є серйозною проблемою для програмного забезпечення клієнтської сторони і додатків, які працюють на менших пристроях з обмеженими можливостями. Щоб вирішити це питання, технологія компілювання Java, звана HotSpot, використовує прийом, іменований адаптивної компіляцією. Ділянка коду, який повторно виконується, може бути тільки малим фрагментом всієї програми, але його поведінка визначає загальну продуктивність програми. Адаптивна компіляція також дозволяє часу виконання Java використовувати переваги нових типів оптимізації, яка не може бути виконана в статично компільованих мовах, звідси твердження, що Java-код працює швидше, ніж C / C ++ в деяких випадках.

Щоб отримати перевагу від цього факту, технологія HotSpot починає як звичайний інтерпретатор байт-коду Java, але з однією різницею: вона вимірює (профілює) код під час його виконання, щоб побачити, які частини коду виконуються багаторазово. Коли вона знає, які частини коду є критичними для продуктивності, технологія HotSpot компілює ці частини в оптимальний власний машинний код. Оскільки вона компілює тільки маленьку частину програми в машинний код, вона має можливість використовувати час, необхідний для оптимізації цих частин. Інша частина програми може взагалі не потребувати компіляції - тільки інтерпретації - що зберігає пам'ять і час. Насправді віртуальна машина Java може працювати в двох режимах: клієнтському і серверному, що визначає, робити акцент на швидкому часу старту і збереженні пам'яті або на швидкості виконання.

У цьому місці виникає природне запитання: навіщо викидати всю цю гарну профільну інформацію кожен раз, коли вимикається додаток? Що ж, компанія Sun торкнулася цієї теми під час релізу Java 5.0 через використання загальних класів і класів тільки для читання, які перманентно зберігаються в оптимізованій формі. Це значно зменшило і час запуску, і накладення багатьох Java-додатків на одній машині. Ця технологія досить складна, але ідея проста: оптимізувати частини програми, які повинні працювати швидко, і не турбуватися про все інше.

3.2 Інструменти і середовище Java

Хоча можна писати, компілювати і запускати додатки Java тільки за допомогою набору для розробки Java-додатків (JDK) і простого текстового редактора (наприклад, vi, Блокнот (Notepad) і ін.), сьогодні величезна частина Java-коду пишеться завдяки інтегрованому середовищу розробки (IDE). Переваги використання IDE включають цілісний погляд на вихідний Java-код з підсвічуванням синтаксису, навігаційної допомогою, контролем за вихідним кодом, інтегрованою документацією, розстановкою, реструктуризацією і розміщенням - все у вас під рукою.

Компанія IBM спочатку була ініціатором The Eclipse Project в 2001 році, що спонукало консорціум постачальників програмного забезпечення створити розширене середовище для розробки, щоб конкурувати з популярним в ту пору середовищем Visual Studio від Microsoft. Сьогодні середовище Eclipse перетворилося на потужну платформу з відкритим кодом, підтримувану як окремими людьми, так і корпораціями, фінансовану процвітаючою екосистемою плагінів і фреймворків. Хоча Java є найпопулярнішою мовою, що асоціюється з Eclipse, IDE підтримує дюжину мов.

Eclipse вимагає установки середовища виконання Java (JRE, Java Runtime Environment).

3. 3 Конфігурація Eclipse і створення проекту

При першому використанні Eclipse необхідно вибрати робочий простір. Це коренева папка для зберігання нового проекту в Eclipse. Місцезнаходження за умовчанням знаходиться всередині самої папки у програмі. Необхідно вибрати місце розташування та натиснути кнопку ОК.

Eclipse зустрічає екраном вітання. Закрити це вікно, закривши вкладку вітання в додатку.

Eclipse зберігає всю інформацію про зміни в папці configuration в папці установки Eclipse. Можна залишити додаток і видалити цю папку.

Якщо необхідно скинути всі вікна додатка до їх установок за замовчуванням, можна вибрати команду меню Window => Reset Perspective (Вікно => Скидання Перспектива).

Необхідно вибрати команду меню File => New => Java Project (Файл => Новий => Проект Java) з меню програми і ввести текст LearningJava в поле Project Name (Назва проекту) у верхній частині діалогового вікна, як показано на рис. 2.1.

Треба переконатися, що в групі елементів управління JRE обрана версія середовища JavaSE-1.7, як показано на рисунку, і натиснути кнопку Next (Далі) у нижній частині діалогового вікна. Потім потрібно буде встановити шлях збірки в

системну бібліотеку Java 7. Обрати вкладку Libraries (Бібліотеки) і видалити бібліотеку Java 1.6. Натиснути кнопку Add Library (Додати бібліотеку) і вибрати бібліотеку JavaSE-1.7. Зараз серед Eclipse налаштована на використання Java 7. Натиснути кнопку Finish (Готово).

Рисунок 3.2- Діалогове вікно для створення нового проекту Java

3.4 Багатосторінкові і односторінкові додатки

Інструменти для створення серверних веб-додатків постійно еволюціонують. Колись в якості стандарту виступала технологія CGI, яка дозволяла обслуговувати запити, надіслані з веб-браузерів, за допомогою скриптових мов програмування, таких як Perl. Деякі веб-сервери пропонували API-інтерфейси, написані на компільованих мовах; наприклад, модулі для веб-сервера Apache написані на C та C++. Однак технологія Java Servlet API, володіючи переносимістю, безпекою і високою продуктивністю, швидко стала найпопулярнішою архітектурою для побудови веб-додатків. У наші дні існує безліч інструментів для створення веб-сервісів; системи, написані на мові Java, конкурують з аналогічними рішеннями

на базі Microsoft .NET і альтернативними платформами, такими як Ruby on Rails. Однак сучасні тенденції такі, що головний пріоритет віддається клієнтським технологіям на зразок JavaScript і HTML5, тому не так уже й важливо, якою мовою реалізовані серверні компоненти або веб-сервіси.

Більшу частину того часу, що існує платформа Java, веб-додатки працювали по одному і тому ж простому принципу: браузер робить запит за певною URL-адресою, а сервер генерує відповідь у вигляді HTML-документа. При цьому будь-які дії користувача приводили до переходу на наступну сторінку. У цьому процесі вся робота (або більша її частина) виконувалася серверної стороною, що, на перший погляд, виглядає цілком логічно, враховуючи те, де саме знаходяться дані і сам веб-сервіс. Однак у такої архітектури є свої природні обмеження, виражені в нестачі чуйності і відсутності цілісності. Складно домогтися від веб-додатку такої ж плавної роботи, як в настільних програмах, якщо користувач змушений «перестрибувати» від однієї сторінки до іншої, кожен раз чекаючи завантаження в браузері. До того ж виникає проблема збереження даних між цими переходами. Зрештою, веб-браузери створювалися для перегляду документів, а не для виконання повноцінних програм.

Але за останні роки в світі веб-розробки відбулися великі зміни. Стандарти HTML і JavaScript вийшли на той рівень, коли більшу частину призначеного для користувача інтерфейсу і бізнес-логіки можна зберігати на клієнтській стороні, запитуючи у сервера дані і різні функції в фоновому режимі. Відповідно до цієї концепції сервер повинен повернути лише одну «сторінку» формату HTML, в якій містяться посилання на скрипти, таблиці стилів і інші ресурси, необхідні для генерування інтерфейсу додатку. далі естафету приймають скрипти, написані на мові JavaScript: вони можуть динамічно створювати нові елементи сторінки і управляти вже існуючими, формуючи призначений для користувача інтерфейс за допомогою багатих можливостей технології HTML DOM. Для отримання даних і виклику віддалених функцій застосовуються асинхронні (фонові) запити. Часто результат повертається в форматі XML, завдяки чому цей вид взаємодії отримав назву AJAX (Asynchronous Javascript and XML - асинхронний JavaScript і XML).

Завдяки цьому новому підходу багато аспектів веб-розробки значно спростилися. Більше немає ніякої необхідності працювати в «посторінковому» режимі, обмінюючись з сервером запитами і відповідями. Тепер клієнт більше схожий на настільний додаток - він може миттєво реагувати на дії користувача і управляти віддаленими даними і функціями, не перериваючи своєї роботи.

3.5 JSP-сценарії

JSP - це ще одна технологія для створення серверних додатків, орієнтованих, в основному, на висновок окремих сторінок (документів). Такі додатки складаються з HTML-коду, всередині якого підтримуються нестандартні теги і синтаксис мови Java. JSP-сценарії динамічно компілюються веб-сервером, в результаті чого виходять сервлети, які генерують вміст сторінок і здатні працювати з Java-інтерфейсами як прямо, так і опосередковано. І хоча весь код виконується на стороні сервера, все виглядає так, ніби бізнес-логіка знаходиться прямо всередині сторінки. У такого підходу є свої переваги і недоліки. Перевага полягає в тому, що стиль програмування, при якому розробник має справу безпосередньо з HTML-документом, досить простий для розуміння і може бути легко поставлений на потік. Але звідси випливає й недолік: сторінки перетворюються в заплутану суміш з бізнес-логіки і коду для відображення призначеного для користувача інтерфейсу. Чим сильніше код змішується зі статичною розміткою, тим складніше стає його підтримувати.

Більшість великомасштабних проектів, заснованих на технології JSP, використовують бібліотеки тегів, щоб зменшити кількість зайвого коду. При написанні JSP-сценаріїв всі складні операції і бізнес-логіку часто виносять в окремий контролер (який теж є сервлетом). У цьому випадку застосовується архітектура «модель-уявлення-контролер» (Model-View-Controller або MVC), коли кожна задача вирішується за допомогою окремих компонентів. Такий поділ нівелює недоліки технології JSP, дозволяючи повною мірою скористатися її перевагами.

XML - це набір стандартів для роботи зі структурованою інформацією, представленою у вигляді тексту. XSL (Extensible Stylesheet Language) - це мова для перетворення XML-документів в інші формати, в тому числі і в HTML. Поєднання сервлетів, які можуть генерувати XML код, і XSL-таблиць, здатних готувати вміст для відображення, володіє великими можливостями. Веб-сервіси використовують XML в якості основного формату даних, завдяки чому вони абсолютно не прив'язані до конкретних платформ і мов. І звичайно ж, варто сказати, що XML є базовим форматом для обміну інформацією з AJAX-додатками, написаними на мові JavaScript.

3.6 Фреймворки для створення веб-додатків

Якщо розглядати веб-додатки в контексті класичної архітектури MVC, то «уявлення» в традиційній «багатосторінкової» моделі генерується в рамках веб-браузера, а модель (дані) і контролер (бізнес-логіка) знаходяться на стороні сервера. Ми вже говорили, чому веб-додатки все частіше використовують «односторінкову» концепцію, де в браузер виноситься більше компонентів; Проте, за ці роки було створено безліч фреймворків, які до сих пір підтримують класичний підхід. Найчастіше такі фреймворки працюють на більш високому рівні в порівнянні з сервлетами; вони надають зручні засоби для написання і підключення контролерів, дозволяючи конфігурувати сторінки для виведення результатів.

Свого часу одним з найпопулярніших інструментів для побудови багатосторінкових веб-додатків був фреймворк Apache Struts Web Application Framework. Підтримуючи парадигму MVC, він має модульну структуру контролерів і розширену бібліотеку тегів для уявлень, в якості яких виступають JSP-сценарії.

Фреймворк Struts поєднує в собі деякі аспекти навігації і розподілу викликів, завдяки яким компоненти об'єднуються в єдиний веб-додаток. Він використовує конфігураційні файли в форматі XML і дозволяє застосовувати декларативний

стиль для зв'язування HTML-форм з об'єктами мови Java, а також виконувати автоматичну перевірку полів введення.

Відповіддю компанії Sun на фреймворк Struts став проект JSF. Він був розроблений в рамках ініціативи Java Community Process (за участю деяких творців Struts) і повинен був стати «офіційно схваленим» фреймворком для веб-додатків на мові Java. Він отримав вдосконалену модель MVC з серверними програмними компонентами, а також більш тонке управління навігацією і подіями. Володіючи досить суперечливою репутацією, він так ніколи і не перевершив в популярності свого конкурента, Struts.

Ще одна популярна система для створення веб-додатків з архітектурою MVC, Spring Web Flow, заснована на фреймворку Spring. І таких прикладів дуже і дуже багато.

3.7 Google Web Toolkit, HTML5, AJAX

Google Web Toolkit (GWT) - це відкритий і безкоштовний фреймворк від компанії Google. З його допомогою розробники можуть компілювати свій Java-код в скрипти на мові JavaScript, які виконуються в веб-браузері і взаємодіють з сервером за допомогою спеціального механізму, заснованого на технології RPC (зовні це дуже схоже на інтерфейси RMI). Середовище виконання фреймворка GWT надає свій власний набір класів для створення призначеного для користувача інтерфейсу, а також багатий набір стандартних бібліотек. Ця потужна платформа дозволяє розробляти великі і складні додатки; завдяки їй ви можете використовувати велику частину переваг мови Java на стороні веб-браузера. Однак в порівнянні з іншими фреймворками GWT має досить високий поріг входження.

Платформа Java знаходиться на серверній стороні веб-додатка. Для створення клієнтської частини, яка буде працювати в веб-браузері, ми змушені скористатися послугами далекого родича мови Java - JavaScript. Спроби стандартизувати просунуті можливості технологій HTML і JavaScript, вжиті в останні роки,

зробили справжню революцію в розробці клієнтських веб-додатків, значно поліпшивши їх продуктивність. За великим рахунком, початок цієї революції поклала технологія AJAX, яка зробила поведінку клієнтської частини більш динамічною. Вибухове зростання популярності мобільних пристроїв, що почався відносно недавно, прискорило впровадження стандарту HTML5; завдяки цьому веб-браузери обзавелися більш повною об'єктною моделлю, підтримкою аудіо- і відеоформатів, стандартними засобами для малювання, інструментами для роботи з векторною графікою і автономним сховищем даних. У міру просування процесу стандартизації ми отримуємо все нові і більш захоплюючі можливості. Наприклад, веб-сокети (WebSockets), які дозволяють обмінюватися даними між веб-браузером і сервером з мінімальною затримкою, повинні породити масу абсолютно нових додатків.

3.8 Веб-додатки на платформі Java

Поки що ми вживали термін «веб-додаток» в загальному сенсі, маючи на увазі будь-яку серверну програму, доступ до якої можна отримати з веб-браузера. В контексті технології Java Servlet API веб-додаток являє собою набір сервлетів і веб-сервісів, які підтримують класи, написані на мові Java, і можуть працювати з такими ресурсами, як HTML- і JSP-сторінки, зображення і конфігураційна інформація. Щоб розгорнути (встановити і запустити) веб-додаток на сервері, його упаковують в WAR-файл. Це звичайний JARархів, який крім програмних файлів містить деякі дані, необхідні для установки. WAR-файли не тільки забезпечують переносимість додатків, але і надають стандартний спосіб для їх розгортання на сервері.

Більшість архівів формату WAR мають власний конфігураційний файл під назвою web.xml. Він містить дані про сервлети, які потрібно розгорнути, їх імена, шляхи, параметри для ініціалізації і масу іншої інформації, включаючи вимоги до безпеки і аутентифікації. Однак в останні роки багато додатків стали ігнорувати цей файл і використовувати замість нього анотації. У більшості випадків для

розгортання сервлетів та веб-сервісів досить помітити свої класи, надавши всю необхідну інформацію, або просто упакувати їх в WAR-архів (можна зробити і те й інше).

Веб-додатки виконуються в певному середовищі. У кожного з них є свій «корневий» шлях на веб-сервері; це означає, що все URL-адреси, що вказують на сервлети і файли, мають загальний унікальний префікс (наприклад, http://www.example.com/someapplication/). Веб-додатки ізольовані один від одного і мають доступ тільки до своїх файлів (хоча, звичайно ж, вони можуть обійти це обмеження за допомогою веб-сервера). Кожна програма має власний контекст - загальне місце, де сервлети можуть обмінюватися інформацією і отримувати ресурси з середовища виконання. Завдяки високому ступеню ізоляції сучасні промислові системи можуть динамічно розвиватися і оновлюватися, не маючи ніяких проблем з надійністю і безпекою. Дійсно, веб-додатки повинні бути автономними і самодостатніми, не пов'язаними між собою. І хоча нічого не заважає налагодити тісну міжпрограмну взаємодію на високому рівні, для цих цілей краще використовувати веб-сервіси.

4 ВИБІР МЕТРИК ДЛЯ WEB-ДОДАТКІВ В ПРОГРАМНОМУ СЕРЕДОВИЩІ JAVA

Вибір метрик - це важливий, але часто ігнорований аспект кожного проекту, що розвивається. Однією з проблем, яка може з'явитись майже у кожному проекті, є необхідність збирати метрики про різні аспекти. продуктивності додатків. Щоб спростити додавання можливостей відстеження метрик для своїх додатків, необхідно створити набір класів Java, який можна буде додати до будь-якого проекту та посилання у працюючому коді для накопичення даних.

Першим кроком у виборі метрик є визначення того, які вимірювання знадобляться. З розвитком додатків, що розробляються, розробляється і бібліотека загальних метрик для різних типів програм.

Набір класів, який використовується для збору метрик, застосовує файл властивостей для визначення доступних метрик і різні типи, які застосовуються до цих метриках. В системі збору метрики можна поділити на п'ять типів:

- метрика часу;
- метрика минулого часу;
- індикатор;
- статична метрика;
- середня метрика.

Ці метрики охоплюють більшість предметів, які необхідні для типової програми.

Метрика часу - це просто значення дати та часу. Як правило, вона використовується для зберігання точного часу події.

Метрика минулого часу схожа на метрику часу, за винятком того, що вона повідомляє про час, який минув з моменту встановлення метрики. Можна розрахувати час, що минув, за значенням позначки часу, але телеметричні класи повідомляють число, не вимагаючи подальших обчислень.

Найпоширенішим типом метрики є Індикатор, оскільки в основному це рахунок. Кожного разу, коли встановлюється показник метрики, значення збільшується до оновленого підрахунку.

Статична метрика - це числове значення, яке зберігає лише останнє або попереднє налаштування.

Середні метрики відстежують вимірювання та підтримують поточне середнє.

Метрики зберігаються у довгих значеннях, тому підсумки обмежуються 2^{63-1}.

Описи метрик зчитуються з файлу властивостей, показаного в лістингу A, класом менеджера метрик. На додаток до окремих метрик у файлі властивостей встановлюються інтервал журналу та хост.

Кожна метрика визначається шляхом присвоєння номера, імені, опису та значення скидання. Ім'я використовується для реєстрації метрики, а властивість reset визначає, чи ця метрика скидається кожного разу, коли вона повідомляється. Скидання корисне для надсилання метрик до іншої програми збору, яка очікує лише змінені значення. Скинуте значення true очистить значення кожної метрики після її реєстрації.

Метрики встановлюються та контролюються класом управління під назвою MetricManager. MetricManager - це синглтон, який читає доступні метрики та встановлює початкові значення для цих метрик. Клас MetricManager показаний у лістингу B.

Шаблон Singleton гарантує наявність лише одного екземпляра класу та забезпечує єдину точку доступу до цього екземпляра.

MetricManager реалізує інтерфейс під назвою TimerCallBack, який ініціює реєстрацію кожного разу, коли досягається інтервал журналу у файлі властивостей. Ця реалізація використовує власний клас TaskTimer, показаний у лістингу C. Одним вдосконаленням цього набору класів було б використання стандартних засобів таймера, доступних у Java.

Клас MetricManager включає різні методи додавання метрик з інших класів. Метод додавання замінено, щоб дозволити додавання цілих чисел, довжини та календарів. Метод вимкнення гарантує, що клас MetricManager має можливість

реєструвати існуючі значення до того, як клас буде знищений. Інший клас, MetricId, зберігає статичні константи для зручності (MetricId також використовується в лістингу B). Ці статичні константи використовуються різними класами для посилання на відповідну метрику під час виклику методу додавання.

Клас MetricManager накопичує метрики з файлу властивостей і використовує цю інформацію для зберігання кожної визначеної метрики в класі метрики. Клас Metric, показаний у лістингу D, визначає функціональність, необхідну для зберігання кожної метрики. Більша частина реалізації класу Metric призначена для накопичення та виведення метричних значень. Цей клас можна змінити на інтерфейс, а інша реалізація інтерфейсу може підтримувати тип даних для кожної окремої метрики.

Щоб використовувати класи відстеження метрик, окремий клас повинен імпортувати відповідний пакет та отримати екземпляр MetricManager. Лістинг E включає зразок класу, який реалізує метрики та додає до одного зі значень.

Цей зразок класу оголошує статичну змінну класу, щоб містити посилання на MetricManager. Далі код використовує MetricManager для додавання значень до метрик запуску та часу роботи програми, посилаючись на константи класу MetricId.

ВИСНОВКИ

Величезні перспективи і самі захоплюючі можливості для розробників програмного забезпечення лежать в приборканні сили мереж.

Додатки, створені сьогодні, для яких би цілей і аудиторії вони не планувалися, майже неодмінно запускаються на машинах, пов'язаних глобальною мережею комп'ютерних ресурсів. Зростаюча важливість мереж висуває нові вимоги до існуючих інструментів і змушує швидко рости список абсолютно нових типів додатків.

Тому, необхідно мати програмне забезпечення, яке працює стабільно на будь-якій платформі і яке добре поєднується з іншими додатками; додатки, які використовують переваги Всесвітньої павутини, здатні отримати доступ до несумірних і розподілених джерел інформації.

Доречно мати розумні додатки, які можуть бродити по мережах замість нас, розшукуючи інформацію і служити електронними агентами.

Досить давно відомо, яке програмне забезпечення необхідне, але в дійсності отримуємо його тільки протягом останніх декількох років.

Принцип обчислення статичних метрик коду оснований на синтаксичному аналізі вихідного коду. При цьому процес збору метрик повинен бути автоматизованим. Завдяки цьому можна виміряти метрики всієї системи незалежно від її розміру.

Більш того, можна прогнозувати всю систему на основі метрик - розробники можуть легко знаходити несправні модулі, оскільки вони мають чітке уявлення про вразливості системи

Статичні метричні коду простіші та широко використовуються на практиці; тому, вони представляють безпечний вибір для прогнозування несправного програмного забезпечення.

Програмне забезпечення забезпечить інструменти для обчислення та візуалізації найбільш необхідних метрик програмного забезпечення. Ці

інструменти мають на меті уніфікувати всі показники програмного забезпечення, щоб отримати до них доступ із простого у використанні додатка.

ПЕРЕЛІК ПОСИЛАНЬ

1. Юрій Грицюк. Аналіз вимог до програмного забезпечення: Видавництво «Львівська політехніка», - 2018. – 456 с.

2. Ірина Бородкіна, Георгій Бородкин. Інженерія програмного забезпечення. Посібник для студентів вищих навчальних закладів: Видавництво «Центр навчальної літератури», - 2018. -204 с.

3. Р. Сеттер, А. Яшин. Java на примерах. Практика, практика и только практика: Видавництво «Наука и техника». - 2016. -256 с.

4. Олег Герман, Юлія Герман. Программирование на Java и C# для студента: Видавництво «БХВ-Петербург». - 2012. – 512 с.

5. Корі Сандлер, Гленфорд Майерс, Том Баджетт. Искусство тестирования программ: Видавництво «Диалектика, Вильямс». - 2012. – 272 с.

6. О. С. Корженевський, М. В. Грайворонський. Система аналізу програмного забезпечення для архітектур, відмінних від x86-x64: Всеукраїнська науково-практична конференція студентів, аспірантів та молодих вчених. - 2015. – 4 с.

7. Bellard Fabrice. QEMU, a Fast and Portable Dynamic Translator // FREENIX Track. – 2005. - 41–46 с.

8. Sikorski Michael, Honig Andrew. Practical malware analysis. — San Francisco: William Pollock, 2012

9. К.М. Лавріщева. Програмна інженерія: Всеукраїнська науково-практична конференція студентів, аспірантів та молодих вчених: Видавництво: «Націанальна академія наук України» - 2008. – 322 с.

10. О.Б. Вовк. Аналіз та оцінювання якості програмного продукту (поняття, терміни, означення) Видавництво: Національний університет “Львівська політехніка”, кафедра “Інформаційні системи та мережі” – 2003. – 73 с.

11. Ю. І. Грицюк, О. Т. Андрущакевич. Засіб для визначення якості програмного забезпечення методами метричного аналізу. Видавництво:

Національний університет "Львівська політехніка", м. Львів, Україна - 2018, – 171 с.

12. Граді Буч, Роберт А. Максимчук, Майкл У. Энгл. Об'єктно-орієнтований аналіз та проектування з прикладами додатків. Видавництво «Вильямс», - 2010, - 720 с.

13. Л. Мухаметова. Методы выборочных обследований. Видавництво: «БИБКОМ», - 2009, - 166 с.

14. А. Т. Яровий, Є. М. Страхов. Багатовимірний статичний аналіз. Навчальний посібник: Одеський національний університет імені і. І. Мечникова Інститут математики, економіки і механіки. Видавництво «Астропринт», - 2015, - 130 с.

15. В.Є. Бахрушин. Навчальний посібник: Методи аналізу даних. Видавництво: «Класичний приватний університет», - 2011, - 269 с.

16. Старух А.І. Класичний приватний університет. Навчальний посібник: Львівський національний університет імені івана франка, -2020, - 43 с.

17. Г.С. Теслер. Метрики и нормы в иерархии категориальных семантик и функций. Навчальний посібник: Математичні машини і системи, - 2005, - 13 с.

18. Салюк М.А. Статистична обробка даних. Експериментального дослідження. Навчальний посібник: «Експериментальна психологія», - 2010, - 28 с.

Додаток А

```
# MetricManager.properties
# Startup parameters for the MetricManager.
#
# host name used for recording metrics.
# Leave blank to use machine's host name
#host=myname

# Log interval.  Number of milliseconds between log writes.
# Minimum is 60000 (1 minute)
logInterval=900000
#logInterval=60000

# Comma separated list of metrics that are available.
metrics=m1,m2,m3,m4,m5,m6,m7
#
# Individual metric entries follow:
#
# Syntax:
# <metricid>.type=<Type of metric>
# <metricid>.name=<String name for the metric to be used in log output>
# <metricid>.reset=<true | false>
# <metricid>.description=<Description to be used in queries etc.>
#
# Note: Metric ids cannot be re-assigned without significant code
#       changes since
#       the id is used by classes for recording their metrics.
#       This is a todo item to make these more easily changed
#       (if necessary).
#
# Example:
# m1.type=3
# m1.name=Attempted Connections
# m1.reset=true
# m1.desc=Total number of connection attempts made since start.
#
# Available types:
#   TIME_STAMP_METRIC   = 1
#       A date time.  This metric is set once and simply stored for retrieval.
#   ELAPSED_TIME_METRIC = 2
#       A date time that returns the time elapsed in milliseconds.
#   INDICATOR_METRIC    = 3
#       A normal numerical metric that is constantly increased by new values.
#   STATIC_METRIC       = 4
#       A numerical metric that is replaced with incoming values.
#   AVERAGE_METRIC     = 5
#       Returns the average (float) of a number of measurements taken over time.

############################################################
m1.type=1
m1.name=Application Started
m1.reset=false
m1.desc=Date and time the application started in UTC.

############################################################
m2.type=2
m2.name= Running Time
m2.reset=false
m2.desc=Total amount of time(milliseconds) the application has been running.
```

```
m3.type=3
m3.name= Connections Attempted
m3.reset=true
m3.desc=Total number of connection attempts.

############################################################
m4.type=3
m4.name= Connections Completed
m4.reset=true
m4.desc=Total number of completed connections.

############################################################
m5.type=3
m5.name=Bytes Received
m5.reset=true
m5.desc=Total number of bytes received.

############################################################
m6.type=4
m6.name=Request Processors
m6.reset=false
m6.desc=Number of request processor threads running to service incoming connections.

############################################################
m7.type=5
m7.name=Average Connect Time
m7.reset=true
m7.desc=Number of milliseconds a connection is active with the application (does not include queue time).
```

Додаток Б

```
import java.io.*;
import java.util.*;
import java.sql.*;
import java.net.InetAddress;
import java.net.UnknownHostException;
//import org.apache.log4j.Category;

/**
 * This class is a singleton that provides tracking and logging of
 * metrics related to the listener.  The metrics are tracked by this
 * class and logged at intervals defined in the properties file.  The
 * available metrics are defined in the properties file.
 *
 * @see com.telemics.listener.Listener
 */
public class MetricManager implements TimerCallBack {
  private static MetricManager mInstance; // The single instance
//   static Category log = Category.getInstance(MetricManager.class.getName());
  private Hashtable mMetrics = new Hashtable();
  private long mTimerInterval = 60000;
  private String mHost;
  private TaskTimer mTimer;

  /**
   * Returns the single instance, creating one if it's the
   * first time this method is called.
   *
   * @return      The single instance of MetricManager.
   */
  static synchronized public MetricManager getInstance() {
    if ( mInstance == null )
                    mInstance = new MetricManager();
    }
    return mInstance;
  }

  /**
  * A private constructor since this is a Singleton.  The constructor reads
  * the properties file and sets up the defaults.  The Metrics to be collected
  * are setup from data in the properties file as well.
  */
  private MetricManager() {
    Properties pProps = new Properties();
    String pFilePrefix = MetricManager.class.getName().replace('.', '/');

    try {
      FileInputStream is = new FileInputStream(pFilePrefix + ".properties");
      pProps.load(is);
      is.close();
    }
    catch (Exception e) {
//        log.error("Unable to read " + pFilePrefix + ".properties, using defaults.");
      return;
    }
    try {
      mHost = pProps.getProperty("host", InetAddress.getLocalHost().getHostName());
    }
    catch(UnknownHostException ue) {
```

```
//        log.error("Unable to retrieve host name of local machine, using 'unknown'", ue);
      }
      try {
        mTimerInterval = Long.parseLong(pProps.getProperty("logInterval", "60000"));
      }
      catch(Exception e) {
//        log.error("Unable to retrieve logInterval property, using default.");
      }
      loadMetrics(pProps);

      mTimer = new TaskTimer(this, "Metric Timer", mTimerInterval);

      mTimer.start();

//      log.info("MetricManager started: Log Interval is " + mTimerInterval / 1000 + ".");
    }

    /**
     * Loads and registers all the metrics.
     *
     * @param     pProps      The properties for MetricManager.
     */
    private void loadMetrics(Properties pProps) {
      String metricId;
      int metricType = 0;
      String metricName;
      boolean metricReset = true;
      String metricDescription;

      String metricsProperty = pProps.getProperty("metrics");
      StringTokenizer metricIds = new StringTokenizer(metricsProperty, ",");

      while (metricIds.hasMoreElements()) {
        metricId = metricIds.nextToken().trim();
        try {
          metricType = Integer.parseInt(pProps.getProperty(metricId + ".type",
String.valueOf(Metric.INDICATOR_METRIC)));
        }
        catch (NumberFormatException ne) {
          metricType = Metric.INDICATOR_METRIC;
        }
        metricName = pProps.getProperty(metricId + ".name", "");
        metricReset = Boolean.valueOf(pProps.getProperty(metricId + ".reset", "true")).booleanValue();
        metricDescription = pProps.getProperty(metricId + ".desc", "");
        try {
          mMetrics.put(metricId, new Metric(metricType, metricId, metricName,
                                            metricReset, metricDescription));
//          log.info("Registered Metric " + metricId + ".");
        }
       catch (Exception propsError) {
//          log.error("Can't register Metric: " + metricId, propsError);
        }
      }
    }

    /**
     * Allows a client to explicity shutdown the metric manager
     * causing all metrics to be logged, the timer thread to be
     * stopped, and the internal reference for the singleton pattern
     * to be cleared.
     */
    public void shutdown() {
```

```
    mTimer.killTimer();
        mTimer.interrupt();
    mTimer = null;
    logMetrics();
    mMetrics.clear();
    mInstance = null;
  }

  /**
   * Called during each expiration of the timer in order to log current
   * metric totals.
   */
  public void timerExpired() {
    this.logMetrics();
  }

  /**
   * Adds to a metrics value, accounting for the type of metric.
   *
   * @param      pMetricId      String id of the metric to add value to.
   * @param      pMetricValue   Value to be added to this metric.
   */
  public void add(String pMetricId, int pMetricValue ) {
    Metric toAdd = (Metric)mMetrics.get(pMetricId);
    if ( toAdd == null ) {
//              log.warn("Metric Id (" + pMetricId + " does not exist.", new InvalidMetricException());
        }
        else {
      toAdd.add(pMetricValue);
    }
  }

  /**
   * Adds to a metrics value, accounting for the type of metric.
            * @param      pMetricId      String id of the metric to add value to.
   * @param      pMetricValue   Value to be added to this metric.
   */
  public void add(String pMetricId, long pMetricValue) {
    Metric toAdd = (Metric)mMetrics.get(pMetricId);
    if ( toAdd == null ) {
//              log.warn("Metric Id (" + pMetricId + " does not exist.", new InvalidMetricException());
        }
        else {
      toAdd.add(pMetricValue);
    }
  }

  /**
  * Adds to a metrics value, accounting for the type of metric.
   *
   * @param      pMetricId      String id of the metric to add value to.
   * @param      pMetricValue   Value to be added to this metric.
   */
  public void add(String pMetricId, java.util.Date pMetricValue) {
    Metric toAdd = (Metric)mMetrics.get(pMetricId);
    if ( toAdd == null ) {
//              log.warn("Metric Id (" + pMetricId + " does not exist.", new InvalidMetricException());
        }
        else {
      toAdd.add(pMetricValue);
    }
  }
```

```
  /**
   * Adds to a metrics value, accounting for the type of metric.
   *
   * @param      pMetricId      String id of the metric to add value to.
   * @param      pMetricValue   Value to be added to this metric.
   */
  public void add(String pMetricId, GregorianCalendar pMetricValue) {
    Metric toAdd = (Metric)mMetrics.get(pMetricId);
    if ( toAdd == null ) {
//              log.warn("Metric Id (" + pMetricId + " does not exist.", new InvalidMetricException());
          }
          else {
      toAdd.add(pMetricValue);
    }
  }

  /**
   * Returns the specified metric's value.
   *
   * @param      pMetricId      String id of the metric to add value to.
   *
   * @return     Value of the metric, 0 is returned if the metric is not an
  *            indicator or static metric type.
   */
  public long getValue(String pMetricId) {
    Metric toRetrieve = (Metric)mMetrics.get(pMetricId);
    if ( toRetrieve == null ) {
//              log.warn("Metric Id (" + pMetricId + " does not exist.", new InvalidMetricException());
                return 0;
          }
          else {
      return toRetrieve.getValue();
    }
  }

  /**
   * Returns the specified metric's average value.
   *
   * @param      pMetricId      String id of the metric to retrieve.
   *
   * @return     Value of the metric, 0 is returned if the metric is not an
   *            indicator or static metric type.
   */
  public float getAverage(String pMetricId) {
    Metric toRetrieve = (Metric)mMetrics.get(pMetricId);
    if ( toRetrieve == null ) {
//              log.warn("Metric Id (" + pMetricId + " does not exist.", new InvalidMetricException());
                return 0;
          }
          else {
      return toRetrieve.getAverage();
    }
  }

  /**
   * Returns the specified metric's TimeStamp value.
  *
   * @param      pMetricId      String id of the metric to retrieve.
   *
   * @return     The Date of the metric or null.
   */
```

```
  public GregorianCalendar getTimeStamp(String pMetricId) {
    Metric toRetrieve = (Metric)mMetrics.get(pMetricId);
    if ( toRetrieve == null ) {
//              log.warn("Metric Id (" + pMetricId + " does not exist.", new InvalidMetricException());
                return null;
            }
            else {
      return toRetrieve.getTimeStamp();
    }
  }

  /**
   * Returns the specified metric's count of data points.
   *
   * @param      pMetricId      String id of the metric to retrieve.
   *
   * @return     Count of datapoints added to this metric.
   */
  public long getDataPointCount(String pMetricId) {
    Metric toRetrieve = (Metric)mMetrics.get(pMetricId);
    if ( toRetrieve == null ) {
//              log.warn("Metric Id (" + pMetricId + " does not exist.", new InvalidMetricException());
                return 0;
            }
            else {
      return toRetrieve.getDataPointCount();
    }
  }

  /**
   * Logs all the metrics to the file.  This routine is called by another
   * thread that is operating on a timer.  The thread controls when this
   * method is called.
   */
  public void logMetrics() {
    Metric toLog;

//    log.info(mHost + " -> Total Memory : " + Runtime.getRuntime().totalMemory() + " bytes.");
//     log.info(mHost + " -> Free Memory : " + Runtime.getRuntime().freeMemory() + " bytes.");
    for ( Enumeration allMetrics = mMetrics.elements() ; allMetrics.hasMoreElements() ;) {
      toLog = (Metric)allMetrics.nextElement();
//       log.info(mHost + " -> " + toLog.getName() + " : " + toLog.toString());
    }
//     log.debug(mHost + " all metrics logged.");
  }

  /**
   * Resets all metrics to zero or null after performing a log of the current
   * values.
   */
  public void reset() {
    Metric toReset;
    this.logMetrics();  // log all the metrics
    for ( Enumeration allMetrics = mMetrics.elements() ; allMetrics.hasMoreElements() ;) {
      toReset = (Metric)allMetrics.nextElement();
      toReset.reset();
    }
//     log.debug(mHost + " reset all metrics.");
  }

  /**
   * Resets a metric to zero or null after performing a log of the metric.
```

```
    *
    * @param      @pMetricId     Metric id to reset
    */
   public void reset(String pMetricId) {
      Metric toReset = (Metric)mMetrics.get(pMetricId);
      if ( toReset != null ) {
         toReset.reset();
      }
//      log.debug(mHost + " reset Metric: " + pMetricId + ".");
   }
}

/**
 * Stores constants used to identify the various metrics that are
 * managed by the MetricManager.
 *
 * @see MetricManager
 */
public class MetricId {
   /**
    * Constant used to store metric id for application start date and time.
    */
   public static final String APPLICATION_STARTED = "m1";

   /**
    * Constant used to store metric id for application running time.
    */
   public static final String RUNNING_TIME = "m2";

   /**
    * Constant used to store metric id for connection attempts.
    */
   public static final String CONNECTIONS_ATTEMPTED = "m3";

   /**
    * Constant used to store metric id for connection attempts.
    */
   public static final String CONNECTIONS_COMPLETED = "m4";

   /**
    * Constant used to store metric id for total bytes received by the application.
    */
   public static final String BYTES_RECEIVED = "m5";

   /**
    * Constant used to store metric id for count of request processors.
    */
   public static final String REQUEST_PROCESSORS = "m6";

   /**
    * Constant used to store metric id for average connection time.
    */
   public static final String CONNECT_TIME = "m7";
}
```

Додаток С

```
import java.io.*;
import java.util.*;

/**
 * Interface to be implemented by objects wishing to use the Timer class
 * functionality.  The timer class can accept a number of listeners that
 * implement this interface.
 *
 * @see TaskTimer
 */
public interface TimerCallBack {
   /**
    * This method is called by the Timer for each object registered as a
    * call back.  The Timer executes a wait loop for the specified timer
    * interval and then calls all the listening objects.
    */
   public void timerExpired();
}

import java.io.*;
import java.util.*;
//import org.apache.log4j.Category;

/**
 * This is a generic timer used by other classes.  An instance of TaskTimer
 * is given a callback (any object implementing the TimerCallBack interface)
 * which is called everytime the timer expires.  The length of the timer is
 * a millisecond value set in the constructor or directly.  The timer uses
 * a separate thread for execution by extending the default Thread.
 *
 * @see  TimerCallBack
 */
public class TaskTimer extends Thread {
//   static Category log = Category.getInstance(TaskTimer.class.getName());
   private Vector mTimerCallBackList = new Vector();
   private long mTimerInterval = 60000;  // milliseconds between calls.
   private boolean mKeepRunning = true;
   private String mThreadName;

   /**
    * The minimum allowed value of the timer interval.  The timer must call
    * each call back object each interval of the timer and therefore must have
    * enough time to make the appropriate calls.  Since the timer value is milliseconds
    * a minimum value assures the timer doesn't spend all it's time making
    * calls to the TimerCallBack objects.
    */
   public static final long MINIMUM_TIMER_INTERVAL = 5000;

   /**
    * Default constructor creates a TaskTimer with not established call back objects.
    */
   public TaskTimer() {
   }

   /**
    * Construct a TaskTimer object with the specified timer setting.
```

```
 *
 * @param      pTimerInterval     millisecond interval value.
 */
public TaskTimer(long pTimerInterval) {
  this.setTimerInterval(pTimerInterval);
}

/**
 * Construct a TaskTimer object with the specified timer name and timer setting.
 *
 * @param      pThreadName       name for this thread.
 * @param      pTimerInterval     millisecond interval value.
 */
public TaskTimer(String pThreadName, long pTimerInterval) {
  this.setTimerInterval(pTimerInterval);
  mThreadName = pThreadName;
}

/**
 * Construct a TaskTimer object with the specified call back, name, and timer setting.
 *
 * @param      pCallBack         Object implementing the TimerCallBack interface that is
 *                               to be called each time the timer expires.
 * @param      pThreadName       name for this thread.
 * @param      pTimerInterval    millisecond interval value.
 */
public TaskTimer(TimerCallBack pCallBack, String pThreadName, long pTimerInterval) {
  this.setTimerInterval(pTimerInterval);
  mThreadName = pThreadName;
  this.addTimerCallBack(pCallBack);
}

/**
 * Construct a TaskTimer object with the specified call back.
 *
 * @param      pCallBack         Object implementing the TimerCallBack interface that is
 *                               to be called each time the timer expires.
 */
public TaskTimer(TimerCallBack pCallBack) {
  this.addTimerCallBack(pCallBack);
}

/**
 * @return     The current timer interval setting.
 */
public long getTimerInterval() {
  return mTimerInterval;
}

/**
 * Set the timer interval.
 *
 * @param      pTimerInterval    Interval value in milliseconds, if less than
 *                               MINIMUM_TIMER_INTERVAL then timer will be set equal
 *                               to the minimum.
 */
public void setTimerInterval(long pTimerInterval) {
  if ( pTimerInterval < MINIMUM_TIMER_INTERVAL ) {
    mTimerInterval = MINIMUM_TIMER_INTERVAL;
  }
  else {
    mTimerInterval = pTimerInterval;
```

```
  }
}

/**
 * Adds an object that will be called each iteration of the timer.
 *
 * @param      pCallBack      Object to be called each time the timer expires.
 */
public synchronized void addTimerCallBack(TimerCallBack pCallBack) {
  mTimerCallBackList.add(pCallBack);
}

/**
* Removes an existing timer call back object from the available call
 * back list.
 *
 * @param      pCallBack      Object to be called each time the timer expires.
 */
public synchronized void removeTimerCallBack(TimerCallBack pCallBack) {
  mTimerCallBackList.remove(pCallBack);
}

/**
 * Main timer loop runs until shutdown command is received. The thread will sleep
 * for the designated timer interval and then call back to all registered objects.
 */
public void run() {
        if ( mThreadName != null ) {
    Thread.currentThread().setName(mThreadName);
  }
  while ( mKeepRunning ) {
    try {
      Thread.currentThread().sleep(mTimerInterval);
      if ( mKeepRunning ) {  // Recheck whether we should be running.
        ListIterator callBackList = mTimerCallBackList.listIterator();
        while ( callBackList.hasNext() ) {
          TimerCallBack callBackObject = (TimerCallBack) callBackList.next();
          callBackObject.timerExpired();
        }
      }
    }
    catch (InterruptedException ie) {
      ;  // The thread was interrupted so we loop and start again.
    }
  }
}

/**
 * This stops the timer, after the current interval expires.
 */
public void stopTimer() {
  mKeepRunning = false;
}

/**
 * Stops the timer and clears all references to call back items.
 */
public synchronized void killTimer() {
        mKeepRunning = false;
        mTimerCallBackList.clear();
        notifyAll();
   }
```

Додаток Д. Демонстраційні матеріали

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ТЕЛЕКОМУНІКАЦІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка web-додатку для статичного дослідження метрик прикладного програмного забезпечення на мові JAVA

Виконав студент 5 курсу

групи ППЗ-52
Кукла Артем Володимирович
Керівник роботи

Коваленко Данило Сергійович

Київ – 2021

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

- **Мета роботи** розрахунок статичних метрик програмного коду на мові Java
- **Об'єкт дослідження** мова програмування java
- **Предмет дослідження** основні метрики програмного коду на мові Java

ТЕХНІЧНІ ЗАВДАННЯ

1. Аналізувати статичний аналіз метрики прикладного ПЗ
2. Визначитися з метриками
3. Написання програми для статичного аналізу прикладного програмного забезпечення на мові java
4.Роль статичного аналізу в новітніх технологіях
5.Підсумки дослідження

3

ПОСЛІДОВНІСТЬ ПРОВЕДЕННЯ СТАТИЧНОГО АНАЛІЗУ ПЗ

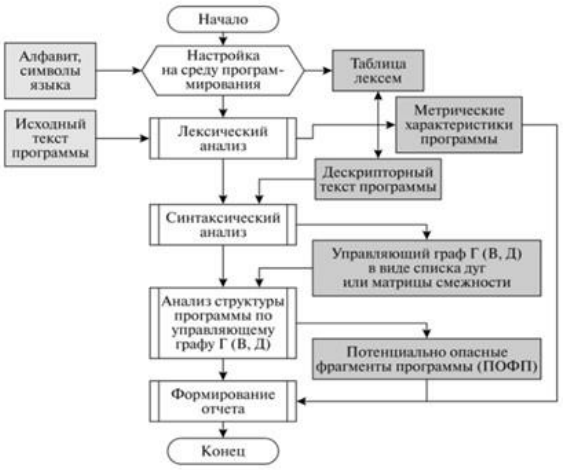
Статичний аналіз

	Час виявлення дефекту				
час внесення дефекту	Виробка вимог	Проектування архітектури	Кодування	Тестування	Після випуску ПЗ
Виробка вимог	1	3	5-10	10	10-100
Проектування архітектури	-	1	10	15	25-100
Кодування	-	-	1	10	10-25

Статичний аналіз

4

СХЕМА АЛГОРИТМУ СИСТЕМИ СТАТИЧНОГО АНАЛІЗУ



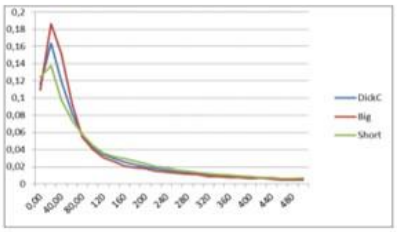
5

ПРОГРАМНЕ ЗАБЕЗПЕННЯ АНАЛІЗУ МЕТРИК БАЙТ-КОДУ

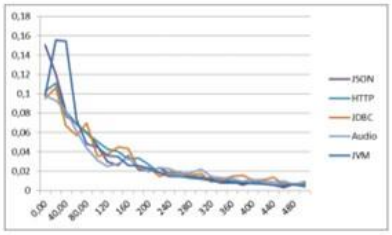
Назва	Ліцензія		Масовий аналіз
Dependency Finder	Вільна (BSD License 2.0)	2010	Так
JDepend	Вільна (BSD License)	2005	Так
CyVis	Вільна (GNU General Public License)	2006	Ні
Ckjm	Вільна (Apache License)	2014	Ні
Jqassistant	Вільна (GPLv3 License)	2018	Ні
Закінчення Таблиці 2.1			
JArchitect	пропрієтарна	2018	Так
Jtest	пропрієтарна	2018	Так
Sonargraph	пропрієтарна	2018	Так

6

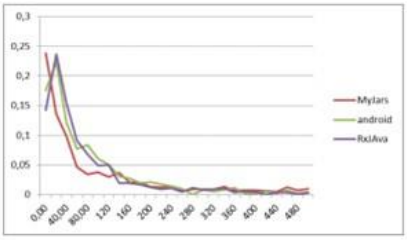
ОЦІНКА ЗІБРАНИХ ДАНИХ



Розподіл метрики MCS для наборів групи 1



Розподіл метрики MCS для наборів групи 2

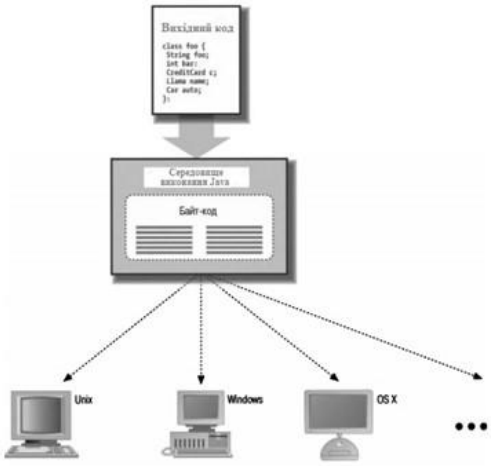


Розподіл метрики MCS для наборів групи 3

Підсумкові параметри оцінки законів розподілу для різних метрик

Метрика	MCS	WMC	CBO	DIT	NoC	NoF	NoM	I	NoFM	Ce	Ca	JNoC	JCS	JFS
Критерий*	0,037	0,128	0,07	0,262	0,069	0,240	0,148	0,120	0,084	0,081	0,024	0,047	0,069	0,041
P-Value	0,993	0,839	0,94	0,215	0,999	0,151	0,695	0,418	0,996	0,997	0,130	0,973	0,984	0,999
Критерий**	0,121	0,287	0,19	0,338	0,287	0,287	0,287	0,187	0,287	0,287	0,28	0,134	0,21	0,154
Распределение	4	2	4	3	7	3	6	1	2	5	3	6	5	2
Ячеек гистограммы	126	21	51	15	21	21	21	51	21	21	22	100	40	75
Шаг выборки	4	1	1	1	1	1	1	0	1	1	1	1	400	2000
Диапазон	504	21	51	15	21	21	21	1,02	21	21	22	100	16000	2E+05
Ранг распределения	1	1	1	1	1	1	4	2	1	1	1	2	1	3

СЕРЕДОВИЩЕ ВИКОНАННЯ JAVA



9

ПРИКЛАД ЛІСТИНГУ ВЕБ-ДОДАТКУ ДЛЯ СТАТИЧНОГО ДОСЛІДЖЕННЯ МЕТРИК ПРИКЛАДНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА МОВІ JAVA

```
#
# MetricManager.properties
# Startup parameters for the MetricManager.
#
# host name used for recording metrics.
# Leave blank to use machine's host name
#host-myname

# Log interval.  Number of milliseconds between log writes.
# Minimum is 60000 (1 minute)
logInterval=300000
#logInterval=60000

# Comma separated list of metrics that are available.
metrics=m1,m2,m3,m4,m5,m6,m7
#
# Individual metric entries follow:
#
# Syntax:
# <MetricID>.type=<Type of metric>
# <MetricID>.name=<String name for the metric to be used in log output>
# <MetricID>.reset=<true | false>
# <MetricID>.description=<Description to be used in queries etc.>
#
# Note: Metric ids cannot be re-assigned without significant code
#       changes since
#       the id is used by classes for recording their metrics.
#       This is a todo item to make these more easily changed
#       ...if necessary!.
#
# Example:
# m1.type=3
# m1.name=Attempted Connections
# m1.reset=true
# m1.desc=Total number of connection attempts made since start.
#
# Available types:
#   TIME_STAMP_METRIC   = 1
#       A date time.  This metric is set once and simply stored for
retrieval.
#   ELAPSED_TIME_METRIC = 2
#       A date time that returns the time elapsed in milliseconds.
#   INDICATOR_METRIC    = 3
#       A normal numerical metric that is constantly increased by new
values.
#   STATIC_METRIC       = 4
#       A numerical metric that is replaced with incoming values.
#   AVERAGE_METRIC      = 5
#       Returns the average (float) of a number of measurements taken over
time.
```

10

ВИСНОВКИ

Застосування статичних метрик у загальному випадку дозволяє визначити складність розробленого проекту, оцінити обсяг виконаних робіт, стилістику розроблюваної програми і зусилля, які докладені кожним розробником для реалізації того чи іншого рішення.
Статичне дослідження метрик ПЗ на мові Java дозволяє виконати:

- перевірку модульної структури ПЗ, а також логічної структури окремих модулів і порівняння цих структур з наведеними в програмної документації;
- підготовку вихідних даних для проведення динамічного аналізу ПЗ і розробки плану тестування ПЗ;
- оцінку конструктивних характеристик програми, ступеня простоти модифікації і супроводу програми;
- визначення наявності недосконалостей в програмі, невикористовуваних ділянок програми, зайвих змінних;
- оцінку текстової складності програми, витрат на її розробку і освоєння;
- експертизу ідентичності програм для встановлення авторства і вирішення правових спорів;
- визначення кількісних характеристик при оцінці рівня якості програми.

Для реалізації вищенаведених завдань було розроблено веб-додаток на мові програмування Java

ДЯКУЮ ЗА УВАГУ!