УДК 004.45
DOI https://doi.org/10.32782/2663-5941/2024.5.1/38

Прокоф'єв І.Г.
Хмельницький національний університет

Савенко О.С.
Хмельницький національний університет

Медзатий Д.М.
Хмельницький національний університет

МЕТОД СТАТИСТИЧНОГО АНАЛІЗУ ПРОГРАМНОГО КОДУ ВИКОРИСТОВУЮЧИ КОМПЛЕКСНІ ПАРАМЕТРИ

Програмне забезпечення поширилося на всі сфери людської діяльності – економіка, наука та бізнес не можуть працювати без програмного забезпечення, а мільйони людей залучені до його написання та підтримки, а це створює безпрецедентний вплив на виклики безпеки. Неважливо, чи це військова сфера, чи роздрібна торгівля, але сьогодні жодна галузь не обходиться без якісного програмного забезпечення, і навіть невеликий збій у розподіленій системі з великою кількістю користувачів може спричинити збій у інших дочірніх галузях. Можна сказати, що з 1990 року кількість програмного забезпечення надзвичайно значно зросла, оскільки цей період ознаменувався широким розвитком комп'ютерних технологій і поширенням комп'ютерів та Інтернету. Протягом цього періоду було написано та випущено багато програмного забезпечення в різних областях, таких як операційні системи, бізнес-програми, ігри, веб-програми, мобільні програми та багато іншого. Крім того, багато програмного забезпечення регулярно оновлюється і розширюється, тому створення нових версій і підтримка існуючих програм триває і сьогодні. Більше того, програмне забезпечення фактично стало звичайним товаром, коли окремі компанії продають рішення API (Application Programming Interface), SaaS (Software as a Service), і це не відрізняється від звичайних ресурсів у сфері енергосервісу чи товарообігу супермаркетів згідно з підходу «Плати – отримуй». У зв'язку з високим навантаженням у популярних програмних рішеннях використовуються нові архітектурні підходи до написання програмного забезпечення, що значно підвищує складність написання підтримки такого програмного забезпечення. І якщо раніше тільки технічні гіганти, такі як Google і Apple, дійсно потребували унікальних алгоритмів для створення високонавантаженого програмного забезпечення, то в 2024 році тренд мікросервісної архітектури вимагає міцних знань побудови розподілених систем. Все це накладає суттєві обмеження на новостворене програмне забезпечення – якість програмного забезпечення має бути вимірюваною величиною і ця цінність має бути принаймні контрольованою як розробниками, так і керівництвом. Якісне програмне забезпечення можливе лише за наявності якісного коду, який ми розглянемо далі. Сучасні програмні системи можуть мати велику складність, тому дуже важливо підтримувати існуючі системи та вимірювати якість. У цій статті розглянуто різні методи оцінювання якості та безпеки коду в сучасних програмних системах. Було проаналізовано різні відомі показники якості коду та те, як їх можна застосувати в залежності від мови програмування. Метою статті є огляд існуючих показників якості програмного коду та запропоновано нову стратегію аналізу якості програмного коду. Дослідницька робота зосереджена на пошуку нових методів, заснованих на машинному навчанні, які дозволяють знаходити різні анти-патерни та codesmell по-новому. Результатом методу є виявлення загальної якості конкретного програмного забезпечення, написаного C++ подібною мовою програмування як C# чи Java

Ключові слова: *якість програмного коду, статичний аналіз коду, показники якості коду.*

Постановка проблеми. На сьогодні достеменно не відомо яка кількість програмного забезпечення (ПЗ) створюється в рік, але можна чітко передбачити, що цей процес не знизиться в найближчі роки, особливо після буму AI технологій у 2020-х роках. Якщо на початку 90-х років минулого століття програмний код в основному писався співробітниками військових інституцій та ентузіастами (Лінукс на Netscape Navigator), то на сьогодні у написання коду залучені також технічні гіганти, які створюють та розповсюджують власні фреймворки (Android та Angular від Google,

React від Фейсбук, .NET від Microsoft). Фреймворки та бібліотеки значно спрощують роботу пересічним програмістам та бізнесу і дозволяють сфокусуватись на вирішенню конкретних проблем без витрачання часу на інфраструктурні речі і наслідком є створення складних розподілених системи реального часу з мільйонами користувачів. В цьому контексті якість коду є не побічною проблемою а мейнстрімом, оскільки втрати якості ПЗ може відбутися внаслідок досить не суттєвих змін у коді що може призвести до повної зупинки критичних сервісів роботи.

«Якість коду» [1] означає загальну якість вихідного коду програмного продукту. Це значення показує, наскільки добре код написаний, організований і підтримується. Якість коду забезпечується кількома принципами: читабельність, масштабованість, продуктивність, безпека, тестування та налагодження. Читабельність [2] означає, що високоякісний код легко читати та розуміти. Це полегшує розробникам співпрацю та підтримку коду з часом. Ремонтопридатність – це коли код, який добре організований і відповідає найкращим практикам, легше підтримувати. Це дозволяє ефективно виправляти помилки, оновлювати та додавати нові функції без появи нових помилок. Масштабованість – це коли якість коду часто пов’язана з тим, наскільки добре програма може масштабуватися. Поганий код може не впоратися зі збільшенням навантаження або зміною вимог. Продуктивність означає, що добре написаний код може бути ефективнішим і мати кращу продуктивність. Це також може зменшити споживання ресурсів, зробивши програмне забезпечення більш чутливим і менш ресурсоємним. Безпека коду тісно пов'язана з безпекою. Поганий код може створювати вразливості, якими можуть скористатися зловмисники. Хороші практики якості коду можуть допомогти зменшити ризики безпеки. Тестування та налагодження відноситься до випадку, коли високоякісний код легше перевірити та налагодити. Це зменшує час і зусилля, необхідні для виявлення та усунення проблем.

Отже, процес визначення якості коду не є тривіальною роботою. І головна проблема полягає в тому, щоб знайти список або правила, які можуть визначити, де конкретний код потребує критичних змін, а де ні. Мета полягає не лише у визначенні деяких правил, які можна легко знайти в Google, а й у пошуку найефективніших способів визначення поганого/проблемного коду програмного забезпечення. Недостатньо просто знайти цикломатичну складність або обчислити кількість рядків коду. Також, сюди відноситься й виявлення code-smell та анти-патернів.

Метод визначення якості програмного забезпечення здійснюється в кілька етапів: компіляція → статичний аналіз [3] → покриття тестами [4] → динамічний аналіз → збір даних ітерації AAA та оцінка -> … зміна коду. База коду не є статичною величиною, її можуть щодня змінювати десятки розробників, особливо якщо використовується ПЗ, яким користуються мільйони людей, наприклад Gmail, Microsoft Office 365, X (раніше Twitter), Netflix. Іншими словами, процес визначення якості коду є перманентним, і кожні нови зміни коду ініціюють нову ітерацію, починаючи з компіляції. Метрики або атрибути якості програмного забезпечення зазвичай відповідають наступним показникам якості коду: цикломатична складність, кількість рядків коду (LOC), покриття коду, дублювання коду, порушення статичного аналізу коду, індекс ремонтопридатності, відтік коду, технічний борг. Розглянемо їх.

Цикломатична складність [5] вимірює складність програми шляхом підрахунку кількості лінійно незалежних шляхів у вихідному коді програми. Висока цикломатична складність може вказувати на складний код, який важко підтримувати. Кількість рядків коду (LOC) [6] це проста, але базова метрика, яка вимірює розмір кодової бази шляхом підрахунку кількості рядків у файлі вихідного коду. Хоча це не є прямим показником якості, занадто довгі методи або класи можуть вказувати на погану організацію коду. Покриття коду вимірює відсоток коду, який виконується автоматичними тестами. Більше охоплення коду зазвичай вказує на більш повну перевірку бази коду. Дублювання коду вимірює кількість дубльованого коду в базі коду. Дубльований код може призвести до проблем із підтримкою та невідповідностей. Порушення статичного аналізу коду вимірює кількість порушень стандартів кодування або найкращих практик, виявлених інструментами статичного аналізу коду. Поширені проблеми включають невикористані змінні, неправильну обробку помилок і порушення правил кодування. Індекс ремонтопридатності – це загальний показник, який поєднує різні атрибути коду, такі як складність, дублювання коду та зміни коду, щоб оцінити загальну ремонтопридатність бази коду. Відтік коду – вимірює частоту змін коду з часом. Значна зміна коду може свідчити про нестабільність або часті рефакторинги, що може вплинути на якість програмного забезпечення. Технічний борг означає зусилля, необхідні для вирішення неоптимальних варіантів дизайну або реа-

лізації в базі коду. Високий технічний борг може уповільнити швидкість розробки та збільшити ймовірність помилок. Управління залежностями вимірює складність і стабільність залежностей у базі коду. Надлишкові залежності або застарілі бібліотеки можуть створювати вразливості та додаткові витрати на обслуговування. Показники перевірки коду пов'язані з процесами кодування, такими як час виконання перевірки, участь рецензента та кількість проблем, виявлених під час перевірки. Розрахунок цикломатичної складності в C++ і об'єктно-орієнтованих мовах програмування (ООП) передбачає аналіз потоку керування кодом, який може включати як процедурні, так і об'єктно-орієнтовані конструкції. В ООП, таких як C++, також потрібно враховувати додаткові фактори – методи класу та зв'язки класу. Методи класу в межах класу можуть містити логічні оператори, такі як оператори if або цикли, які сприяють загальній цикломатичній складності класу. Якщо використовуються успадкування та поліморфізм, потік керування може стати більш складним через перевизначення методу та динамічну диспетчеризацію. Кожен перевизначений метод додає пункти прийняття рішення до складності. Якщо код передбачає взаємодію між класами, наприклад виклики методів або співпрацю, ці взаємодії також необхідно враховувати при обчисленні цикломатичної складності.

Інколи до якості ПЗ також включають атрибути пов'язані з безпекою. На сьогодні найбільш поширеним типом аплікацій є інтернет сайти за рахунок відносної простоти створення – системи керування контенту та фреймворки дають про себе знати. І при застосуванні навіть досить простої розподіленої [7, 8] бот атаки більшість простих сайтів просто зупиниться і не буде працювати. І якщо раніше в 2000-х роках DDOS атаки можна було припинити шляхом встановленням додаткового мережевого обладнання, то з появою botnets для забезпечення функціонуванню сайту роль якості ПЗ є однією з перших і мова не про прості sql ін'єкції та відомі OWASP правила, а перш за все з продуманню правильної архітектури.

До безпекових викликів також належать і мобільні застосунки ріст яких припав на 2010-і роки в зв'язку з поширенням iOS/Android [9] ОС коли зловмисник може спробувати поцупити дані на фізичному пристрої. Ізраїльська розвідка давно використовує ПЗ що зламує приватні дані на будь-якому пристрої, але з встановленою ОС більш ранніх поколінь. Досить поширеним також є злом застосунків Інтернету Речей IoT [10].

Після розгляду усіх існуючих атрибутів якості ПЗ виявилось що метрика Кемерера є досить суттєвою. Сама метрика була запропонована Реймондом Кемерером для оцінки складності та тестування програмних систем, особливо об'єктно-орієнтованих. Він враховує кількість класів програми, кількість інтерфейсів і залежностей між класами. Метрика Кемерера базується на метриках залишкових з'єднань і стабільності пакетів в архітектурі програми.

Обидва ці показники можуть бути корисними для оцінки складності програмного коду та архітектури програмних систем. Вони допомагають визначити потенційні проблеми та області для вдосконалення програм і надають огляд рівня складності програми.

Показники успадкування – це показники, які використовуються для вимірювання характеристик успадкування в об'єктно-орієнтованих програмах. Спадкування в ООП визначає, як класи можуть успадковувати властивості та методи від інших класів.

Аналіз останніх досліджень і публікацій. Перш, ніж визначати власні/спеціальні методи та алгоритми, розглянемо існуючі методи виявлення атрибутів якості коду та роботу, виконану в цій області. Є кілька популярних комерційних програмних інструментів, таких як SonarQube [11], DeepSource або Embold, але їхні алгоритми недоступні, навіть якщо вони засновані на атрибуті Cyclomatic Complexity і Code Coverage.

Більшість дослідницьких статей зосереджено на конкретних атрибутах якості коду, але вони не шукають залежності між атрибутами або не акцентують на цьому достатньої уваги.

Автор [11] описав емпіричну перевірку об'єктно-орієнтованих показників дерева глибини успадкування (DIT) і кількості нащадків (NOC), запропонованих Чідамбером і Кемерером.

Показано, що різні інструменти статичного аналізу для C++ подібних мов можуть мати різні результати, і не існує золотого стандарту для визначення вразливостей коду. На завершення: кожен з різних інструментів статичного аналізу коду має свої переваги та недоліки.

Відповідно до статичного аналізу інструменти для виявлення дефектів виконання та вразливостей безпеки можна розділити на кілька груп.

Результати експерименту в роботі [12] показують, що системи без успадкування легше модифікувати, ніж системи з трьома або п'ятьма рівнями успадкування. Це відкриття викликає питання про ефективність використання спадковості.

Ідея цього дослідження полягає не тільки в тому, щоб визначити список атрибутів якості коду або переглянути їх ще раз, але й у тому, щоб встановити пріоритети між ними. Тому, визначальними є наступні питання, які потребують опрацювання:

1) визначити атрибути якості ПЗ, якими можна знехтувати, та встановити етапи на яких це можна зробити;

2) визначити достатність використання атрибуту цикломатичної складності;

3) визначити можливість вибору лише частини із атрибутів якості програмного забезпечення для досягнення належного результату тестування;

4) визначити потребу у постійному врахуванні атрибуту Code Coverage.

Постановка завдання. Метою статті є отримання оптимального алгоритму для визначення якості коду виходячи з існуючих параметрів якості коду. Іншими словами, маючи масив параметрів A..Z потрібно встановити, які з них мають найбільший вплив. Постає вибір щодо атрибутів якості ПЗ, якими можна знехтувати у випадку складних систем з мільйоном рядків коду.

Виклад основного матеріалу. Мета методу полягає в аналізі існуючої кодової бази, написаної на C++ або схожій мові, і пошуку/виявлення відомих зразків коду та антишаблонів на основі загальних і спеціальних правил. Відповідно до загальних правил переглядаємо атрибути якості коду, такі як рядки коду, цикломатична складність, показники Чідамбера та Кемерера. Відповідно до спеціальних правил розглядаємо додаткові алгоритми, які можуть ідентифікувати конкретні антишаблони. На відміну від пропрієтарних систем, таких як SonarQube [13] та Embold, мета пропонованого методу полягає в тому, щоб розширити інструмент аналізу коду додатковими правилами. Іншими словами, користувач може написати власні правила, а система повинна «вивчити» та ідентифікувати такі правила в майбутніх ітераціях. Мова програмування і середовище виконання повинні бути обрані заздалегідь, оскільки мови ООП сильно відрізняються від процедурних і, відповідно, повинні застосовуватися зовсім інші правила і шаблони. Це може бути будь-яка мова, схожа на C++, як-от Java, PHP тощо. Має автоматично виявлятися на основі розширення файлу. Деякі пункти можуть бути визначені користувачем, коли, наприклад, вихідна папка містить суміш розширень, але для нас важливо використовувати лише певні.

Розглянемо основні кроки алгоритму.

1. Компіляція коду.

2. Підготовка попередньо визначених правил для статичного аналізу.

3. Встановлення мінімальних/максимальних значення для наступних змінних, які обчислюватимуться на рівні класу/методу: Cyclomatic Complexity, Number Of Lines, Response For Class.

4. Рекурсивний пошук файлів на основі розширення (розширень): *.cpp, *.css.

5. Створення набору класів і підмножин методів: знаходження кількості методів, полів, вкладених класів.

6. Розрахунок атрибутів по кожному класу.

7. Розрахунок антишаблонів [14] кожного класу.

8. Знаходження дублікатів коду кожного методу у форматі «клас – метод – кількість дубльованих рядків».

9. Збереження обчислених даних у зовнішньому сховищі, наприклад БД.

На кроці компіляції коду можна аналізувати лише скомпільований код, оскільки не скомпільований код не завжди розпізнається з точки зору аналізатору. Метою цих кроків є застосування відповідного компілятора на основі обраної мови програмування та середовища. Наприклад, компілятор Roslyn слід застосовувати, якщо у нас є ПЗ введення ASP.NET, написане мовою C#.

Підготовка попередньо визначених правил для статичного аналізатора щодо обраної мови програмування містить наступні правила: правила форматування коду, правила іменування, правила безпеки коду, правила коментування коду, правила використання бібліотек і фреймворків, правила розподілу відповідальності. Правила форматування коду – це правила, які визначають стиль написання коду, наприклад відступи, розміщення дужок, розмір і розміщення пробілів тощо. Правила іменування визначають правила іменування змінних, функцій, класів та інших елементів програмного коду. Правила безпеки коду спрямовані на виявлення потенційних проблем безпеки, таких як можливі вразливості, витоки пам’яті, неправильна обробка винятків тощо. Правила коментування коду перевіряють наявність і якість коментарів у програмному коді, допомагаючи переконатися, що код є зрозумілим і задокументованим. Правила використання бібліотек і фреймворків перевіряють відповідність використання бібліотек і фреймворків певним стандартам і найкращим практикам. Правила розподілу обов'язків (SRP) перевіряють, чи виконують класи та модулі в програмному коді лише одну відповідальність.

Визначення класів і методів.

1. Читання та розбір файлів, видалення порожніх символів. На цьому етапі відбувається парсинг файлів, ідентифікація унікальних класів.

2. Розкладка класу.

3. Ідентифікація унікальних класів.

4. Визначення сторонніх бібліотек і фреймворків.

5. Визначення зв'язків між методами та класами.

6. Аналіз стилів коду.

7. Визначення точки входу в програму – не весь код і класи можуть бути задіяні тому важливо розуміти де сама точка входу в програмі. Приклад – ПЗ може містити 100 класів, але реально лише 5 використовуються.

Визначення ознак якості ПЗ кожним методом.

1. Визначення кількості рядків коду.

2. Розрахунок цикломатичної складності.

3. Визначення зважених методів для кожного класу.

4. Виконання методів у класі. Обчислення загальної кількості методів у класі.

5. Оцінка трудомісткості кожного методу. Для кожного способу потрібно визначити його складність. Для цього найчастіше використовується цикломатична складність, яка вимірює кількість незалежних шляхів через метод. Метод підсумовування ваги. Сумма значень складності для всіх методів у класі. Якщо всі методи мають однакову вагу (наприклад, вага 1), то здійснити визначення глибини дерева успадкування [15], ідентифікувати всі класи та їхніх суперкласів. Для цього потрібно зібрати усі класи в кодовій базі разом із їхніми прямими суперкласами. Інакше ініціалізувати значення глибини дерева успадкування. Створити словник для зберігання значення глибини дерева успадкування для кожного класу, ініціалізуючи кожне значення DIT рівним 0. Здійснити обхід дерева успадкування. Для кожного класу потрібно пройти його ланцюжок успадкування до кореневого класу та обчислити кількість рівнів.

6. Визначення відповіді для класу. Визначити всі методи в класі. Знайти всі методи, визначені в класі. Визначити зовнішні методи, викликаних кожним методом класу. Для кожного методу класу знайдіть усі методи (як у тому ж класі, так і в інших класах), які він викликає. Обчислення кількості унікальних методів, які можна викликати.

7. Визначення кількості породжуючих класів. Ідентифікація всіх класів та їхніх суперкласів. Розрахунок кількості суперкласів. Для кожного класу потрібно визначити кількість безпосередніх надкласів. Визначення кількості породжених класів. Ідентифікація всіх класів та їх підкласів. Розрахунок кількості підкласів. Для кожного класу потрібно визначити кількість безпосередніх підкласів. Визначити співвідношення породжуючих і породжених класів. Ідентифікація всіх класів та їхніх батьківських зв’язків. Для кожного класу обчислити відношення кількості підкласів до кількості суперкласів.

На основі розглянутих методів і їх модифікацій було проведено експеримент. Мета експерименту – визначити ефективність застосування різних атрибутів якості коду і дослідження залежності від пріоритету кожного з них. Наприклад, якщо є 10 добре відомих атрибутів, то потрібно визначити, які з них важливі в першу чергу та чи є зв'язок між певними атрибутами.

Експеримент показує, що потрібно уникати складного успадкування. Це також показує, що цикломатична складність та пропуск одного з них не дадуть точних результатів вимірювання якості коду.

Етапи експерименту:

1. Завантаження коду 10-20 популярних проектів з відкритим програмним кодом, написаних на мові C#.

2. Задання списку правил. Цикломатична складність має бути в діапазоні від 20 до 30, де 20 або менше є хорошим рейтингом, а більше 30 – поганим рейтингом. Методи з цикломатичною складністю понад 100–150 необхідно негайно переглянути. Перевищена кількість рядків коду певного класу не повинна перевищувати 800.

3. Написання власного ПЗ, яке може обчислювати показники коду кожного методу/класу на основі розширення файлу (.cs).

4. Програмне забезпечення повинно підтримувати плагін архітектуру, і нові правила можна розширювати. Цей спосіб також дозволяє працювати з набором користувальницьких правил якості коду незалежно.

5. Правила можна записати в текстовому форматі. Наприклад, він може містити діапазон кожного атрибута якості та застосовувати рейтинг програмного забезпечення на основі правила.

Результати експерименту можна представити з такими характеристиками.

1. Назва класу – унікальна назва класу в межах проекта.

2. Назва методу.

3. Номер ітерації – звичайний лічильник. Нарощується кожний раз при новій зміні.

4. Кількість рядків класу. Сюди не відносять коментарі, лише та частина коду яка компілюється.

5. Кількість рядків методу.

6. Сума складності методів у класі.

7. Цикломатична складність методу – ціле число. Чим вище значення тим вища складність класу. Сума значень цикломатичної складності усіх методів класу.

8. Глибина дерева успадкування.

9. Кількість викликів – підраховує кількість унікальних викликів методу та класу.

10. Кількість наслідників класу – кількість класів що наслідуються від поточного класу.

11. Кількість рядків з дубльованим кодом.

12. Технічний борг методу.

13. Частота змін методу.

14. Частота змін класу – як часто метод змінюється.

15. Інформація про анти шаблони.

15.1. Визначення складності написаного методу. Такі методи важко налагоджувати та тестувати а мінімальні зміни в них можуть породжувати не зовсім очевидні баги. Зазвичай це методи з високою цикломатичною складністю.

15.2. Клас виконує занадто багато одночасних функцій [16] і порушує принцип єдиної відповідальності. Зазвичай такі класи важко модифікувати. Також з нього випливають інші проблеми такі як дублікати коду, погана швидкодія.

15.3. Код який не виконується.

15.4. Код, в якому у методу є потенційні проблеми з найменуванням та форматуванням. Зазвичай виловлюється за допомогою статичних аналізаторів коду. В англомовній літературі це зазначається як code smell [17].

Отже, якісний код форматований та зрозумілий, у ньому не має назв змінних які не розкривають її суть. Якісний код містить коментарі. Якісний код не містить дубляжу коду чи логіки. Останнє іноді не зовсім очевидне але якщо відбувається зчитування з файла та видаленого ФТП сервера то дуже часто цю логіку можна звести під один знаменник. Якісний код не містить методів методів з високою Cyclomatic Complexity і таким чином у ньому не має анти-шаблонів God-class чи Спагетті-коду. Якісний код реалізує безпекові принципи, OWASP правил буде достатньо для більшості ПЗ. У випадку якісного коду при ООП підході глибина дерева успадкування та кількість наслідника класу є малими величинами. Якісний код містить автоматизовані тести. Для більшості простих задач буде достатньо почати з простих статичних аналізаторів коду [18, 19]. Отже, найбільш можливою моделлю може бути така, що повертає бали для певного методу:

quality_per_method = (1000 * CC + DIT* 100 + RFC * 20 + NOP * 15) / LOC, (1)

де CC – цикломатична складність

DIT – глибина успадкування

RFC – кількість унікальних викликів іншими методами та класами

NOP – кількість батьківських класів

LOC – кількість рядків методу

Згідно цієї моделі, які задано формулою (1) отримуємо результати для певного методу.

Висновки. Таким чином, отримано оптимальний алгоритм знаходження якості коду згідно з ряду проведених експериментів. Експеримент показав що визначення якості коду є досить складною задачею, оскільки дуже часто прямої залежності між атрибутами якості коду не має. Також, потрібно оптимізувати алгоритми знаходження потенційно складних методів та класів. Особливо слід звернути увагу на алгоритми пошуку анти шаблонів.

Список літератури:

1. Willenbring J., Walia G. (2024). The utility of complexity metrics during code reviews for CSE software projects.

2. Mi Q., Hao Y., Liwei Ou, Ma W. (2021). Towards using visual, semantic and structural features to improve code readability classification.

3. Mashhadi E., Chowdhury S., Modaberi S., Hemmati H. (2024). An empirical study on bug severity estimation using source code metrics and static analysis.

4. Aghamohammadi A., Mirian-Hosseinabadi S. (2021). Statement frequency coverage: A code coverage criterion for assessing test suite effectiveness.

5. Abualkishik A., Lavazza L. (2021). An empirical evaluation of the “Cognitive Complexity” measure as a predictor of code understandability.

6. Ge K., Han Q. (2024). Hidden code vulnerability detection: A study of the Graph-BiLSTM algorithm.

7. Savenko O., Sachenko A., Lysenko S., Markowsky G., & Vasylkiv N. (2020). BOTNET DETECTION APPROACH BASED ON THE DISTRIBUTED SYSTEMS. International Journal of Computing, 19(2), 190-198. https://doi.org/10.47839/ijc.19.2.1761

8. Savenko B., Lysenko S., Bobrovnikova K., Savenko O., Markowsky G. Detection DNS Tunneling Botnets // Proceedings of the 2021 IEEE 11th International Conference on Intelligent Data Acquisition and

Advanced Computing Systems: Technology and Applications (IDAACS), IDAACS'2021, Cracow, Poland, September 22-25, 2021.

9. Nicheporuk A., Savenko O., Nicheporuk A., Nicheporuk Y. An android malware detection method based on CNN mixed-data model, CEUR Workshop Proceedings, Vol. 2732, 2020, pp. 198-213

10. Lysenko S, Bobrovnikova K, Kharchenko V, Savenko O. IoT multi-vector cyberattack detection based on machine learning algorithms: traffic features analysis, experiments, and efficiency. Algorithms. 2022;15(7):239

11. Iftikhar U., Bin Ali N., Börstler J., Usman M. (2022). A tertiary study on links between source code metrics and external quality attributes.

12. Lei M., Li H., Aundhkar H. (2021). Deep learning application on code clone detection: A review of current knowledge.

13. Alfayez R., Winn R., Alwehaibi W., Venson E., Boehm B. (2022). How SonarQube-identified technical debt is prioritized: An exploratory case study.

14. Huang Z., Yu H., Fan G., Shao Z., Zhou Z., Li M. (2023). On the effectiveness of developer features in code smell prioritization: A replication study.

15. Kumar L., Tummalapalli S., Rathi S., Murthy S., Krishna A., Misra S. (2022). Machine learning with word embedding for detecting web-services anti-patterns.

16. Alkharabsheh K., Alawadi S., Ignaim K., Zanoon N., Crespo Y., Manso Y., A. Taboada J. (2022). Prioritization of god class design smell: A multi-criteria based approach.

17. Alazba A., Aljamaan H. (2020). Code smell detection using feature selection and stacking ensemble: An empirical investigation.

18. Van de Laar P., Corvino R., J. Mooij A. (2023). Custom static analysis to enhance insight into the usage of in-house libraries.

19. Papamichail M., Symeonidis A. (2020). A generic methodology for early identification of non-maintainable source code components through analysis of software releases.

20. Lenarduzzi V., Pecorelli F. (2022). A critical comparison on six static analysis tools: Detection, agreement, and precision.

21. Gnoyke P., Schulze S. (2023). Evolution patterns of software-architecture smells: An empirical study of intra- and inter-version smells.

22. Rhmann W., Pandey B., Ansari G., D.K. Pandey. (2020). Software fault prediction based on change metrics using hybrid algorithms: An empirical study.

23. Aldaeej A., Seaman C. (2023). A rule-based decision model to support technical debt decisions: A multiple case study of web and mobile app startups.

24. Kaur N., Singh H. (2020). An empirical assessment of threshold techniques to discriminate the fault status of software.

25. Abbad-Andaloussi A. (2022). On the relationship between source-code metrics and cognitive load: A systematic tertiary review.

26. Cai Y., Kazman R. (2022). Software design analysis and technical debt management based on design rule theory.

27. Sas C., Capiluppi A. (2021). Antipatterns in software classification taxonomies.

28. Yu D., Yang Q., Chen X. (2023). Actionable code smell identification with fusion learning of metrics and semantics.

29. Zakeri-Nasrabadi M., Parsa S. (2022). A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges.

30. AlOmar E., Ivanov A., Kurbatova Z. (2022). Just-in-time code duplicates extraction.

31. Singh M., Chhabra J. (2023). Machine learning based improved cross-project software defect prediction using new structural features in object oriented software.

32. Recupito G., Pecorelli F., Catolino G. (2023). Technical debt in AI-enabled systems: On the prevalence, severity, impact, and management strategies for code and architecture.

33. Huang F., Madeira H. (2023). Advancing modern code review effectiveness through human error mechanisms.

34. Khatri Y. (2022). An effective software cross-project fault prediction model for quality improvement.

35. Madeyski L., Lewowski T. (2022). Detecting code smells using industry-relevant data.

36. Bibiano A., Uchôa A., K.G. Assunção W. (2022). Composite refactoring: Representations, characteristics and effects on software projects.

37. Alcocer J.P.S., Antezana A. S. (2020). Improving the success rate of applying the extract method refactoring.

38. Fahmideh M., Grundy J., Beydoun G., Zowghi D., Susilo W., Mougouei D. (2020). A model-driven approach to reengineering processes in cloud computing.

39. Zakeri-Nasrabadi M., Parsa S., Jafari S. (2023). Measuring and improving software testability at the design level.

40. N. Alkhomsan M., Alshayeb M., Baslyman M. (2024). Toward a novel taxonomy to capture code smells caused by refactoring.

41. Politowski C., Khomh F., Romano S., Scanniello G., Petrillo F., Guéhéneuc Y., Maiga A. (2020). A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension.

42. Alkharabsheh K. (2021). A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of God class.

Prokofiev I.G., Savenko O.S., Medzatiy D.M. METHOD OF STATISTICAL ANALYSIS OF SOFTWARE CODE USING COMPLEX PARAMETERS

Software has permeated all areas of human activity – the economy, science, and business cannot function without software, and millions of people are involved in writing and maintaining it, creating an unprecedented impact on security challenges. Whether it's the military or retail, no industry can do without quality software these days, and even a small failure in a distributed system with a large number of users can cause other subsidiary industries to fail. It can be said that since 1990, the amount of software has increased tremendously, as this period was marked by the wide development of computer technology and the spread of computers and the Internet. During this period, a lot of software was written and released in various fields such as operating systems, business applications, games, web applications, mobile applications and many more. In addition, many software are regularly updated and expanded, so the creation of new versions and support of existing programs continues today. Moreover, software has actually become a common commodity, with individual companies selling API (Application Programming Interface), SaaS (Software as a Service) solutions, and this is no different from conventional resources in the field of energy service or supermarket turnover according to the approach "Pay – Get". Due to the high workload, popular software solutions use new architectural approaches to writing software, which significantly increases the complexity of writing support for such software. And if earlier only technical giants such as Google and Apple really needed unique algorithms to create highly loaded software, then in 2024 the trend of microservice architecture requires solid knowledge of building distributed systems. All this imposes significant limitations on the newly created software – the quality of the software must be a measurable quantity and this value must be at least controlled by both developers and management. Quality software is only possible with quality code, which we'll cover next. Today's software systems can be very complex, so it is very important to maintain existing systems and measure quality. In this article, we will look at various methods of evaluating code quality and security in modern software systems. We'll look at various known code quality metrics and how they can be applied depending on the programming language. The purpose of the article is to review existing software code quality indicators and propose new methods of software code quality analysis. The research work is focused on finding new methods based on machine learning that allow finding different anti-patterns and codesmell in a new way. The result of the method is to identify the overall quality of specific software written in C++ in a programming language similar to C# or Java.

Key words: *software code quality, static code analyser, code quality attributes.*