Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет ________________Комп'ютерних наук________________________
(повна назва)

Кафедра ________________Програмної інженерії_________________________
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

______________другий (магістерський)______________
(рівень вищої освіти)

Дослідження впливу SATD на якість програмного коду
(тема)

Виконав: студент 2 курсу, групи ІПЗм-19-3___
спеціальності 121- Інженерія програмного забезпечення
(код і повна назва спеціальності)
Освітньо-наукової програми
Інженерія програмного забезпечення

___________________Гринько А.М.______________
(прізвище, ініціали)
Керівник _____________проф. Смеляков К.С.__________
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____________________ З.В.Дудар

2021 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121– Інженерія програмного забезпечення

(код і повна назва)

Освітньо-наукова програма Інженерія програмного забезпечення

(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ______________
(підпис)
«_____»________________ 20 ___ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

Студентові Гринько Аліні Миколаївні

(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження впливу SATD на якість програмного коду

затверджена наказом по університету від «26» березня 2021 р № 386

2. Термін подання студентом роботи до екзаменаційної комісії «10» травня 2021 р.

3. Вихідні дані до роботи *Теоретичні відомості про вплив SATD на якість програмного коду, відповідь на питання дослідження, програмна реалізація та доповнений отриманими результатами датасет*

4. Перелік питань, що потрібно опрацювати в роботі *аналіз проблемної галузі і постановка задачі, опис проблем методів дослідження, використовувані методи та алгоритми, опис процессу аналізу, аналіз отриманих результатів та відповідь на дослідницькі питання*

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) *Мета завдання, обґрунтування доцільності розроблення, постановка задачі, демонстраційні матеріали*

6 Консультанти розділів роботи

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Спецчастина			

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка *
1.	Аналіз предметної галузі	3 квітня 2021 р.	виконано
2.	Огляд існуючих методів	5 квітня 2021 р.	виконано
3.	Підготовка програми	7 квітня 2021 р.	виконано
4.	Підготовка пояснювальної записки	17 квітня 2021 р.	виконано
5.	Спецчастина	23 квітня 2021 р.	виконано
6.	Підготовка презентації та доповіді	25 квітня 2021 р.	виконано
7.	Нормоконтроль	4 травня 2021 р.	виконано
8.	Рецензування	7 травня 2021 р.	виконано
9.	Занесення диплома в електронний архів	8 травня 2021 р.	виконано
10.	Передзахист	11 травня 2021 р.	виконано
* заповнюється вручну після виконання чергового пункту			

Дата видачі завдання _____ _____________2021 р.

Студент ____________________________________

(підпис)

Керівник роботи ________________________ __проф. Смеляков К.С002Е___

(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до атестаційної роботи: 73 с., 10 рис., 9 табл., 27 джер.

ТЕХНІЧНИЙ БОРГ (TD), ПІДТВЕРДЖЕННИЙ ТЕХНІЧНИЙ БОРГ (SATD), ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАММНОГО КОДУ

Об’єктом дослідження є зв'язок між SATD та якістю програмного коду.

Метою роботи є дослідження зв’язку між SATD та якістю програмного коду, оціненої SonarQube. Питання дослідження включають: чи існує зв'язок між розміром проекту та відсотком SATD, які типи недоліків є найбільш поширеними в коді, позначеними SATD, чи введення SATD вплинуло на час виправлення помилок?

В результаті не було виявлено зв'язку між розміром проекту та відсотком SATD. Певні типи недоліків пов'язані з SATD. Впровадження SATD має незначний позитивний вплив на час виправлення помилок.

TECHNICAL DEBT (TD), SELF-ADMITTED TECHNICAL DEBT (SATD), ESTIMATION OF SOFTWARE CODE QUALITY

The object of research is a connection between SATD and source code quality.

The purpose of this study is to investigate a connection between SATD and source code quality. The research questions include: is there a connection between the size of the project and the SATD percentage, which types of issues are the most widespread in the code, marked by SATD, did the introduction of SATD influence the bug fixing time?

As a result, no connection between the size of the project and the percentage of SATD was found. There are certain issues that seem to relate to the SATD. The introduction of SATD has a minor positive effect on bug fixing time.

Я, студентка Гринько Аліна Миколаївна гр. ІПЗм-19-3, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему « Дослідження впливу SATD на якість програмного коду », що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

Незважаючи на численні дослідження якості програмного забезпечення, розробники все ще пишуть недосконалий код, який потребує переробки в майбутньому, або спричиняє проблеми. Серед прикладів даного явища - неправильний вибір структури коду, дублікати коду, жорстко закодовані параметри тощо. Це зазвичай робиться для того, щоб пришвидшити процес розробки, укластися в терміни або зменшити витрати [11]. Саме це називається «технічним боргом» (англ. TD). Ця метафора була вперше представлена В. Каннінгемом у 1994 році [2] і використана для інкапсуляції численних проблем якості програмного забезпечення [21]. Це явище я доволі поширеним. Введення TD загалом означає, що розробник знижує якість вихідного коду, роблячи завдання виявлення та виправлення початкової проблеми більш складним. Незважаючи на те, що ці практики очевидно погані [4, 10, 16, 17, 21], технічний борг може бути частково виправданий прискоренням розробки й найшвидшим отриманням результату [11].

Аналогічно «боргу» в економіці, технічний борг може допомогти досягти деяких короткотермінових цілей, але його слід повернути (недосконалий код слід переробити) якомога швидше. Несплата технічної заборгованості може призвести до збільшення витрат у майбутньому. Також може бути набагато складніше завершити рефакторинг на пізнішому етапі, а також знайти непередбачувану помилку у старому коді. Подібним чином існує декілька типів проблем, які ідентифікують потенційні вразливості архітектури, наприклад, «дубльований код», тощо.

Ситуацію, коли розробники чітко усвідомлюють, що вони «беруть технічну заборгованість» і згадують про неї, роскривають TD. Потдар А. та Шихаб Е. у своєму дослідженні [4]. Вони запропонували термін SATD (англ. self-admitted technical debt), який загалом стосується ситуації, коли розробник вводить код із коментарем, наприклад «ToDo: Виправте це пізніше» або залишає примітку в будь-якому іншому каналі зв'язку (наприклад, квитки на Jira [13]). Одним з поширених

методів знаходження SATD у текстових коментарях подібного типу є алгоритми обробки природної мови (NLP) [24].

Отже, метою роботи є дослідження якості вихідного коду у зв'язку з само-підтвердженим технічним боргом (SATD). Це дуже велика тема в розробці програмного забезпечення. Короткий виклад концепції якості програмного забезпечення було дано в [21]. Автори роблять висновок, що якість - це досить складне та контекстне поняття, яке не може мати універсального визначення. Існують також різні погляди на якість програмного забезпечення. Наприклад, погляд користувача на якість програмного забезпечення зосереджений на тому, як програма виконує свою функцію. З точки зору виробництва, якість може бути виміряна критеріями правильного вибору архітектури, витрат на обслуговування тощо. Вимоги до якості можуть бути численними і їх слід визначати в рамках організації або конкретного проекту.

Вплив SATD на якість програмного забезпечення неоднозначний. Дослідження [5] показало, що, незважаючи на низький відсоток SATD, його наявність все одно може негативно вплинути на якість програмного забезпечення. Він також може залишатися в коді протягом тривалого часу: «загалом, час, протягом якого технічно заборгованість, що визнається авторами, залишається в проекті, варіюється в залежності від проекту: медіани коливаються від 18,2 до 172,8 днів; з середнім значенням від 82 до 613,2 днів» [12]. Однак, згідно з [5], «Існує чітка тенденція, яка показує, що після введення SATD спостерігається більший відсоток виправлення дефектів». Ось чому це питання цікаво дослідити.

У кожній ітерації процесу розробки програмного забезпечення, після написання коду що виконує певні функції та не створює помилок, повинні бути задоволені вимоги до якості. Існують різні типи цих вимог, такі як ефективність, надійність, читабельність, здатність до переробки та виправлення, тощо. Існує широкий спектр різних класифікацій, вимірювань та підходів, пов’язаних з якістю коду.

Деякі показники, які, можливо, можуть бути використані для цієї мети, були запропоновані в [22]. Автори виявили певний взаємозв'язок між кількома показниками якості, що свідчить про вимір одних і тих самих властивостей.

Ми зосередимося на більш широкій класифікації, що використовується такими засобами статичного аналізу коду, як SonarQube. Причина використання SonarQube пов'язана з високою популярністю та широким спектром застосувань.

Більш конкретно, поточний проект спрямований на вивчення зв'язку між технічно визнаним боргом [4] та проблемами з кодом, виявленими SonarQube.

1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

Технічний борг (або TD) описує ситуацію, коли розробник не вирішує проблеми негайно, а відкладає їх на майбутнє. Отже, він або вона метафорично «бере борг». Це не недавня метафора. Вона була запроваджена в 1994 р. [2], і є багато досліджень пов’язаних з TD. Його негативний ефект очевидний і був описаний у різних дослідженнях [4, 10, 16, 17, 21]. Будь-яку заборгованість слід повернути, інакше збір за виплату стане занадто високим. Однак є й інша сторона цього питання.

Деякі дослідники стверджують, що технічного боргу не уникнути [11]. Іноді кожна секунда затримки процесу доставки товару на ринок може бути критичною для бізнесу. З цієї точки зору має існувати певний баланс між діловими та технічними цілями. Тому менеджери іноді можуть протистояти технічним спеціалістам щодо вимог до якості коду, важливості рефакторингу та подібних питань. Іноді з’являються такі принципи, як «якщо щось не зламано, не виправляйте це». Це робить ситуацію з технічним боргом доволі сумнівною.

Загалом, ніхто не може стверджувати, що ефективне спілкування між членами команди не є важливим. Навіть якщо технічна заборгованість з'являється через деякі вимоги бізнесу, вона повинна бути видимою, і члени команди повинні про неї знати. Слід створювати спеціальні дискусії, інформаційні дошки та віки. Існує декілька способів інформувати членів команди про TD. Загалом, це стосується ситуацій, коли розробник чітко розуміє, що він бере технічну заборгованість, і повідомляє про це за допомогою каналів зв'язку. Ми частіше розглядаємо SATD у коментарях вихідного коду [4, 6, 7, 8], але важливо розуміти, що це не єдиний підхід. Наприклад, проблеми можуть відзначатися в такій системі відстеження, як Jira [13], або просто у файлах із назвою «ToDo», та ін. Тут ми

будемо обговорювати SATD, який міститься в коментарях до коду. Отже, які там коментарі?

Таблиця 1.1- Приклади коментарів вихідного коду, маркованих як SATD

№	Оригінальний коментар (цитовано англійською)
1	// TODO: Do I need this? Hmmm, maybe I do.
2	«// The token is pointless for kerberos // TODO verify all columns»

3	«// Should be about a 3 second scan
	// Try to find the active scan for about 15seconds
	// TODO: any way to tell if the client address is accurate? could be local IP, host, loopback...?
	// Scan ID should be a long (throwing an exception if it fails to parse)»

4	«// combine all histories by target
	// !!! FIXME: temporary until velocity templates are implemented
	// !!! hmmmmmm
	// set dispatch credentials
	// set all other dispatch properties»

5	«// TODO: log this»
6	«//@todo we should parse the value in case its an Expression»
7	«// HACK.. Why??»
8	«/** * Creates a file system manager instance. * @todo Load manager config from a file.»
9	«// @@@FIXME: check for other dsig structures»

10	«// don't re-establish connection if we are closing
	// If we are in read-only mode, seek for read/write server
	// closing so this is expected
	// this is ugly, you have a better way speak up»

У таблиці вище є кілька прикладів. Ці коментарі були вибрані випадковим чином з усіх коментарів, зібраних в поточному дослідження. Як ми бачимо, будь-який коментар тут визначається в межах методу, тому, якщо є колекція коментарів, кілька однорядкових коментарів тощо, вони представляються як один.

Як ми також можемо побачити тут, більшість із цих коментарів містять ключове слово «todo». Використання цього слова в коментарях є традиційним способом інформування про SATD. Ці коментарі часто виділяються середовищем розробки, системою контролю версій тощо. Існують також правила SonarQube, які використовується для обробки цих коментарів.

Ще одним чудовим коментарем є номер вісім у таблиці 1.1, оскільки він показує, як SATD може бути представлений у коментарях JavaDoc. Також те, як виглядає SATD, сильно залежить від мовних звичок автора. Наприклад, третій рядок у коментарі номер чотири «!!! Hmmmm», безумовно, означає, що автор не впевнений у наступних рядках, однак, це не граматично-правильне слово, і воно не має жодного значення. Його також неможливо знайти за допомогою будь-яких методів виявлення на основі ключових слів.

Як бачимо, SATD може бути різних форм і типів, що мають особливо специфічні властивості. Тоді як ми визначаємо SATD? Існують різні методології, використовувані в попередніх дослідженнях. Це обговорюється в розділі «Техніка виявлення SATD». Загалом, ми маємо зауважити, що це завдання не є простим, оскільки ми маємо справу з проблемою обробки природної мови. І ця мова не завжди є офіційною та правильною. Методи для обробки природних мов представленні у дослідженні [24]. Який код зазвичай коментують розробники? Це питання, на яке ми збираємося відповісти нижче.

Існують різні способи пошуку та розпізнавання технічної заборгованості. Це не тривіальне завдання. Зазворка та ін. [10] описують приклад, коли розробників попросили передивитися вручну вихідний код і спробувати знайти TD. Це ж завдання було виконано паралельно із спеціальним програмним забезпеченням. Як

результат, визначений людиною TD і автоматично визначений (для цього випадку з використанням інструменту Findbug) був не однаковим, але може перекриватися, особливо у деяких випадках. Тому виявлення технічної заборгованості не є простою задачею.

Експерти, які перевіряють базу коду вручну, дадуть найбільш точні результати виявлення, але це дуже дорого та працеємно. Автори «Технічного набору даних про борг» [3] пропонують використовувати для цієї задачі метрики SonarQube.

Різні пізніші дослідження [5, 6] мали на меті дослідити причини та особливості технічного боргу, однак вони проводились із використанням різних наборів даних, тому результати важко дослідити та порівняти. Тому використання загального набору даних може допомогти порівняти отримані результати з будь-якими потенційними майбутніми роботами.

1.1 Постановка задачі

Стверджується, що SATD є чимось неминучим і навіть корисним на деяких стадіях розвитку, особливо з точки зору менеджерів [11]. Він також широко поширений.

Однак у інших роботах [5, 4] визнано негативний вплив SATD. Він також може тривалий час залишатися в коді [12]. Справжні причини та вплив SATD видаються сумнівними, отже цікавими для дослідження.

Цей проект спрямований на пошук зв'язку між SATD та проблемами, виявленими в проектах за допомогою засобів автоматизованого аналізу. Для цього буде використано SonarQube.

Питання дослідження, на які дадуть відповіді, наведені в таблиці нижче:

Таблиця 1.2 – Дослідницькі питання

Позначення	Дослідницьке питання
RQ1	Чи є зв’язок між розміром проекту та відсотком SATD?
RQ2	Які види недоліків програмного коду є найбільш розповсюдженими у ділянках коду з технічною заборгованістю?
RQ3	Як впливає введення SATD на час виправлення недоліків?

Зв'язок між розміром проекту та відсотком SATD вже досліджувався в [4] на невеликому наборі з 3 проектів та в [6] на більшому наборі з 159 проектів. Однак спосіб визначення SATD був іншим. В обох попередніх роботах була використана методологія, що визначає SATD, на основі 62 ключових слів та словосполучень [4]. У [6] «відсоток SATD» означає відсоток коментарів із SATD від усіх коментарів, тоді як у поточній роботі та в [4] це означає відсоток файлів, що містять SATD від усіх файлів. Враховуючи це, ми взяли метод із [4] як базовий і розширили його за рахунок більшого обсягу проекту та додаткової методології визначення SATD [9].

Цікаво дослідити також типи питань, які є найбільш поширеними в коді, позначеному SATD. Раніше декілька типів недоліків розглядались стосовно TD [16, 17], тоді як SATD не розглядався. Крім того, як виявлено в [4], досвідченіші розробники, як правило, впроваджують більше SATD, ніж менш досвідчені. Питання про те, які проблеми частіше позначаються коментарями SATD, залишається відкритим. Зв'язок між часом виправлення проблем та введенням SATD досліджено в [12]. Однак немає чіткого порівняння фактичного часу, необхідного для виправлення проблеми. Інша робота стверджує, що «існує чітка тенденція, яка показує, що після введення SATD спостерігається більший відсоток виправлення дефектів» [5]. Однак фактичний час не вимірювався. Це спонукало нас провести конкретне порівняння часу, необхідного для виправлення проблем, пов’язаних із SATD.

1.2 Кодова база та цільова група

В якості кодової бази було використано 30 проектів Apache з відкритим кодом. Ці проекти включені до набору даних [3], згаданого вище. Мовою програмування аналізованих файлів є Java.

Цільовою групою цього проекту є професійні розробники, які вважають якість коду одним із своїх головних інтересів. Він також може включати дослідників, які досліджують SATD, оскільки набір даних, що містить результати цього проекту, може бути корисним для подальших дослідницьких цілей.

1.3 Методи дослідження

Відповідно до поставлених дослідницьких питань, основною метою даної роботи є дослідити, чи існує зв'язок між проблемами якості програмного забезпечення та SATD, і якщо він існує, який вплив може мати SATD.

Пов'язані роботи були проаналізовані з метою вивчення галузі дослідження, формулювання дослідницьких питань, вибору найкращої стратегії виявлення SATD та вивчення результатів подібних робіт.

З метою відповіді на дослідницькі питання було проведено ретроспективне тематичне дослідження [19].

Дані, зібрані в наборі даних [3], представляють неодноразові спостереження за характеристиками якості коду, зібрані засобом статичного аналізу коду (SonarQube) з усієї історії проектів.

Ми також проаналізували дані з системи контролю версій (Git) для збору даних, пов'язаних із коментарями SATD. На основі цих даних ми можемо встановити зв'язок між SATD та проблемами вихідного коду. Після встановлення зв’язку дані можна розділити на дві групи, виходячи з критеріїв, пов’язана проблема з SATD чи ні.

На основі дослідницьких питань ми формулюємо гіпотези і, навпаки, нульові гіпотези.

H1 (RQ1): Існує значний зв’язок між розміром проекту та відсотком SATD.

Як уже зазначалося, вибір між методологіями виявлення SATD базувався на надійності та складності впровадження. Два методи було вибрано за цими критеріями. Перший - основна методологія, описана в [4]. Щоб застосувати його, ми порівняли текст кожного коментаря з 62 ключовими словами, знайденими попередніми дослідниками [4], коли вони вручну перевірили 101 762 коментарі у вихідному коді. Якщо ключове слово існує в тексті коментаря, тоді цей коментар

вважається представленням SATD. Прикладом тут може бути «ToDo», «FixMe» та інші подібні ключові слова.

Другий метод заснований на використанні готового рішення для аналізу тексту, наданому попереднім дослідженням [9]. Це готова до використання бібліотека Java, яка містить попередньо навчену нейронну сітку для аналізу тексту. Дана модель містить чотири етапи: попередня обробка тексту, функція відбору, навчання підкласифікаторів та голосування класифікаторів. Як набір даних моделі було використано 212 413 коментарів, наданих Мальдонадо та Шихабом [22]. Цей метод є новішим та надійнішим [9].

Було вирішено використовувати комбінацію основного методу, використовуваного в попередніх дослідженнях [4], та більш сучасного та ефективного [9] як методології, що визначає SATD.

Щоб визначити, чи залежать результати від методу виявлення SATD, ми використали дві групи даних: «SATD, визначені обома методами» та «SATD, визначені принаймні одним методом». Для того, щоб знайти статистичний зв’язок між двома змінними, розміром проекту та відсотком SATD, буде використаний кореляційний тест Пірсона [20].

Щоб відповісти на RQ2, нам потрібно буде порівняти проблеми, пов'язані з SATD, та всі проблеми загалом. Критеріями, необхідними для визначення того, що коментар SATD пов'язаний з проблемою SonarQube, буде те, що вони повинні бути присутніми в тому самому блоці коду одночасно. Очікувалося, що проблеми, пов'язані з SATD, будуть іншого типу порівняно з не пов'язаними з SATD.

Для аналізу даних буде використана описова статистика. Дані були згруповані за типом випуску та його частотою. Для перевірки різниці між двома розподілами буде використано тест хі-квадрат. Він був обраний, оскільки його можна використовувати для доведення незалежності категоріальних змінних.

H1 (RQ3): Існує суттєва різниця між часом існування проблем, пов’язаних із SATD, та проблемами, не пов’язаними з SATD.

Автори набору даних [3] вимірювали часові проміжки між релізами, виконуючи алгоритм SZZ. Алгоритм заснований на зв'язуванні системи контролю версій, напр. Git, до системи відстеження проблем (Jira, Bugzilla). Реалізація, що використовується тут [23], називається OpenSZZ, і вона приймає URL-адресу проекту Git та URL-адресу проекту Jira як вхідні дані та повертає список комітів, що викликають несправності та виправляють несправності, як вихідні дані. Він отримує коміти, вказані в Jira, та визначає, яка частина коду була змінена в цьому коміті. Потім він виконує певну оцінку за допомогою семантичного та синтаксичного аналізу цих комітів та фільтрує їх. Спочатку вибираються та оцінюються коміти з усунення несправностей. Потім обираються коміти, що викликають несправності, які можуть бути пов’язані з тим самим компонентом та проблемою в Jira. Більш детальне пояснення та оцінка дано в [23].

Що стосується статистичного аналізу, то групи даних будуть представлені на графіках та буде проведено тест ANOVA [20]. Його було обрано, оскільки тест ANOVA - це класичний спосіб вказати, чи існує суттєва різниця між групами даних.

2. ОГЛЯД ІСНУЮЧИХ МЕТОДІВ ВИЯВЛЕННЯ SATD

2.1 Визначення та вплив SATD

SATD та проблеми, пов'язані з ним, викликають кілька питань. Чи є ці проблеми більш серйозними порівняно з проблемами, не пов'язаними з SATD? Чи є у розробників, які їх впроваджують, щось спільне? Пордар і Шихаб виявили, що «розробники з вищим досвідом мають тенденцію вводити більшу частину SATD», і що «тимчасовий тиск і складність коду не корелюють із сумою SATD» [4]. Вони також визнають, що не існує прямого взаємозв'язку між цикломатичною складністю та SATD [4] і що «у деяких проектах файли SATD мають більше змін у виправленні помилок, тоді як в інших проектах, SATD файли мають більший відсоток дефектів» [5]. У «Широкомасштабному емпіричному дослідженні SATD» [6] автори не виявили зв'язку між зв’язністю, складністю, зручністю читання та SATD.

Отже, це піднімає питання, чи вводять найдосвідченіші розробники більше SATD, оскільки вони бачать більше можливих удосконалень коду, тоді як менш досвідчені, як правило, ігнорують це. У цьому випадку SATD може бути менш значним, ніж TD.

У цих дослідженнях визначення «досвідчений розробник» пов’язане із загальною кількістю комітів, виконаних розробниками до виправлення SATD [4], або кількістю комітів, виконаних у поточному файлі до фіксації SATD [6], а не фактичний досвід розробників.

Дослідження [5] показало, що, незважаючи на низький відсоток SATD, він все одно має негативний вплив на якість програмного забезпечення. Він також може залишатися в коді протягом тривалого часу: «Загалом, час, протягом якого технічно заборгованість, що визнається авторами, залишається в проекті,

варіюється в залежності від проекту: медіани становлять від 18,2 до 172,8 днів і в середньому складають від 82 до 613,2 днів» [12]. Отже, виникає наступне запитання: чи існує зв'язок між розміром або тривалістю проекту та сумою внесеного SATD?

У своїх роботах 2016 року [6] Бавота та Руссо повідомили про високу дифузію SATD в екосистемних проектах Apache. Вони визнали, що сума SATD «збільшується з часом за рахунок введення нових екземплярів, які не фіксуються розробниками» [6].

Але чи є цей вплив за своєю суттю негативним? Згідно з [5], «існує чітка тенденція, яка показує, що після введення SATD спостерігається більший відсоток виправлення дефектів».

2.2 Техніка виявлення SATD

Завдання пошуку SATD в коді є окремим і досить складним. Існують різні методології, що використовуються для визначення SATD. Методологія, виявлена в дослідженні Потдара та Шихаба [4], базується на 62 текстових шаблонах і є, мабуть, найбільш традиційною. Пізніше були введені такі методи, як CVM-TD [14]. Він базується на поєднанні ключових слів, частин мови та тегів. Він був ефективнішим [15], ніж попередні, і показав хороші результати на думку респондентів, які брали участь в інтерв'ю описаному в [15].

Існують також методології, засновані на обробці природних мов [7], видобутку тексту [9], n-грам IDF [8] та ін. Останні дві є нещодавно запровадженими та досить цікавими і засновані на підходах машинного навчання. Автори [9] надають корисний інструмент у вигляді бібліотеки JAR, який ми використали у поєднанні його з найбільш базовим підходом, описаним у [4].

Крім того, не тільки коментарі вихідного коду можуть бути проаналізовані для виявлення SATD. Для цього можуть використовуватися системи, такі як Jira [13]. Будь-які проблеми, створені там, можуть бути позначені як пов'язані з TD. Цей підхід, безумовно, цікавий, проте він виходить за межі даного дослідження.

2.3 Інструменти для виявлення TD

Існують різні інструменти для аналізу TD. За даними опитуваних респондентів [21], найпоширенішими інструментами, що використовуються для цієї мети, є інструменти контролю помилок, такі як Redmine, Jira та Team Foundation Server. Також слід згадати аналіз залежностей (наприклад, SonarQube, Understand), перевірку правил коду (наприклад CPPCheck, Findbugs, SonarQube) та інструменти метрики коду (наприклад Sloccount) [21]. 50% респондентів заявили, що взагалі не використовують будь-які інструменти.

Зазворка та ін. [10] використовувли Findbugs у своєму дослідженні. Це програма статичного аналізу, яка використовує байт-код, тому програмне забезпечення потрібно компілювати. Серед результатів даного дослідження - технічний борг виявлений вручну та технічний борг виявлений автоматично виявилися не однаковими. Однак Findbugs сьогодні не є найпоширенішим інструментом для цієї мети. Він також вимагає компіляції коду, що не завжди можливо.

Як інструмент автоматизованої перевірки коду, SonarQube набув величезної популярності і в даний час використовується понад 120 000 користувачами [1]. Він реалізований для 27 мов програмування та інтегрований з найпопулярнішими інструментами CI/CD [1]. Він також інтегрується з іншими інструментами аналізу, такими як Findbugs.

Після встановлення SonarQube можна запускати в команді менеджера пакетів (наприклад, Maven або Gradle), що забезпечує повний аналіз вихідного коду в проекті. Він визначає проблеми, їх місцезнаходження, ступінь тяжкості, тип, технічну заборгованість тощо [1].

Під «проблемами» ми маємо на увазі фрагменти коду, які не відповідають заздалегідь визначеним вимогам або не відповідають певним правилам [18].

Існує три типи проблем, а саме «issues», «vulnerabilities» та «code smells».

«Помилки» стосуються коду, який, можливо, вже зламаний або не відповідає вимогам надійності. Ці проблеми є найбільш серйозними і потребують якнайшвидшого вирішення. Прикладами тут можуть бути нескінченний цикл, неправильна кількість аргументів у методі тощо. «Вразливості» визначають потенційні ризики для безпеки. Це означає, що цей код може вказувати на слабке місце в системі і може використовуватися людиною з наміром заподіяти їй шкоду. Найпоширеніші приклади тут - порушення рівнів модифікаторів доступу. Нарешті, є проблеми із «code smells». Це різні проблеми, які не є помилками «bugs» чи вразливими місцями «vulnerabilities». Вони можуть бути відносно нешкідливими, але в інших випадках можуть вказувати на потенційні архітектурні помилки. Щодо них існує безліч прикладів, серед яких дублювання коду, довгі методи, висока когнітивна складність методів тощо.

Проблеми характеризуються також своєю гостротою. Існує п’ять типів тяжкості: «BLOCKER», «CRITICAL», «MAJOR», «MINOR», «INFO». Перші два представляють негативний вплив, який вони мають на систему. «BLOCKER» стосується проблеми, яка, швидше за все, дасть негативний ефект і зробить код менш надійним. Ці проблеми слід негайно вирішити. Проблеми з «CRITICAL» серйозністю також повинні бути виправлені, але, порівняно з «BLOCKER», вони не такі актуальні. «MAJOR» та «MINOR» відображають вплив на продуктивність розробника. Рівень «MAJOR» означає, що програмне забезпечення сильно впливає на продуктивність розробника, а «MINOR» порівняно нижчий. «INFO» - це лише

інформаційні проблеми, які слід виправити, котрі мають низький рівень серйозності.

Отже, для кожної проблеми SonarQube збирає таку інформацію, як її тип та важкість, правила та повідомлення з поясненнями, рядки коду тощо. Він також оцінює, скільки часу потрібно для виправлення цієї проблеми. Це називається «зусиллями» та «технічним боргом» випуску. Тут «технічний борг» має чисельне значення та відрізняється від визначеного вище, тож має обговорюватися окремо.

Одним з найцікавіших джерел тут є «Еволюція технічного боргу в екосистемі Apache» [16]. Автори перевірили розвиток 66 проектів Apache. Вони також використовували SonarQube як головний інструмент дослідження. Вони досліджували, як технічний борг у цих системах розвивається з часом. Відповідь на це запитання: «у більшості систем, які ми вивчали, спостерігається суттєва тенденція до збільшення кількості проблем з ростом складності проекту. З іншого боку, нормалізований технічний борг зменшується в міру розвитку проекту» [16]. Більше того, досліджувались найчастіші види технічних боргів. Дослідники стверджують, що «найдорожчі види технічного боргу, які необхідно повернути в екосистемі, насправді є проблемами вищого рівня з дублюванням коду та обробкою винятків» [16]. Ці висновки важливі, оскільки вони пов’язані з даною роботою.

Ще одна цікава робота - «Як розробники виправляють проблеми та повертають технічний борг в екосистемі Apache?» [17]. Це пов’язано з еволюцією технічного боргу. Автори також використовували SonarQube для виявлення боргу. Для цього вони відібрали 57 проектів на основі Java з екосистеми Apache. Вони не знайшли зв'язку між рівнем виправлення проблем (відсоток виправлених проблем) та розміром проекту. Виявлено три класи питань, які становлять більшу частину технічного боргу: складність методу, дублювання коду, обробка винятків.

Щодо часу виправлення проблем, це дослідження стверджує, що «майже 20% (≈30 тис. / 155 тис.) питань виправляються протягом одного місяця з моменту їх введення» і «понад 50% питань вирішуються протягом першого року» [17].

3. ОПИС ПРОВЕДЕНИХ ДОСЛІДЖЕНЬ

Спочатку більшість інформації передбачалось знаходити в основному наборі даних. Однак багато даних там бракувало. Автори набору даних TD надають таблиці GIT_COMMITS_CHANGES інформацію про хеш коміту, ім'я файлу, тип зміни (додавання, видалення, змінення). У ньому 891 711 записів. Однак під час ручного огляду було виявлено деякі відсутні в цій таблиці коміти. Після реалізації кроку 1 була заповнена таблиця GIT_CHANGES_PARSED, яка містить загалом 3 830 007 записів. Крім того, неможливість порівняти рядки в коді різних станів комітів спричинила потребу в додатковому аналізі SonarQube (крок 5).

Підводячи підсумок, для відповіді на питання дослідження були потрібні такі дані:

- Git-коміти та назви файлів, змінені в цих комітах.
- Коментарі SATD, знайдені у файлах, їх текст, час додавання та видалення, ім'я файлу, якому вони належать, та рядки методів, що коментуються.
- Проблеми, які були знайдені відповідними методами.

Основний клас парсера написаний на Java 8. Зважаючи на специфіку кожного проекту, для їх запуску ми використовували дві різні операційні системи: Windows 10 (v. 1809) та Ubuntu 18.04 LTS, обидві встановлені на одній машині. Архітектура систем та взаємодія між компонентами однакова для обох середовищ, однак синтаксис сценарію дещо відрізняється. Ми використовували скрипт sh для Ubuntu та скрипт bat для Windows.

Для побудови проекту Apache Beam використовувалася ОС Ubuntu, оскільки система Windows мала проблеми з запуском збірки Gradle.

Технічні характеристики комп'ютера були наступними: Процесор Intel Core i7-8550U 1,80 ГГц та 16,0 ГБ встановленої оперативної пам'яті DDR3.

Впроваджене програмне забезпечення побудовано на базі Maven (v. 3.6.1). Усі залежності перелічені у pom.xml.

Було проведено чотири етапи аналізу та кожен запуск за допомогою окремого компонента Java. Результати кожного кроку відображаються в базі даних SQLite (v. 3.0). Його головна перевага полягає в тому, наскільки він портативний. Він також підтримує всю необхідну функціональність SQL. Цей самий тип бази даних використовувався для збереження початкового набору даних. Для доступу до нього використовувався стандартний JDBS.

Перший крок аналізу (рис. 3.1) був мотивований недостатністю таблиці GIT_CHANGES у наборі даних. Як уже було сказано, у ньому є 891 711 записів, тоді як після цього кроку було зібрано загалом 3 830 007 записів. Для синтаксичного аналізу інформації про сховище використовувався jGit (v. 5.6). Він надає можливість переглядати всі коміти з усіх гілок та аналізувати інформацію про зміни (наприклад, які файли брали участь у зміні коміту, тип змін тощо). В результаті ми отримали інформацію про коміт (його хеш і позначку часу), імена змінених файлів, тип змін (додавання, видалення, змінення). Вся ця інформація була збережена у таблиці GIT_CHANGES_PARSED.

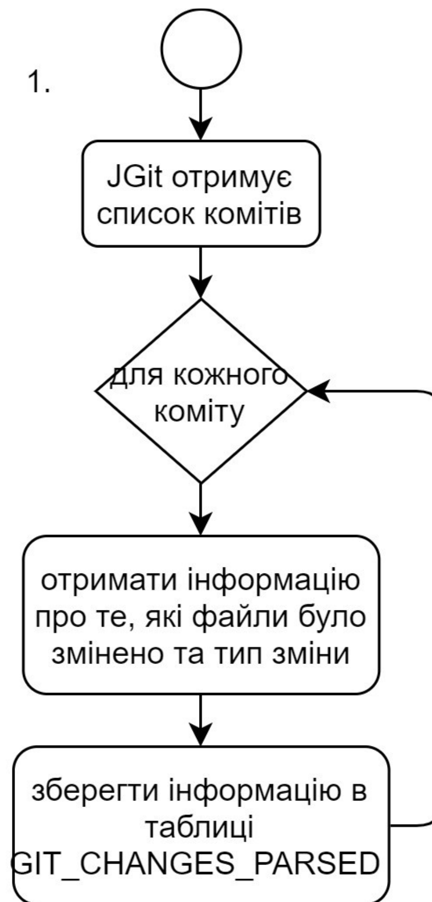


Рисунок 3.1 - Потік кроку 1

Як другий крок аналізу, вся інформація, зібрана в таблиці GIT_CHANGES_PARSED, повторюється за допомогою jGit, щоб знайти вміст файлу в кожному коміті та передати цю інформацію JavaParser (v 3.13.3), який аналізує введений файл Java та автоматично створює абстрактне дерево синтаксису. Потім він реалізує шаблон Visitor і проходить усі вузли коду (наприклад, клас, метод, цикл, коментар тощо). Цей підхід був використаний для збору повної інформації про коментарі. Це було необхідно, оскільки деякі багаторядкові коментарі можна подати як колекцію однорядкових коментарів, але до них не слід підходити як до кількох коментарів. Крім того, кілька коментарів у межах одного методу повинні бути представлені як один коментар. Потім інформація про кожен коментар (його текст, ім'я файлу, якому він належить, рядки

коментованого методу тощо) зберігається в таблиці COMMENTS_ALL. Такий крок мотивований необхідністю застосування різних технік виявлення SATD, тому всі коментарі (як SATD, так і не пов'язані з SATD) зберігаються в базі даних.

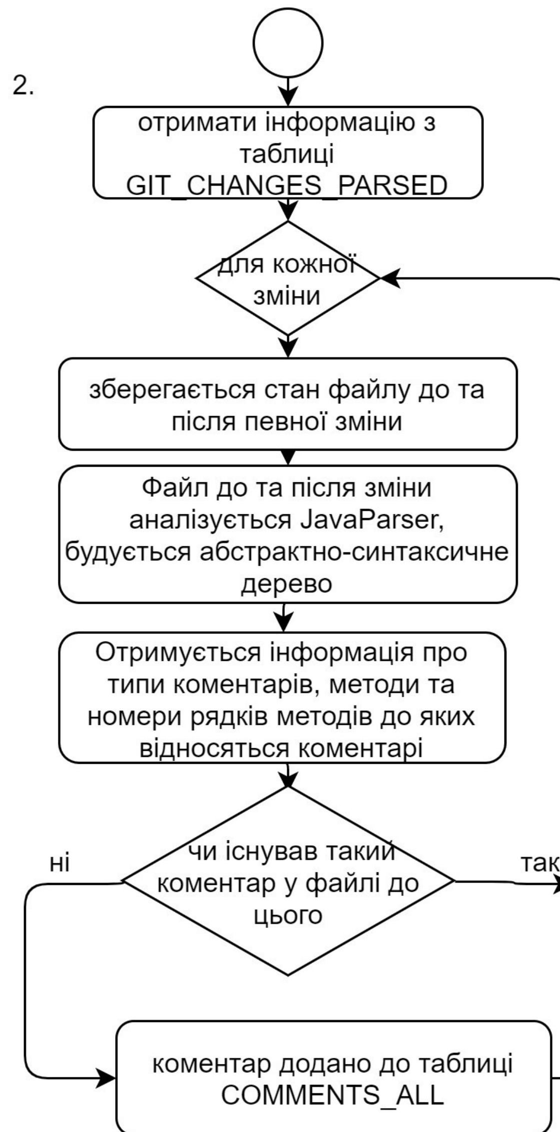


Рисунок 3.2 - Потік кроку 2

Третій крок (див. рис. 3.3) був мотивований дуже великою кількістю коментарів у таблиці COMMENTS_ALL та тим, що там було знайдено багато дубльованих коментарів. Віна мала величезну кількість записів (6 392 996). Ці записи перевірялись на наявність унікальної комбінації тексту коментаря та імені

файлу. Як результат, таблиця COMMENTS_DISTINCT була заповнена. Кількість коментарів там є більш реалістичною, загалом відображається 282 929 записів

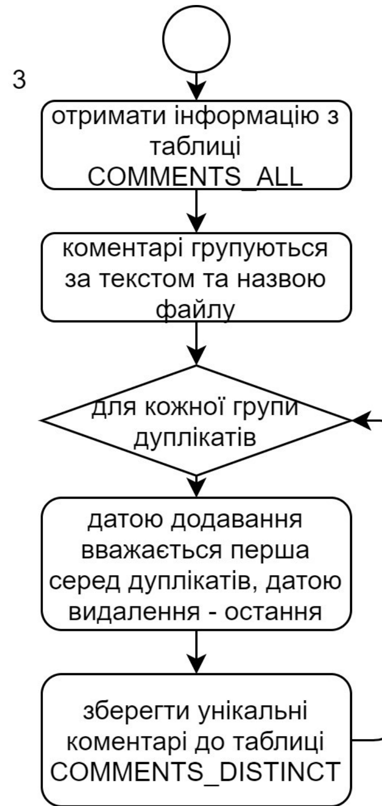


Рисунок 3.3 - Потік кроку 3

Щоб уточнити, представляє коментар SATD чи ні, було введено четвертий крок (див. рис. 3.4). Для цього використовували два методи виявлення SATD. Метод обробки тексту є більш сучасним та ефективним [9]. Щоб застосувати його, ми підключили файл JAR, наданий авторами дослідження. Метод ключових слів [4] є базовим і включає 62 шаблони тексту, зібрані авторами вручну. Після ітерації всіх коментарів таблиця COMMENTS_DISTINCT була оновлена з інформацією про те, чи представляє коментар SATD і які методи його там виявили. Щоб звузити фокус і зменшити час виконання, SATD був обмежений підмножиною цих двох методів.

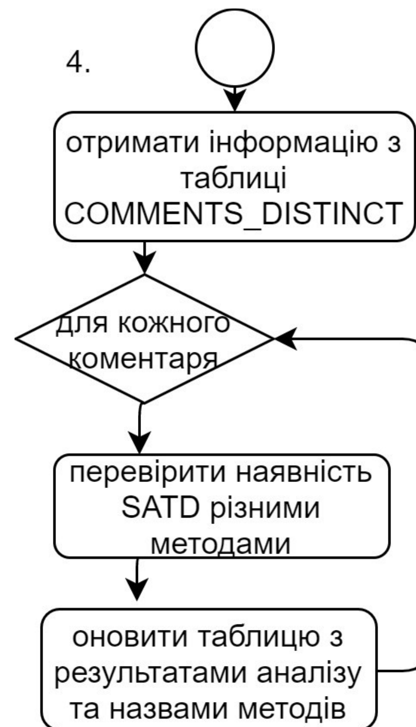


Рисунок 3.4 - Потік кроку 4

Спочатку планувалося використовувати для дослідження проблеми вихідного коду, знайдені авторами набору даних [3] за допомогою SonarQube. Однак кількість проблем, які можна було використовувати для порівняння, була досить низькою через різні стани комітів, тому виникла необхідність аналізу коду з різних комітів. Для цього була побудована наступна система (див. рис. 3.5). Він складається з основного класу синтаксичного аналізатора, який повторює всі коміти та передає їх як аргумент у скрипт sh або bat. Потім скрипт відновлює робоче дерево конкретного коміту, компілює код за допомогою Maven або Gradle і запускає аналіз SonarQube.

Ми встановили версію спільноти SonarQube v8.2.0.32929. Налаштування було відредаговано для встановлення максимального обсягу обчислювального механізму 2048 МБ, що набагато більше, ніж значення за замовчуванням. Аналізатор SonarQube можна запускати з найпоширенішими засобами

автоматизації збірки, а саме Maven або Gradle. Тому всі досліджені нами проекти налаштовані таким чином, щоб використовувати Maven або Gradle.

Після запуску SonarQube запускає веб-хук до вказаного порту. Для прослуховування цього порту був реалізований невеликий сервіс на SpringBoot (v. 2.2). Технологію обрано, оскільки вона зручна у використанні та швидко реалізується. Далі служба надсилає запит до SonarQube API і отримує відповідь у форматі JSON із виявленими проблемами. З метою аналізу відповіді була використана бібліотека Json-Simple (v. 1.1.1). Отримані проблеми були збережені в таблиці SONAR_ISSUES_PARSED.

Ми паралельно будували проекти в різних каталогах, однак безкоштовна версія SonarQube обмежена лише одним потоком обчислень. Аналіз одного коміту зайняв у середньому SonarQube 10–12 хвилин. Отже, нам довелося обмежити кількість комітів, проаналізованих SonarQube.

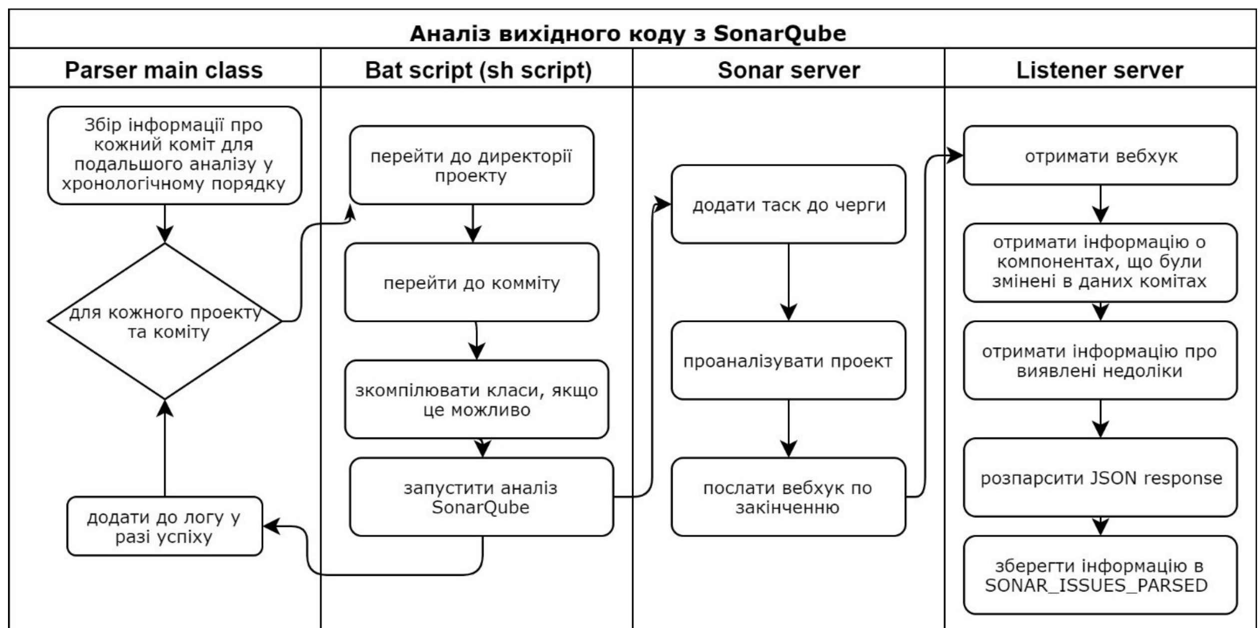


Рисунок 3.5 - Крок аналізу SonarQube

Після того, як були зібрані всі дані, був проведений певний статистичний аналіз. Це буде описано далі в розділі аналізу.

Зв'язок між розміром проекту та відсотком SATD вже досліджувався в [4] на невеликому наборі з 3 проектів та в [6] на більшому наборі з 159 проектів. Однак спосіб визначення SATD був іншим. В обох попередніх роботах була використана методологія, що визначає SATD, із 62 шаблонами тексту [4]. У [6] «відсоток SATD» означає відсоток коментарів із SATD від усіх коментарів, тоді як у поточній роботі та в [4] це означає відсоток файлів, що містять SATD від усіх файлів. Враховуючи це, ми взяли метод із [4] як базовий і розширили його за рахунок більшого обсягу проекту та додаткової методології визначення SATD [9].

Цікаво дослідити також типи проблем, які є найбільш поширеними в коді, позначеному SATD. Крім того, як виявлено в [4], досвідченіші розробники, як правило, впроваджують більше SATD, ніж менш досвідчені. Значення того, які проблеми частіше відзначаються коментарями SATD, залишається відкритим питанням.

Зв'язок між часом виправлення проблем та введенням SATD досліджено в [12]. Однак немає чіткого порівняння фактичного часу, необхідного для виправлення проблеми. Інша робота стверджує, що «існує чітка тенденція, яка показує, що після введення SATD спостерігається більший відсоток виправлення дефектів» [5]. Однак фактичний час не вимірювався. Це спонукало нас провести конкретне порівняння часу, необхідного для виправлення проблем, пов’язаних із SATD.

4. АНАЛІЗ РЕЗУЛЬТАТІВ ДОСЛІДЖЕНЬ

Аналіз даних проводили різними видами статистичних методів, і для цього використовували мову програмування R. В якості вхідних даних були взяті файли .csv із отриманими даними.

4.1 Аналіз зв'язку між розміром проекту та відсотком SATD

Для відповіді на RQ1 був розрахований відсоток файлів, що містять SATD. Повні таблиці з результатами аналізу наведені в додатку А. Кількість проектів становить 30. SATD на рівні деталізації на рівні файлів коливається від 0% – 20,83% (середнє значення - 8,8, стандартне відхилення - 4,87), якщо ми розглядаємо SATD, визначене принаймні одним методом, і від 0% –18,06% (середнє значення - 5,9, стандартне відхилення - 4,17), якщо SATD визначали обома методами. Нижче наведена таблиця 4.1 5 найкращих проектів SATD в порядку їх відсотка від SATD:

Таблиця 4.1 - 5 найкращих проектів за відсотком SATD та їх розміром

Назва проекту	Кількість файлів
mina-sshd	2277
commons-bcel	1357
commons-dbcp	433
commons-codec	348
commons-exec	72

Рисунок 4.1 - 5 найкращих проектів SATD із відсотком SATD, визначеним одним методом, та відсотком SATD, визначеним обома методами

За допомогою тесту кореляції Пірсона було перевірено, чи існує кореляція між розміром проекту та відсотком SATD. Результати наведені в додатку А. Значення Р занадто високі, кореляція не виявлена.

У [4] було виявлено 2,4% -31% SATD щодо рівня деталізації файлів. Результат базувався на трьох проектах. Оскільки ми маємо справу з відсотками, заснованими на 30 проектах, ширший діапазон близько 0% –20,83% виглядає правдоподібним.

Відсоток файлів із SATD відрізняється від проекту до проекту. Два фактори, які можуть на це вплинути:

По-перше це корпоративна культура: команди використовують різні підходи до управління TD.

«Управління технічним боргом передбачає пошук найкращого компромісу для команди проекту. Це передбачає готовність прийняти деякі технічні ризики для досягнення бізнес-цілей та розуміння необхідності загальмувати очікування споживачів щодо забезпечення якості програмного забезпечення» [11]. Отже, ми чітко бачимо, що впровадження та управління TD дуже залежать від бізнес-цілей проекту та його замовників.

По-друге - проблеми синтаксичного аналізу: певні коміти можуть мати код, який не компілюється нормально або мати різні проблеми.

Як було обговорено раніше, проблеми аналізу в основному пов'язані зі спеціальними структурами коду, які, як правило, є загальним шаблоном для одного конкретного проекту.

В результаті, на основі отриманих результатів, відповідь на RQ1: зв’язок між розміром файлу та відсотком SATD не було знайдено.

4.2 Порівняння проблем пов’язаних з SATD з іншими видами проблем

Ми проаналізували проблеми вихідного коду SonarQube, виявлені в блоках коду з коментарями SATD. Дані були отримані з використанням SQL-запиту INNER JOIN між таблицями, тому, якщо одним методом виявлено кілька проблем, вони будуть представлені в різних запитах. Критеріями збігу між проблемою та коментарем були рядки коду в коментованому методі, тому ми могли брати до уваги лише відповідні хеші комітів. У наступних комітах деякий код можна було додати або видалити, а номера рядків буде змінено.

Результати, зібрані таким чином, були перетворені у файли .csv та використані як вхідні дані для функцій R.

Існує порівняння питань, пов’язаних із SATD і не пов’язаних із SATD за кількома параметрами. Оскільки наші результати включають лише проблеми, пов'язані з SATD, і оскільки ми не проаналізували всі проекти та коміти, ми вирішили також порівняти ці результати із загальними проблемами, виявленими авторами вихідного набору даних [3].

Порівняння проводили з використанням відсотків, через різні розміри вибірки даних.

Рисунок 5.2 - Порівняння відсотків різних типів проблем

Як ми бачимо на малюнку вище, code smell - найпоширеніший тип в обох випадках. У проблемах, пов’язаних із SATD, більше помилок, ніж вразливостей, тоді як у цілому ситуація протилежна. Для того, щоб перевірити залежність змінних було проведено тест Пі-Пісона Хі-квадрат.

Результати представлені нижче в таблиці 5.2. Значення Х-квадрата вище критичного, тому ми можемо відхилити H0 про відсутність різниці між розподілами.

Таблиця 5.2 - Тест хі-квадрат типів недоліків

	X-squared	df	p-value
Pearson's Chi-squared test	74.947	2	< 2.2e-16
Critical value (.99)	13.81		

Рисунок 5.3 - Порівняння ступеня серйозності проблем, пов'язаних з SATD, та проблем загалом

Як ми бачимо на малюнку вище, проблеми, пов'язані з SATD, мають вищий відсоток major та critical рівня тяжкості, тоді як набагато менше тяжкості info та blocker. Можна сказати, що проблеми, що стосуються SATD, мабуть, мають більший ступінь серйозності порівняно з проблемами загалом.

Таблиця 5.3 - Тест хі-квадрата на серйозність проблеми

	X-squared	df	p-value
Pearson's Chi-squared test	817.66	4	< 2.2e-16
Critical value (.99)	18.46		

Для того, щоб перевірити залежність змінних було проведено тест Пі-Пісона Хі-квадрат. Результати представлені вище в таблиці 5.3. Значення Х-квадрата вище критичного, тому ми можемо відхилити Н0 про відсутність різниці між розподілами. Найпоширеніші проблеми представлені нижче.

Таблиця 5.4 - Проблеми, пов’язані з SATD, що відповідають правилам SonarQube

Правило SonarQube	Кількість проблем	Відсоток проблем
S1172 - Невикористовувані парамерти методів необхідно видалити	716	6.98%
S3776 - Когнитивна складніть методів не має буди занадто високою	688	6.71%
S116 - Перейменуйте це поле	646	6.3%
S1192 - Строкові літерали не повинні повторюватись	622	6.07%
Повторювані блоки	401	3.91%
S125 Видаліть закоментовані рядки	398	3.88%
Інше	6784	66.15%

Як зазначено в таблиці 5.4, найпоширеніші проблеми, пов'язані з SATD, пов'язані з дублюванням коду, складністю методів, а також ті, що потребують незначних змін. Вони, швидше за все, спричинені швидкими виправленнями, такими як вставка копій, код коментування тощо. Отже, здається розумним, що ці швидкі виправлення можуть співвідноситися з коментарями SATD.

Представлено дві групи проблем, пов’язаних із дублюванням коду, що в цілому становить 9,98%. Проблеми когнітивної складності складають 6,71%, тоді як менш значущі, такі як «Параметри невикористаного методу» та «Перейменувати це поле», складають загалом 13,01%.

Таблиця 5.5 - Найпоширеніші правила SonarQube в цілому

Правило SonarQube	Кількість проблем	Відсоток проблем
Зайвий імпорт	130564	6.72%
Надлишковій викид помилок	108698	5.6%
S1166 - Виключення необхідно додати в лог	101654	5.24%
S134 - Код не має мітити більше трьох вкладених конструкцій (if/for/while/switch/try)	91184	4.7%
S1192 - Визначте константу замість повторювання	93799	4.83%

цього літералу		
Інше	1276120	6573%

Найбільш універсальні проблеми в тих самих проектах представлені в таблицях 5.5. Виявлено, що є деякі проблеми, пов'язані з дублікатами - 4,83%, тоді як проблеми, пов'язані з складністю (S134) - 4,7%, питання обробки винятків - 5,24% та незначні проблеми рефакторінгу - загалом 12,32%.

Таблиця 5.6 - Тест хі-квадрат розподілу правил SonarQube

	X-squared	df	p-value
Pearson's Chi-squared test	22946	240	< 2.2e-16
Critical value (.99)	313.43		

Для того, щоб перевірити залежність змінних було проведено тест Пі-Пісона Хі-квадрат. Результати представлені вище в таблиці 5.6. Значення Х-квадрата вище критичного, тому ми можемо відхилити H0 про відсутність різниці між розподілами.

Як ми можемо помітити, існує велика кількість проблем вихідного коду, проте пропорції залишаються незмінними. Більшість мають тип code smell та major рівень складності. Проблеми, пов’язані з SATD, як правило, мають більший відсоток серйозних проблем, проблем дублювання коду (9,98% порівняно з 4,83%), а також більший відсоток проблем, пов’язаних із когнітивною складністю (6,71% порівняно з 4,7%). Загалом, найпоширеніші проблеми стосуються різних правил SonarQube.

Підводячи підсумок, відповідь на RQ2 полягає в тому, що найпоширенішими проблемами, пов'язаними з SATD, є:

- Невикористані параметри методу слід видалити
- Когнітивна складність методів не повинна бути надто високою

- "Перейменувати це поле"
- Строкові літерали не слід дублювати
- Дубльовані блоки
- Цей блок коментованих рядків коду слід видалити.

Результати показують, що такі типи недоліків стосуються таких серйозних проблем, як методи з високою когнітивною складністю, продубльований код, а також незначних проблем, таких як невикористані параметри методу та неправильна назва поля. Тому розробники залишають коментарі SATD у ситуації, коли потрібні серйозні архітектурні вдосконалення, а також у переважно неякісному коді.

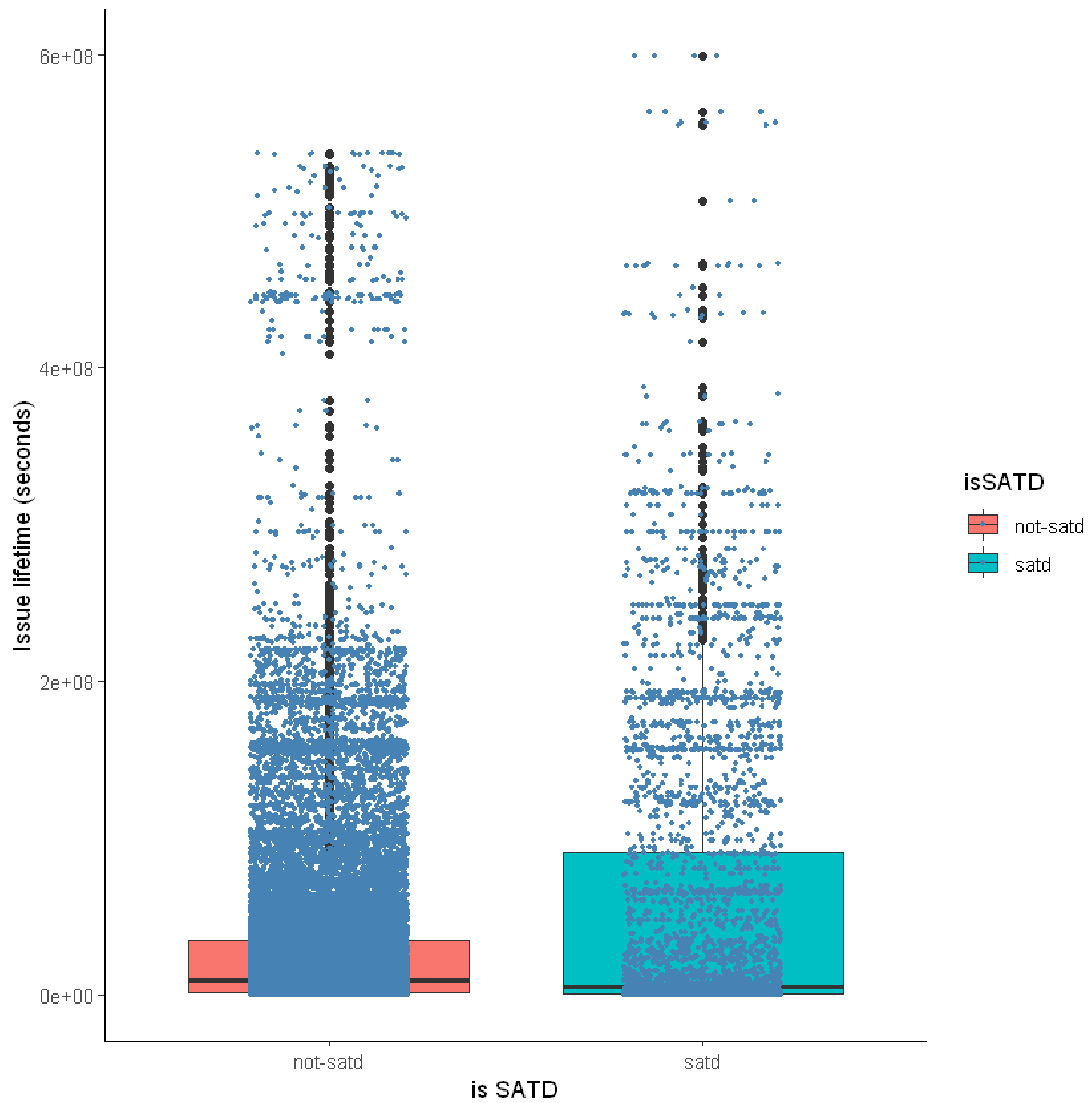
4.3 Аналіз часу виправлення проблем, пов'язаних з SATD

Для того, щоб відповісти на це запитання, слід порівняти дві групи «з SATD» та «без SATD». Ми виконали деякі підготовчі роботи, видаливши нульові значення часу виправлення помилок. Ці значення представляють пошкоджені дані і можуть з’являтися якщо час виправлення помилок не був визначений. Потім ми очистили зразки від дублікатів.

Проаналізована вибірка складається з 40 932 комітів, які не пов'язані з SATD, та 4 107 комітів, що пов’язані. Таким чином, кількість проблем, пов'язаних із SATD, набагато менша.

Термін дії проблеми в основному вимірювали автори набору даних [3], використовуючи алгоритм SZZ. Одиниця виміру - секунда. На рисунку 4.4 наведено графіки, що ілюструють різницю у життєвому періоді проблеми в залежності від наявності SATD. Є деякі викиди.

Рисунок 4.4 - Порівняння часу виправлення помилок з і без SATD



Потім був проведений тест ANOVA, щоб перевірити, чи існує суттєва різниця між групами.

Таблиця 4.7 - Тест ANOVA для порівняння часу випуску вихідного коду з і без SATD.

	Df	Sum Sq	Mean Sq	F value	Pr (>F)	
isSATD	1	3. 862e+18	3.862e+18	1182	<2e-16	***
Residuals	44957	1.469e+20	3.267e+15			
Signif. Code:						

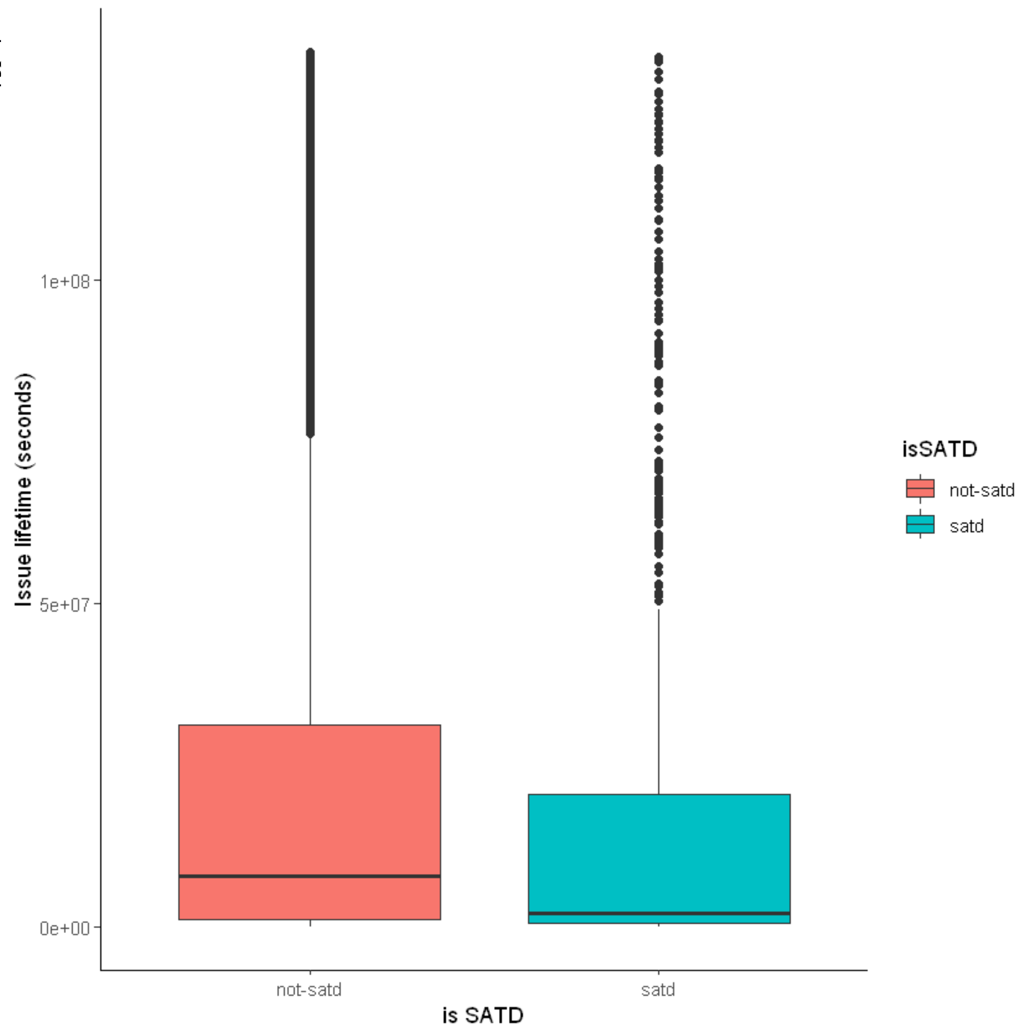


Рисунок 4.5 - Порівняння часу виправлення помилок з і без SATD без викидів

Потім ми видалили викиди (див. рис. 4.5) і отримали інший малюнок. Група з проблемами, які не пов’язані з SATD, є ширшою та ілюструє, що для вирішення цих проблем потрібно більше часу.

Як ми бачимо з тесту ANOVA у таблиці 5.7, між цими групами існує суттєва різниця. Діапазон питань SATD ширший, що може бути спричинено меншими вибірками або іншими ефектами, як описано в [6], однак середній час життя

проблеми менший. Це свідчить про те, що більшість питань вирішується швидше із введенням SATD.

Як результат, ми можемо відповісти на RQ3 наступним: введення SATD зменшує час існування проблеми в коді. Це може стосуватися підвищення очевидності проблемних місць в коді з введенням SATD. Це може означати, що код із загалом низькою якістю і, як правило, має більше SATD.

4.4 Порівняння результатів з отриманими іншими дослідниками.

Отриманий результат демонструє, що впровадження SATD, як правило, пов'язане з певними типами недоліків (RQ2). Ці типи проблем вказують на методи з високою когнітивною складністю, продубльований код, а також на такі проблеми, як невикористані параметри методу та неправильна назва поля. Перші два є найбільш значущими, оскільки вони вказують на найвищий TD [17] і можуть бути пов’язані із серйозною архітектурною проблемою. Останні два є протилежними, оскільки вони просто виявляють «code smells», що означає загалом низьку якість коду. Примітно, що розробники залишають коментарі SATD у ситуації, коли може знадобитися серйозне поліпшення архітектури, а також у переважно неякісному коді.

Подібні висновки можна зробити, розглядаючи зв'язок між часом виправлення проблеми та наявністю SATD у відповідному коді (RQ3). Тест показує суттєву різницю між групами проблем із SATD та групами без нього. Через це нульову гіпотезу можна відкинути.

Що стосується проблем, пов’язаних із SATD, діапазон термінів їх вирішення був набагато ширшим, при цьому медіана значення була нижчою по відношенню до проблем без SATD. Для серйозних питань, пов'язаних з архітектурою, час

виправлення вищий. Видалення дубльованого коду або рефакторинг складних методів може потребувати багато часу, що не завжди відповідає бізнес-цілям. Як це було обговорено в [11], деякі розробники дотримуються принципу «якщо щось не зламано, не виправляйте це». Отже, час виправлення для цієї категорії питань буде подовжено. Інші проблеми, такі як невикористані параметри методу та неправильна назва поля можна швидко виправити. Ці два типи разом створюють згаданий широкий діапазон часу виправлення проблем.

Що стосується зв'язку між розміром проекту та сумою SATD (RQ1), він не виявлений, а це означає, що нульову гіпотезу не можна відкинути.

Під час нашого аналізу ми не виявили жодного зв'язку між розміром проекту та відсотком файлів SATD у ньому. Однак при розгляді великої вибірки, де аналізується більше проектів, це може змінитися. Найменші проекти (категорія <500) мають найширший діапазон SATD, а більші - вужчий. Це звучить логічно і зрозуміло, а також добре сумісно з [16], оскільки там було виявлено, що більшість проектів, як правило, зменшують кількість TD з часом. Тест ANOVA цього не довів, але це може бути пов’язано з таким малим обсягом вибірки.

У великому обсязі вибірки (159 програмних проектів) [6] було виявлено зв'язок між розміром системи та кількістю SATD. Автори стверджують, що «кількість екземплярів SATD зростає протягом історії змін програмних систем через введення нових екземплярів» [6].

Загалом, відсоток SATD на рівні деталізації файлів схожий на попереднє дослідження. У роботі [4] загалом було виявлено 2,4% -31% SATD на рівні деталізації файлу. Результат базувався на 3 проектах, і якщо порівнювати їх із відсотками на основі 30 інших проектів, діапазон приблизно від 0% до 20,83% виглядає правдоподібним.

У роботі [17] автори виявили, що проблеми, пов'язані з дублюванням коду, обробкою винятків та складністю, пов'язані з високим рівнем TD. Ті самі типи питань є одними з найбільш розповсюджених у зібраних даних стосовно SATD. Це

також добре сумісно з висновками [16]. Автори виявили, що дубльований код та винятки мають найбільшу технічну заборгованість у проектах. Справді, відсоток проблем, подібних до дубльованого коду, у групі проектів, пов’язаних із SATD, набагато вищий, ніж в інших.

Оскільки зв’язок між TD та SATD не однозначний, ми не можемо сказати, що в цьому випадку ми отримали порівнянні результати. Однак це може означати, що розробники, як правило, виявляють код з високим рівнем TD і коментують його. Причина полягає в тому, що для поточного дослідження «проблеми, пов'язані з SATD», не передбачають того, що проблеми та SATD були додані одночасно (в одному коміті). За нашим визначенням - проблеми та SATD пов’язані, якщо їх було виявлено в одному і тому ж коді.

Що стосується найпоширеніших типів недоліків, пов’язаних із SATD, тут були виявлені проблеми дублювання коду та складності сприйняття. Вони стосуються архітектурних питань і вимагають значного рефакторингу загалом. Це підтверджується висновками, зробленими дослідниками в роботі [5]: «Зміни SATD є складнішими, ніж зміни, що не стосуються SATD». Крім того, в [21] автори роблять висновок, що провідним джерелом технічного боргу є вибір архітектури.

Що стосується тривалості вирішення проблем, досліджених у RQ3, його середнє значення (362 дні) відповідає діапазону, заданому [12] (82–613 днів), тому воно також видається надійним. Інша робота стверджує, що «існує чітка тенденція, яка показує, що після введення SATD спостерігається більший відсоток виправлення дефектів» [5]. Загалом, це пояснює, чому медіана часу виправлення недоліків для змін SATD нижча, порівняно з не SATD. Нарешті, обмеження, розглянуті в цій роботі, подібні до тих, з якими стикалися автори [25]. Це такі проблеми, як відсутні залежності, синтаксичні помилки та неоднозначні типи змінних.

ВИСНОВКИ

Метою даної роботи було дослідити зв'язок між проблемами якості коду, виявленими SonarQube, та проблемами, позначеними як SATD. Кодова база, яка використовувалася для досліджень, була обмежена 30 репозиторіями Apache з відкритим кодом. Мовою програмування, яка використовувалася для аналізу, була Java. Оскільки результати базуються лише на цих проектах і існували певні обмеження, результати не є повною мірою загальними.

Для відповіді на RQ1 був розрахований відсоток файлів, що містять SATD. Повні таблиці з результатами аналізу наведені в додатку А. Діапазон SATD щодо рівня деталізації на рівні файлу відрізняється від 0% –20,83%, якщо ми розглядаємо SATD, визначений принаймні одним методом, і від 0% –18,06%, якщо SATD було визначено обома методами. За допомогою тесту кореляції Пірсона було перевірено, чи існує кореляція між розміром проекту та відсотком SATD. Результати наведені в додатку Б. Значення P занадто високі, кореляції не виявлено. На основі отриманих результатів, відповідь на RQ1: Не знайдено зв'язку між розміром файлу та відсотком SATD.

Для відповіді на питання, пов'язані з RQ2 SATD, та всі недоліки в цілому були порівняні. Критеріями, необхідними для визначення того, що коментар SATD пов'язаний з проблемою SonarQube, було те, що вони повинні бути присутніми в тому самому блоці коду одночасно. Для аналізу даних використовували описову статистику. Для підтвердження різниці між двома розподілами використовували тест хі-квадрат. На основі статистичного аналізу зібраних даних можна зробити висновок, що впровадження SATD пов'язане з певним типом недоліків. Результати показують, що такі типи недоліків стосуються таких серйозних проблем, як методи з високою когнітивною складністю, продубльований код та таких незначних як невикористані параметри методу та неправильна назва поля. Тому розробники

залишають коментарі SATD у ситуації, коли потрібні серйозні архітектурні вдосконалення, а також у переважно неякісному коді.

Для того, щоб відповісти на RQ3, було порівняно дві групи «з SATD» та «без SATD». Автори набору даних [3] вимірювали час на виправлення помилок, виконуючи алгоритм SZZ. Що стосується статистичного аналізу, то групи даних були представлені на графіках та проведено тест ANOVA [20]. Його було обрано, оскільки тест ANOVA - це класичний спосіб вказати, чи існує суттєва різниця між групами даних. Як результат, час виправлення проблеми, пов’язаний із SATD, менший за медіаною значень. Впровадження SATD зменшує час виправлення помилок. Це може бути пов’язано з проблемами коду, що стають більш очевидними з введенням SATD. Як варіант, це може означати, що код із загалом низькою якістю має більше SATD

Отримані результати представляють досить широкий спектр типів недоліків, різних проектів та файлів. Кількість проаналізованих файлів - 81 740, в них недоліків, що позначені SATD - 10 255. Однак реальна кількість може перевищувати ці цифри через різні помилки синтаксичного аналізу, білда проекту тощо. Якщо аналіз закінчувався помилкою, його результати не включалися. Найбільша кількість помилок була пов'язана з аналізом SonarQube. У найкращому випадку всі непрацюючі коміти, код, що не компілюється, непрацюючі тести, недійсний pom.xml можна виправити вручну. Однак це нереально. Ще один ідеальний і нереалістичний сценарій - це коли експерти доменів вручну перевіряють код, пов’язаний із SATD. Це може зробити виявлення SATD дуже точним. Чим більше даних ми зможемо зібрати і чим точнішим стане наш статистичний аналіз, тим точніші висновки ми можемо представити.

Незважаючи на всі обмеження, отримані дані все ще є цінними, а вибірки досить великі. Ми сподіваємось, що поточні висновки можуть допомогти покращити підходи до оцінки якості коду та політику розробки.

Є декілька можливих подальших досліджень на цю тему. По-перше, слід збільшити обсяг проектів, а також дослідити інші мови програмування. Як ми можемо бачити в дослідженнях [19], можуть бути деякі відмінності залежно від того, яку мову програмування ми використовуємо. Інші мови також менше згадуються у дослідженнях, пов'язаних з TD. Отже, може бути цікавим дослідити та порівняти характеристики SATD та якості коду використовуючи різні мови програмування та екосистеми. Крім того, слід застосовувати різні методи виявлення SATD. Як ми знаємо, існують різні методи виявлення SATD, і не всі вони засновані на аналізі коментарів. Є деякі, які використовують, наприклад, системи відстеження проблем [13]. Ці SATD можуть бути різного типу та пов'язані з різними типами недоліків коду, тому це питання також цікаво дослідити.

У цій дипломній роботі не було виявлено суттєвої різниці між розміром проекту та кількістю SATD, але це може бути спричинено невеликим обсягом вибірки. Було б цікаво перевірити це, використовуючи більше проектів. Як ми знаємо, у великому обсязі вибірки (159 програмних проектів) [6] було виявлено зв'язок між розміром системи та кількістю SATD. Це можна перевірити на інших рівнях деталізації (метод, клас), на інших розмірах зразків тощо. Теоретично, в якості вхідних даних можна використовувати ключові слова, подані не в текстовому вигляді [26, 27].

Слід придбати ліцензію розробника SonarQube. Це дозволяє використовувати більше одного аналізатора в ланцюжку, що значно пришвидшить аналіз. Завдяки можливості аналізувати кожен окремий файл у кожному окремому стані коміту, група питань, не пов'язаних із SATD, буде визначена більш точно. Нарешті, можна взяти до уваги більше дослідницьких питань. За результатами досліджень опублікована стаття на 25-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті».

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. SonarQube official website, «SonarQube» https://www.sonarqube.org/ (Accessed . Mar 24, 2021)

2. Cunningham W. The WyCash portfolio management system //ACM SIGPLAN OOPS Messenger. – 1992. – Т. 4. – №. 2. – P. 29-30.

3. Lenarduzzi V., Saarimäki N., Taibi D. The technical debt dataset //Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering. – 2019. – С. 2-11.

4. Potdar A., Shihab E. An exploratory study on self-admitted technical debt //2014 IEEE International Conference on Software Maintenance and Evolution. – IEEE, 2014. – С. 91-100.

5. Wehaibi S., Shihab E., Guerrouj L. Examining the impact of self-admitted technical debt on software quality //2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER). – IEEE, 2016. – Т. 1. – С. 179-188.

6. Bavota G., Russo B. A large-scale empirical study on self-admitted technical debt //Proceedings of the 13th international conference on mining software repositories. – 2016. – С. 315-326.

7. da Silva Maldonado E., Shihab E., Tsantalis N. Using natural language processing to automatically detect self-admitted technical debt //IEEE Transactions on Software Engineering. – 2017. – Т. 43. – №. 11. – С. 1044-1062.

8. Wattanakriengkrai S. et al. Identifying design and requirement self-admitted technical debt using n-gram idf //2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP). – IEEE, 2018. – С. 7-12.

9. Liu Z. et al. Satd detector: A text-mining-based self-admitted technical debt detection tool //Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. – 2018. – С. 9-12.

10. Zazworka N. et al. A case study on effectively identifying technical debt //Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering. – 2013. – С. 42-47.

11. Lim E., Taksande N., Seaman C. A balancing act: What software practitioners have to say about technical debt //IEEE software. – 2012. – Т. 29. – №. 6. – С. 22-27.

12. Maldonado E. S. et al. An empirical study on the removal of self-admitted technical debt //2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). – IEEE, 2017. – С. 238-248.

13. Xavier L. et al. Beyond the code: Mining self-admitted technical debt in issue tracker systems //Proceedings of the 17th International Conference on Mining Software Repositories. – 2020. – С. 137-146.

14. de Freitas Farias M. A. et al. A contextualized vocabulary model for identifying technical debt on code comments //2015 IEEE 7th international workshop on managing technical debt (MTD). – IEEE, 2015. – С. 25-32.

15. de Freitas Farias M. A. et al. Investigating the identification of technical debt through code comment analysis //International Conference on Enterprise Information Systems. – Springer, Cham, 2016. – С. 284-309.

16. Digkas G. et al. The evolution of technical debt in the apache ecosystem //European Conference on Software Architecture. – Springer, Cham, 2017. – С. 51-66.

17. Digkas G. et al. How do developers fix issues and pay back technical debt in the apache ecosystem? //2018 IEEE 25th International Conference on Software analysis, evolution and reengineering (SANER). – IEEE, 2018. – С. 153-163.

18. SonarQube official website "SonarQube concepts", SonarSource S.A https://docs.sonarqube.org/latest/user-guide/concepts/ (Accessed Apr. 24, 2020)

19. Wohlin C., Höst M., Henningsson K. Empirical research methods in software engineering //Empirical methods and studies in software engineering. – Springer, Berlin, Heidelberg, 2003. – C. 7-23.

20. Forsyth D. Resources and Extras //Probability and Statistics for Computer Science. – Springer, Cham, 2018. – C. 355-361.

21. Ernst N. A. et al. Measure it? manage it? ignore it? software practitioners and technical debt //Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. – 2015. – C. 50-60.

22. Maldonado E. S., Shihab E. Detecting and quantifying different types of self-admitted technical debt //2015 IEEE 7Th international workshop on managing technical debt (MTD). – IEEE, 2015. – C. 9-15.

23. L. Pellegrini, V. Lenarduzzi, and D. Taibi. "OpenSZZ: A Free, Open-Source, Web-Accessible Implementation of the SZZ Algorithm", 2019. https://github.com/clowee/OpenSZZ (Accessed Apr. 24, 2020) https://doi.org/10.5281/zenodo.3337791 (Accessed Apr. 24, 2020)

24. Smelyakov K., Chupryna A., Karachevtsev D., Kulemza D., Samoilenko Y., Patlan O. Effectiveness of Preprocessing Algorithms for Natural Language Processing Applications // 2020 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), 6-9 Oct.

25. Barkmann H., Lincke R., Löwe W. Quantitative evaluation of software quality metrics in open-source projects //2009 International Conference on Advanced Information Networking and Applications Workshops. – IEEE, 2009. – C. 1067-1072.

26. Smelyakov K., Yeremenko D., Sakhon A., Polezhai V., Chupryna A. Braille character recognition based on neural networks // IEEE Second International Conference on Data Stream Mining & Processing (DSMP), August 21-25. – 2018. – P. 509-513.

27. Smelyakov K., Datsenko A., Skrypka V., Akhundov A. Efficiency of Image Reduction Algorithms with Small-Sized and Linear Details // 2019 IEEE International

Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), 8-11 Oct. 2019, Kyiv, Ukraine. – P. 745-750.