**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**ІМЕНІ ТАРАСА ШЕВЧЕНКА**

Факультет комп’ютерних наук та кібернетики

Кафедра інтелектуальних програмних систем

**Курсова робота**

за спеціальністю 121 Програмна інженерія

на тему:

**ПОРІВНЯННЯ ІСНУЮЧИХ ЗАСОБІВ СТАТИЧНОГО АНАЛІЗУ КОДУ, ПОБУДОВА НАБОРІВ ТЕСТІВ, ЩО ДЕМОНСТРУЮТЬ ЇХ МОЖЛИВОСТІ**

Виконав студент 3-го курсу

Подважук Олександр Іванович \_\_\_\_\_\_\_\_\_\_

Науковий керівник:  (підпис)

асистент, кандидат фіз.-мат. наук \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Жереб Костянтин Анатолійович (підпис)

Засвідчую, що в цій роботі

немає запозичень з праць інших авторів

без відповідних посилань.

Студент  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

Київ – 2020

**РЕФЕРАТ**

Обсяг роботи 37 сторінок, 18 використаних джерел, 6 рисунків і 2 таблиці. Ключовi слова: SonarQube, DeepCode, SpotBugs, Вразливість, Засіб статичного аналізу коду, Попередження, Порівняння.

Об’єктом курсової роботи є засоби статичного аналізу коду.

Метою цієї роботи є продемонструвати ідею і можливості деяких засобів статичного аналізу коду за допомогою побудованих наборів тестів, проаналізувати результати цих тестів та визначити найбільш ефективні засоби.

За допомогою методу порівняння було порівняно три засоби статичного аналізу коду. Для цього порівняння були обрані засоби, що проводять аналіз коду написаного мовою Java, а саме SonarQube, DeepCode, SpotBugs. Також пояснено за якими факторами вони порівнювалися. В результаті дослідження були роз’яснені всі переваги та недоліки цих аналізаторів та зроблений висновок, що серед даних трьох засобів неможливо визначити найкращий, адже кожен з них кращий в залежності від того, які завдання ставить перед ним користувач.

**ЗМІСТ**

ВСТУП 5

РОЗДІЛ 1 ЗАГАЛЬНА ІДЕЯ Засобів статичного аналізу коду 8

* 1. Історія розвитку методів статичного аналізу коду 8
  2. Особливості роботи засобів статичного аналізу коду 10
  3. Приклади дефектів коду 12
  4. Можливості засобів статичного аналізу коду 12
  5. Існуючі засоби статичного аналізу коду 13
     1. Мови C та С++ 13
     2. Мова С# 14
     3. Мова Java 15
     4. Декілька мов 15

РОЗДІЛ 2 Особливості порівняння засобів статичного аналізу коду 17

* 1. Ідея порівняння засобів статичного аналізу коду 17
  2. Які параметри порівнювати не бажано 17
  3. Відбір корисних попереджень 19

РОЗДІЛ 3 Порівняння засобів статичного аналізу коду з використанням тестової програми 22

* 1. Засоби обрані для порівняння 22
  2. Засіб SonarQube 22
     1. Системні вимоги 23
     2. Ефективність 23
     3. Зручність використання 24
     4. Інтерфейс 24
     5. Підтримка 25
  3. Засіб DeepCode 25
     1. Системні вимоги 26
     2. Ефективність 27
     3. Зручність використання 27
     4. Інтерфейс 27
     5. Підтримка 28
  4. Засіб SpotBugs 28
     1. Системні вимоги 29
     2. Ефективність 29
     3. Зручність використання 30
     4. Інтерфейс 30
     5. Підтримка 30
  5. Швидкість роботи засобів 31
  6. Аналіз попереджень виданих засобами 31
  7. Результати тестування засобів 32

ВИСНОВКИ 34

БІБЛІОГРАФІЯ 36

**ВСТУП**

**Актуальність**. Як можна помітити, на сьогодні індустрія створення програм для всіляких обчислювальних пристроїв знаходиться в стані активного розвитку. Постійно збільшується потужність обчислювальних систем, що надає все більше і більше можливостей для створення високопродуктивних, у тому числі паралельних, програм і програмних комплексів. У зв’язку з цим постійно зростає складність програмного забезпечення, що керує обчислювальними системами. Через високу складність програмних систем процес забезпечення якості програмного забезпечення, що розробляється, потребує нових підходів до процесу перевірки коректності програм як на відповідність потребам користувачів, так і на наявність критичних дефектів і вразливостей безпеки.

Одним із методів контроля якості програмного забезпечення є застосування інструментальних засобів програміста, які призначені для аналізу програм. Галузь створення інструментальних засобів статичного та динамічного аналізу програм активно розвивається з початку 2000-х років. При статичному аналізі коду відбувається аналіз програми без її реального виконання, а при динамічному – в процесі виконання. Як правило, під статичним аналізом мають на увазі аналіз, який здійснюється за допомогою автоматизованих інструментів вихідного або виконуваного коду.

Історично перші інструменти статичного аналізу (часто в їхній назві використовується слово lint) застосовувались для пошуку найпростіших дефектів програм. Вони використовували простий пошук по сигнатурам, тобто виявляли збіги з наявними сигнатурами в базі перевірок. Вони застосовуються і до сих пір та дозволяють визначати «підозрілі» конструкції в коді, які можуть викликати збій в роботі програми при виконанні.

Але в такого метода немало недоліків. Основним є те, що багато «підозрілих» конструкцій в коді не завжди є дефектами. В більшості випадків такий код може бути синтаксично правильним і працювати коректно. Співвідношення «шуму» до реальних дефектів може досягати 100 до 1 на великих проектах. Таким чином, розробнику доводиться тратити багато часу на його відсіювання від реальних дефектів, що зводить всі плюси автоматизованого пошуку на ні. Не дивлячись на очевидні недоліки, такі прості утиліти для пошуку вразливостей до сих пір використовуються. Як правило, вони безкоштовні, так як комерційного застосування вони, по зрозумілим причинам, не отримали.

Друге покоління інструментів статичного аналізу в доповнення до простого пошуку збігів по шаблонам оснащене технологіями аналізу, які до цього застосовувались в компіляторах для оптимізації програм. Ці методи дозволяли завдяки аналізу вихідного коду складати графи потоку керування і потоку даних, що представляють собою модель виконання програми і модель залежностей одних змінних від інших. Маючи дані, графи, можна моделювати, визначаючи, як буде виконуватись програма (по якому шляху і з якими даними). Так як програма складається із багатьох функцій, процедур модулів, що можуть залежати один від одного, не достатньо аналізувати кожен файл окремо. Для повноцінного міжпроцедурного аналізу необхідно всі файли програми і залежності. Основною перевагою такого типу аналізаторів є набагато менша кількість «шуму» за рахунок часткового моделювання виконання програм і здатність виявляти більш складні дефекти.

Наразі розробляється велика кількість академічних і промислових засобів і інструментів аналізу програм. Через фундаментальні обмеження та інженерні компроміси, на які довелося піти, для збільшення продуктивності інструменти статичного аналізу не завжди можуть забезпечити відсутність помилок першого роду в результатах своєї роботи. При цьому аналіз попереджень може займати багато часу роботи висококваліфікованого експерта в області розробки і забезпечення якості програмного забезпечення. У зв’язку з цим виникає потреба пошуку найбільш ефективного засобу статичного аналізу коду, який би забезпечив найкращу якість результатів аналізу та найбільшу продуктивність своєї роботи. Саме тому необхідно провести дослідження та визначити найкращий засіб статичного аналізу коду.

**Метою** цієї роботи є продемонструвати ідею і можливості деяких засобів статичного аналізу коду за допомогою побудованих наборів тестів, проаналізувати результати цих тестів та визначити найбільш ефективні засоби.

**Об’єктом** даної роботи є засоби статичного аналізу коду.

**Предмет** цієї роботи – демонстрація можливостей засобів статичного аналізу коду.

Для досягнення визначеної мети необхідно виконати наступні **завдання**:

1. Розглянути загальну ідею, що лежить в основі роботи засобів статичного аналізу коду, розглянути приклади використання цих засобів.
2. Обрати декілька найбільш вживаних засобів статичного аналізу коду та побудувати для них набори тестів, що демонструють їх можливості.
3. Проаналізувати результати побудованих тестів, обрати найбільш ефективний засіб та зробити висновки.

**РОЗДІЛ 1 ЗАГАЛЬНА ІДЕЯ засобів статичного аналізу коду**

**1.1 Історія розвитку методів статичного аналізу коду**

На зорі комп’ютерної ери досить багато уваги приділялося тестуванню, як методу забезпечення якості програмних систем. У 60-х роках XX століття тести намагалися проводити таким чином, щоб покрити всі можливі ділянки програми на всіх можливих вхідних даних і показати коректність роботи програми (так зване вичерпне тестування). Але кількість можливих шляхів виконання хоч трохи складної програми і кількість значень вхідних даних виявлялася настільки великою, що подібний підхід до забезпечення якості програмного забезпечення був визнаний непрактичним [1]. У лекції «Про надійність програм» Едскер Вібе Дейкстра стверджує, що тестування може використовуватися дуже ефективно для того, щоб показати наявність помилки, але не може показати їх відсутність [2].

Як наголошується в статті «Що ми знаємо про методи виявлення помилок?» [3], в процесі інспекції коду програм виявляється в середньому від 25% до 50% дефектів в програмі, а в процесі тестування від 30% до 60%. Тобто в середньому у добре протестованій програмі, що була перевірена досвідченими розробниками програмного забезпечення, залишається близько половини дефектів, не виявлених на етапах, призначених для забезпечення якості програми. При цьому експерт, який проводить інспекцію коду, в середньому виявляє від 1 до 2,5 дефектів за годину.

У статті, присвяченій історії створення мови С [4], Денніс Рітчі відзначає, що незважаючи на те, що в першій редакції книги «Мова програмування С» було зазначено більшість правил, які привели систему типів мови С до її сучасної форми, багато програм були написані в старому, більш вільному стилі, а компілятори це дозволяли. Для того, щоб звернути увагу розробників програм на офіційні правила мови, виявляти хоч і дозволені, але підозрілі конструкції, та допомогти знайти невідповідність інтерфейсів, що не виявляються простими механізмами в процесі роздільної компіляції, Стів Джонсон адаптував свій компілятор **pcc** [5] для створення інструменту **lint** [6], який сканував файли вихідного коду програми і відзначав сумнівні конструкції. Особливістю програми lint була можливість порівнювати відповідності і знаходити відсутність протиріч у всій програмі, шляхом сканування безлічі вихідних файлів і порівняння типів аргументів функцій в місці виклику з типами аргументів функції, зазначених у її визначенні.

Згідно з тезами технічного звіту «Наступне покоління статичного аналізу» [7] програму lint можна назвати першим інструментом статичного аналізу коду, але насправді lint не розробляється з метою виявлення дефектів, які призводять до проблем часу виконання. Насправді, метою створення інструменту було виявлення підозрілих конструкцій в коді з метою допомогти розробникам створювати більш узгоджений код. Під «підозрілими конструкціями» мається на увазі технічно коректний, з точки зору специфікації мови, вихідний код (наприклад, С, С++), але який може привести до такого виконання програми, яке не передбачалося розробником. Проблема полягала в тому, що такий підозрілий код міг виконуватися і часто виконувався коректно. Тому через обмежені можливості інструменту lint рівень шуму (хибнопозитивних попереджень про помилки) був екстремально високий, часто перевищуючи співвідношення шуму і реальних дефектів, що в середньому становило 10 до 1. На початку 2000-х років почали з’являтися інструменти статичного аналізу програм другого покоління [7], такі як: Coverity Prevent, Klocwork і інші [8]. Основною особливістю даних інструментів було зміщення фокусу з «підозрілих конструкцій» на «дефекти часу виконання». Для цього були потрібні комбінації витонченого аналізу шляхів виконання з міжпроцедурним аналізом, щоб зрозуміти, що відбувається, коли керування передається від однієї функції до іншої всередині програмної системи [9]. Через постійну боротьбу за точність і масштабованість аналізу інструменти другого покоління могли знаходити обмежений набір дефектів досить точно, але або аналіз не масштабувався на мільйони рядків вихідного коду, або міг працювати досить швидко, проте мав малу точність, як інструмент lint, привносячи вже відомі проблеми шуму попереджень [9, 10]. Боротьба за ілюзорний баланс між точністю, масштабованістю і продуктивністю аналізу призвела до проблеми хибнопозитивних попереджень про дефекти в інструментах другого покоління. Якщо для інструментів першого покоління проблема шуму в результатах роботи перешкоджала їхньому впровадженню в індустрію, то інструменти другого покоління зрушили розвиток інструментів статичного аналізу програм з технологічно мертвої точки, дозволяючи виявляти осмислені дефекти в програмах, але розробники програмного забезпечення все частіше очікують кращої точності результатів аналізу.

У зв’язку з розвитком алгоритміки і появою інструментів для вирішення формул в булевих обмеженнях (вирішувачів) [11], останнім часом почали з’являтися інструменти статичного аналізу третього покоління, які поєднали класичні методи аналізу шляхів виконання програми з застосуванням вирішувачів. Згідно з дослідженням Чельфа і Чу [7] застосування такої комбінації дозволило отримати подальші технологічні переваги для статичного аналізу і знизити високі рівні попереджень помилок першого роду (хибнопозитивних попереджень) на 15%.

**1.2 Особливості роботи засобів статичного аналізу коду**

Статичний аналіз вихідних текстів програмного забезпечення тісно пов’язаний з принципами роботи компіляторів. Багато підходів такого аналізу засновані на деяких елементах компіляції, зокрема, проміжне представлення вихідних текстів в статичних аналізаторах еволюціонує разом з розвитком теорії компіляторів. Сучасні методи аналізу з метою уніфікації алгоритмів використовують різні моделі представлення коду, наприклад лексичний розбір, синтаксичне дерево, дерево Канторовича, графи потоку даних і керування і т. д. [12].

Підхід, який отримав назву сигнатурний аналіз, має на увазі пошук дефектів в програмному коді шляхом зіставлення фрагментів коду зі зразками з бази даних шаблонів (сигнатур) дефектів безпеки. Залежно від технології, застосовуваної при зіставленні фрагмента коду і шаблону, а також від проміжного представлення, можуть використовуватися як звичайні алгоритми пошуку підрядка в рядку, так і регулярні вирази, спеціальні мови запитів по структурованій інформації, зокрема **XQuery** для XML, або спеціально розроблені для цього завдання способи зіставлення. В [13] наведені приклади правил формування сигнатур помилок, що відповідають стандарту CWE. Ознакам потенційно небезпечних конструкцій в програмних системах і методам оцінки їх безпеки присвячені статті [14, 15].

Найпростіший спосіб пошуку потенційно небезпечних конструкцій – застосування регулярних виразів. Цей варіант досить простий в реалізації (легко додаються нові шаблони вразливостей), однак має кілька критичних недоліків. В першу чергу до них відноситься неможливість створити шаблони для складних дефектів, причому навіть для простих дефектів коду регулярний вираз може займати дуже багато місця, а також дуже низька швидкість пошуку.

Більш ефективним вважається побудова проміжного представлення коду і його аналіз. Саме такий спосіб використовується майже у всіх сучасних аналізаторах коду. Загальне уявлення структури статичного аналізатора показано на рисунку 1.



Рисунок 1 – загальна структура статичного аналізатора

На початковому етапі рішення задачі аналізу вихідного коду нагадує дії компілятора або інтерпретатора (у випадку динамічної мови). Вихідні тексти розбиваються на лексеми, на підставі яких згодом будується **абстрактне синтаксичне дерево** (Abstract syntax tree – AST) і в подальшому **семантичний граф** (Abstract semantic graph – ASG). Далі AST і ASG аналізуються механізмом, подібним пошуку регулярних виразів в тексті, або використовуються скріпти для аналізу. Крім цього, аналіз графів виконання і потоків даних (Data Flow analysis) може дати корисну інформацію.

**1.3 Приклади дефектів коду**

Найпоширеніші види дефектів коду – помилки програмування і зловмисно зіпсований код. Слід зауважити, що не всяка помилка програмування призводить до виникнення уразливості, а тільки ті, в результаті експлуатації яких страждають захищеність, цілісність і доступність даних. Зловмисно зіпсований код, як правило, дозволяє отримати несанкціонований доступ до системи, але бувають і інші типи дефектів, що призводять, наприклад, до віддаленого виконання коду, збору і відправлення конфіденційної інформації або до реалізації несанкціонованих дій при настанні певного моменту часу (time bomb).

До найбільш відомих і небезпечним вразливостей відносять вразливості наступних видів:

1. Відсутність перевірки введених даних – може привести до реалізації SQL-ін’єкцій, ін’єкцій коду, міжсайтового скріптингу, обходу каталогів.
2. Помилки роботи з пам’яттю – переповнення буфера, необнулені показники.
3. Помилки реалізації системи безпеки – дозволяють експлуатувати давно відомі «закриті» уразливості.

**1.4 Можливості засобів статичного аналізу коду**

Насправді, на практиці завдання, що вирішуються аналізом, набагато ширші. Адже в цілому статичний аналіз – це будь-яка перевірка вихідних кодів, здійснювана до їх запуску. Ось деякі речі, які можна робити:

1. Перевірка стилю кодування в широкому сенсі цього слова. Сюди входить як перевірка форматування, так і пошук використання порожніх (зайвих) дужок, установка порогових значень на метрики, наприклад, кількості рядків, цикломатичної складності методу і т. д. – все, що потенційно ускладнює читання і підтримування коду. Програми такого класу зазвичай називаються «лінтерами».
2. Аналізу може піддаватися не тільки виконуваний код. Файли ресурсів, такі як JSON, YAML, XML, .properties можуть автоматично перевірятися на валідність. Адже краще дізнатися про те, що порушена структура JSON на ранньому етапі автоматичної перевірки.
3. Компіляція (або парсинг для динамічних мов програмування) – це теж вид статичного аналізу. Як правило, компілятори здатні видавати попередження, що сигналізують про проблеми з якістю вихідного коду, і їх не слід ігнорувати.
4. Іноді компіляція – це не тільки компіляція виконуваного коду. Наприклад, якщо у вас документація в форматі AsciiDoctor, то в момент перетворення її в HTML / PDF обробник AsciiDoctor (Maven plugin) може видавати попередження, наприклад, про порушені внутрішні посилання.
5. Перевірка правопису – теж вид статичного аналізу. Утиліта aspell здатна перевіряти правопис не тільки в документації, а й у вихідних кодах програм (коментарях і літералах) на різних мовах програмування, включаючи C, C++, Java і Python. Помилка правопису в інтерфейсі або документації – це теж дефект.
6. Конфігураційні тести, хоча і виконуються в середовищі виконання модульних тестів типу pytest, насправді також є різновидом статичного аналізу, так як не виконують вихідний код в процесі свого виконання.

**1.5 Існуючі засоби статичного аналізу коду**

На ринку існує досить велика кількість засобів статичного аналізу коду для різних мов програмування, таких як:

**1.5.1 Мови C та С++**

1. **Cppcheck.** Дуже популярний безкоштовний відкритий проект. Заявляє про прагнення досягти 100% відсутності помилкових спрацьовувань. Однак побудований на ідеї аналізу за допомогою регулярних виразів. І, як результат, це багато в чому заважає досягненню поставлених цілей. Також через це не може виконати багато діагностики. Також аналізатор не підтримує багато конструкцій, описувані новими стандартами мови C++.
2. **Clang Static Analyzer.** Статичний аналізатор коду для мов C, C++ та Objective-C, вбудований в компілятор Clang. Популярність Clang Static Analyzer викликана відкритістю його коду і можливістю написання власних розширень.
3. **Clang-Tidy** – відрізняється від діагностичних можливостей Clang Static Analyzer тим, що Clang-Tidy – лінтер, що перевіряє відповідність коду стандартам кодування.
4. **Lint.** Згадується тут, тому що має велику історичну цінність. Цей інструмент можна назвати першим статичним аналізатором коду для мови Сі. Назви багатьох сучасних аналізаторів утворюються від слова «lint» (cpplint, PC-Lint, Splint, JSLint, Rpmlint, Puppet Lint, Pylint).

**1.5.2 Мова С#**

1. **ReSharper.** Не є статичним аналізатором в класичному розумінні, оскільки має мало сценаріїв використання. Плагін для Visual Studio, проводить статичний аналіз коду на мові C# і не тільки. У 2020 році ціна на нього за місяць становила $ 12.90 для індивідуального використання.
2. **Roslyn Analyzers.** Набір статичних аналізаторів коду для мов C# і Visual Basic на основі .Net Compiler Platform («Roslyn»).
3. **Security Code Scan.** Статичний аналізатор коду на основі .Net Compiler Platform («Roslyn») для мов C # і Visual Basic для пошуку патернів помилок, пов’язаних з безпекою додатків: SQL Injection, Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), XML eXternal Entity Injection (XXE) і інші. Аналізує вихідний код.
4. **CodeRush.** Плагін для Visual Studio. Продукт комерційний, але є пробна версія.
5. **Parasoft dotTEST.** Набір інструментів для тестування додатків .NET, що включає в себе статичний аналізатор коду. Працює як плагін для Visual Studio. Як і в попередньому випадку, продукт комерційний, є пробна версія.

**1.5.3 Мова Java**

1. **FindBugs.** Найбільш відомий безкоштовний статичний аналізатор Java коду. Аналізує байт-код програми. На даний момент проект мертвий.
2. **SpotBugs.** Став наступником FindBugs. Проект з відкритим вихідним кодом.
3. **IntelliJ IDEA.** Середовище розробки від компанії JetBrains, що містить набір інспекцій коду, які дозволяють знайти, підсвітити і виправити аномалії в коді.

**1.5.3 Декілька мов**

1. **Coverity.** Вважається одним з кращих на ринку статичних аналізаторів коду. Підтримує програму з безкоштовної перевірки відкритих додатків.
2. **Klocwork Insight.** Один з лідерів ринку статичних аналізаторів коду.
3. **SonarQube.** Статичний аналізатор коду для мов Java, C#, Visual Basic, C, C++, Ruby, JavaScript, Perl та ін. для пошуку «запахів» коду, помилок і вразливостей, що розробляється компанією SonarSource.
4. **Checkmarx CxSuite.** Статичний аналізатор для аналізу коду на предмет наявності закладок і вразливостей, що дозволяє виявляти і усувати проблеми безпеки на рівні вихідного коду. Підтримує широкий спектр мов програмування: Java, C#, Visual Basic, C, C++, Ruby, JavaScript, Perl та інші.
5. **DeepCode.** Автори цього швейцарського стартапу стверджують, що DeepCode здатний виявляти такі уразливості, як, наприклад, можливості для міжсайтового скриптинга і SQL-ін’єкцій, так як закладені в нього алгоритми не просто аналізують код як набір символів, а намагаються зрозуміти сенс і мету роботи написаної програми. Побудований на базі машинного навчання.
6. **SapFix.** Побудований на базі машинного навчання. Створений інженерами з Facebook для скорочення часу, що витрачається інженерами на налагодження, а також прискорення процесу розгортання нового програмного забезпечення.
7. **CodeGuru.** Авторами CodeGuru є інженери з Amazon. Побудований на базі машинного навчання. Сильною стороною свого аналізатора вони вважають допомогу в пошуку способів оптимізації продуктивності вашого коду.
8. **Infer.** Побудований на базі машинного навчання. Статичний аналізатор для мов C, C++, Objective-C і Java, розроблений компанією Facebook. Має відкритий вихідний код.

**РОЗДІЛ 2 Особливості порівняння засобів статичного аналізу коду**

**2.1 Ідея порівняння засобів статичного аналізу коду**

Бажання користувачів порівняти між собою різні аналізатори коду зрозуміло і природно. Однак реалізувати це бажання зовсім не так просто як може здатися на перший погляд. Справа в тому, що незрозуміло які конкретно фактори між собою порівнювати.

Якщо відкинути вже зовсім безглузді ідеї типу «порівняти кількість діагностованих помилок» або «порівняти кількість повідомлень, які видав інструмент», то навіть розумний фактор «співвідношення сигнал-шум» не здається ідеальним критерієм оцінки аналізатора коду.

**2.2 Які параметри порівнювати не бажано**

Розглянемо таку просту на перший погляд характеристику, як кількість діагностичних перевірок. Здається, що чим їх більше, тим краще. Але для кінцевого користувача, що використовує певний набір операційних систем і компіляторів, загальна кількість правил нічого не означає. Правила діагностики, актуальні для систем, бібліотек і компіляторів, які він не використовує, нічого йому не дають. Вони навіть заважають йому, обтяжуючи систему налаштувань і документацію, ускладнюють використання і впровадження інструменту.

Тут доречна наступна аналогія. Людина заходить в магазин, щоб купити обігрівач. Їй цікавий відділ побутової техніки, і якщо в цьому відділі великий вибір, то це добре. А ось інші відділи йому не цікаві. Немає нічого поганого в тому, що якщо в цьому магазині можна купити розумний холодильник, диван і пральну машинку. Але наявність відділу розумних холодильників ніяк не покращує асортимент кондиціонерів.

Візьмемо, наприклад, інструмент Klocwork, що підтримує велику кількість різноманітних систем, в тому числі і екзотичних. В одній з таких систем є компілятор, який «ковтає» такий код: «inline int x;».

Аналізатор Klocwork має діагностичне повідомлення, що дозволяє виявити цю аномалію в коді: «The 'inline' keyword is applied to something other than a function or method». Виходить, що ніби добре, що є така діагностика. Але, скажімо, розробник, що використовує компілятор Microsoft Visual C++ або інший адекватний компілятор, ніякої користі від цієї діагностики не має. Visual C++ просто не компілює такий код: «error C2433: 'x': 'inline' not permitted on data declarations».

Інший приклад. Деякі компілятори погано підтримують тип **bool**. Тому Klocwork може попередити про ситуацію, коли член класу має тип bool: «PORTING.STRUCT.BOOL: This checker detects situations in which a struct / class has a bool member». Зрозуміло, що користь від такого діагностичного повідомлення може отримати вкрай малий відсоток розробників.

Подібних прикладів можна навести дуже багато. І виходить, що загальна кількість діагностичних правил ніяк не пов’язана з тим, яку кількість помилок аналізатор виявить в певному проекті. Аналізатор, який реалізує 100 діагностик і орієнтований для Windows-додатків, може знайти набагато більше помилок в проекті, створеному в Microsoft Visual Studio, ніж крос-платформний аналізатор, який реалізує тисячі діагностик.

Отже, ми дійшли до того, що не бажано порівнювати користь аналізатора по такому параметру, як число діагностичних перевірок. Тоді як варіант можна розглянути порівняння кількості перевірок, актуальних для певної системи. Наприклад, відберемо всі правила, які дозволяють шукати помилки в Windows програмах. Але і такий підхід не спрацює з двох причин: по-перше, нерідко буває так, що в якомусь аналізаторі перевірка реалізована одним діагностичним правилом, а в іншом – декількома правилами. І якщо порівнювати за кількістю правил діагностики, то виходить, що один з аналізаторів краще, хоча вони однакові по кількості виявлених помилок. По-друге, реалізація тих чи інших діагностик може бути різної якості. Наприклад, майже в усіх аналізаторах є пошук «магічних чисел». Але, наприклад, в одному аналізаторі можуть виявлятися тільки магічні числа, небезпечні з точки зору перенесення коду на 64-бітові системи (4, 8, 32 і т.д.), то в іншому – просто всі магічні числа (1, 2 , 3 і т.д.). І в таблиці порівнянь в графі «пошук магічних чисел» поставити і там, і там «плюсик» буде недостатньо.

Також часто наводять таку характеристику, як швидкість роботи інструменту, або кількість оброблюваних рядків коду в хвилину. Але ж і вона іноді не має практичного сенсу. Немає ніякого зв’язку між швидкістю роботи аналізатора коду і швидкістю аналізу проекту людиною. По-перше, нерідко запуск аналізу коду проводиться автоматично під час нічних збірок. І важливо просто «встигнути» перевірити до ранку. Хоча, часто розробники програмного забезпечення стикаються з проблемою нестачі часу, адже доводиться мати справу з дедлайнами, тому іноді потрібно якомога швидше проаналізувати програму та знайти можливі помилки та влазливості.

Також часто забувають при порівнянні про такий параметр як зручність використання. Дуже важливо, щоб користувачу було зручно працювати з аналізатором, адже це значно полегшує процес аналізу. Хороший засіб статичного аналізу коду повинен мати зручний та зрозумілий інтерфейс, де повинні виводитись попередження про помилки разом з роз’ясненням їхньої причини та поясненням як їх краще усунути.

**2.3 Відбір корисних попереджень**

Попередження аналізатора пропонують уважно перевірити певні місця в коді. Хоча, часто це не помилки, такі місця заслуговують уваги. Корисно провести рефакторінг таких місць, щоб вони стали більш зрозумілими не тільки аналізатору, а й іншим програмістам.

Тим не менше, бажано порівняти саме кількість помилок або місць, що можуть привести до помилок з часом. Досить важко дати формальне визначення того, що саме можна вважати по справжньому небезпечним фрагментом коду в програмі. Тому розглянемо на прикладі, що буде вважатись за потенційно небезпечний фрагмент коду. Існують діагностики, що попереджують, що функція використовує досить багато стекової пам’яті, наприклад, 16 кілобайт (по замовчуванню розмір стеку програми дорівнює 2 мегабайти). Таке відбувається, якщо в функції, наприклад, створити локальний масив «byte x[2000]». Звичайно це не помилка. Такі функції доволі часто зустрічаються в програмах і, як правило, не приводять до виникнення проблем. Тому такі попередження не цікаві в даному дослідженні і не будуть враховуватись.

А тепер інше попередження, наприклад, виклик в циклі метода add() у статичного екземпляра класу ArrayList. В Java час життя статичних змінних як правило співпадає з часом роботи застосунку. І якщо використовувати метод add() в циклі, то поки не завершиться робота програми пам’ять не буде очищена. А якби цей екземпляр класу ArrayList не був статичним, то пам’ять би очищалась одразу після завершення функції, адже були б видалені всі об’єкти, на які в програмі більше не було б активних посилань. Отже, якщо колекції або об’єкти оголошені як статичні, то вони залишаються в пам’яті протягом всього терміну роботи програми, тим самим блокуючи ресурси, які можна було б використовувати в іншому місці. Таке попередження буде вважатися потенційно небезпечним і буде враховуватись в дослідженні.

Тому при виборі всіх попереджень, які можна оцінити як корисні, було дотримано наступного принципу: діагностичне попередження вважається корисним, якщо вказує на явну помилку або на фрагмент коду, що легко може стати помилкою в процесі рефакторінгу. Розглянемо декілька випадків, щоб було більш зрозуміло, які фрагменти коду вважались небезпечними, а які – ні.

Приклад явної помилки показано на рисунку 2. Не там поставлена дужка. В результаті перевірка працює не коректно.

D:\Kursach\img2.png

Рисунок 2 – Фрагмент коду

Приклад потенційної помилки показано на рисунку 3. Зсув від’ємного значення призводить до невизначеної поведінки. Такий код може давати на деяких компіляторах той результат, який очікує програміст. Але це лише вдача. В будь-який момент код може відмовити. Тому таке попередження вважається цікавим та корисним.

D:\Kursach\img3.png

Рисунок 3 – Фрагмент коду

Приклад коду, який враховувався на рисунку 4. При використанні такої конструкції, яка показана на рисунку 4, деякі аналізатори видавали попередження, що вона не відповідає вимогам і пропонували використовувати конструкцію logger. Але такий фрагмент коду навряд чи приведе до помилок в майбутньому, тому такі попередження вважались не цікавими.

**РОЗДІЛ 3 Порівняння засобів статичного аналізу коду з використанням тестової програми**

**3.1 Засоби обрані для порівняння**

Оскільки на третьому курсі навчання основною мовою програмування, яка вивчалась, була мова програмування Java, то я вирішив обрати для порівняння саме аналізатори, які сканують проекти, що написані мовою Java. Через свою популярність та поширеність для порівняння були обрані наступні засоби статичного аналізу коду:

1. SonarQube
2. DeepCode
3. SpotBugs

Даними аналізаторами була протестована спеціально розроблена тестова програма, на якій вони продемонстрували свої можливості.

**3.2 Засіб SonarQube**

SonarQube – це інструмент з відкритим вихідним кодом, який аналізує якість коду і складає звіти. Він сканує ваш вихідний код в пошуках потенційних помилок, вразливостей і проблем з підтримкою, а потім представляє результати в звіті, який дозволить вам виявити потенційні проблеми у застосунку [16].

Інструмент SonarQube складається з двох субзастосунків: механізму аналізу, який встановлюється локально на комп’ютері розробника, і централізованого сервера для ведення обліку та звітності. Один екземпляр сервера SonarQube може підтримувати кілька сканерів, що дозволяє централізувати звіти про якість коду від багатьох розробників.

SonarQube надає наступні можливості:

1. Підтримка мов Java, C, C++, C#, Objective-C, Swift, PHP, JavaScript, Python і ін.
2. Надає звіти про дублювання коду, дотримання стандартів кодування, покриття коду модульними тестами, можливі помилки в коді, щільність коментарів в коді, технічний борг і інше.
3. Зберігає історію метрик і будує графіки зміни цих метрик в часі.
4. Забезпечує повністю автоматизований аналіз: інтегрується з Maven, Ant, Gradle і поширеними системами безперервної інтеграції.
5. Дозволяє інтегруватися з такими IDE, як Visual Studio, IntelliJ IDEA і Eclipse за допомогою плагіна SonarLint.
6. Забезпечує інтеграцію з зовнішніми інструментами: JIRA, Mantis, LDAP, Fortify і т.д.
7. Можна розширювати існуючу функціональність за допомогою сторонніх плагінів.
8. Реалізує методологію SQALE для оцінки технічного боргу.

**3.2.1 Системні вимоги**

SonarQube сумісний з платформами MacOS X, Unix та Windows. Він інтегрований з JBuilder, BlueJ, CodeGuide, NetBeans, Sun Java Studio Enterprise, IntelliJ IDEA, TextPad, Maven, Ant, Gel, JCreator та Emacs як плагін для кожної IDE.

**3.2.2 Ефективність**

За допомогою SonarQube просканували вихідний код тестової програми і цей аналізатор успішно знайшов потенційні проблеми, які він, як зазначено в документації, і повинен знаходити, наприклад, знаходження пустих операторів try-catch та так званий dead code. Також, крім цих проблем, SonarQube також виявив складні вкладені цикли та інші небажані техніки кодування, такі як використання, наприклад, коротких назв змінних. SonarQube можливо зможе допомогти розробнику покращити свої навички та стиль кодування. Завдяки великій кількості наборів правил, SonarQube відображає досить багато порушень правил, які хоч і не завжди важливі та мають високий ризик призвести до помилок, і тому розробник може сам вирішувати чи враховувати такі попередження і покращувати свій код.

Але велика кількість помилок може стати проблемою, якщо мова йде про файли з дуже великою кількість вихідного коду, тому SonarQube розділяє ризиковані попередження та помилки і показує їх по категоріям, я сортуються по рівню ризику та потенційних проблем, до яких можуть привести дані небезпечні рядки коду. Важливо зазначити, що SonarQube не претендує на те, щоб виявляти вразливості ін’єкцій та такі небезпечні техніки, як жорстко закодовані паролі. З цієї причини SonarQube не повинен використовуватись як аналізатор, що перевіряє безпеку вихідного коду програми, оскільки він не має можливості ідентифікувати такі небезпечні дірки в системі безпеки програми, які допускають проникнення вірусів.

**3.2.3 Зручність використання**

SonarQube – швидкий і простий в налаштуванні, має інструкції щодо встановлення та інформацію про його використання, які доступні в його онлайн посібнику. Детальні інструкції з скріншотами щодо використання його у терміналі та інтеграцією з іншими середовищами там теж можна знайти. Доступний в інтеграції з Maven, Gradle, але потребує додаткового встановлення SonarQube сервера, де будуть зручно виводитися результати перевірки. Після запуску аналізатора на сервері можна знайти детально інформацію про помилки, що розділені на рівні в залежності від їхньої тяжкості та типу порушення. Також пропонує шляхи усунення знайдених проблем.

**3.2.4 Інтерфейс**

Інтерфейс засобу статичного аналізатора коду SonarQube зображений на рисунку 4.

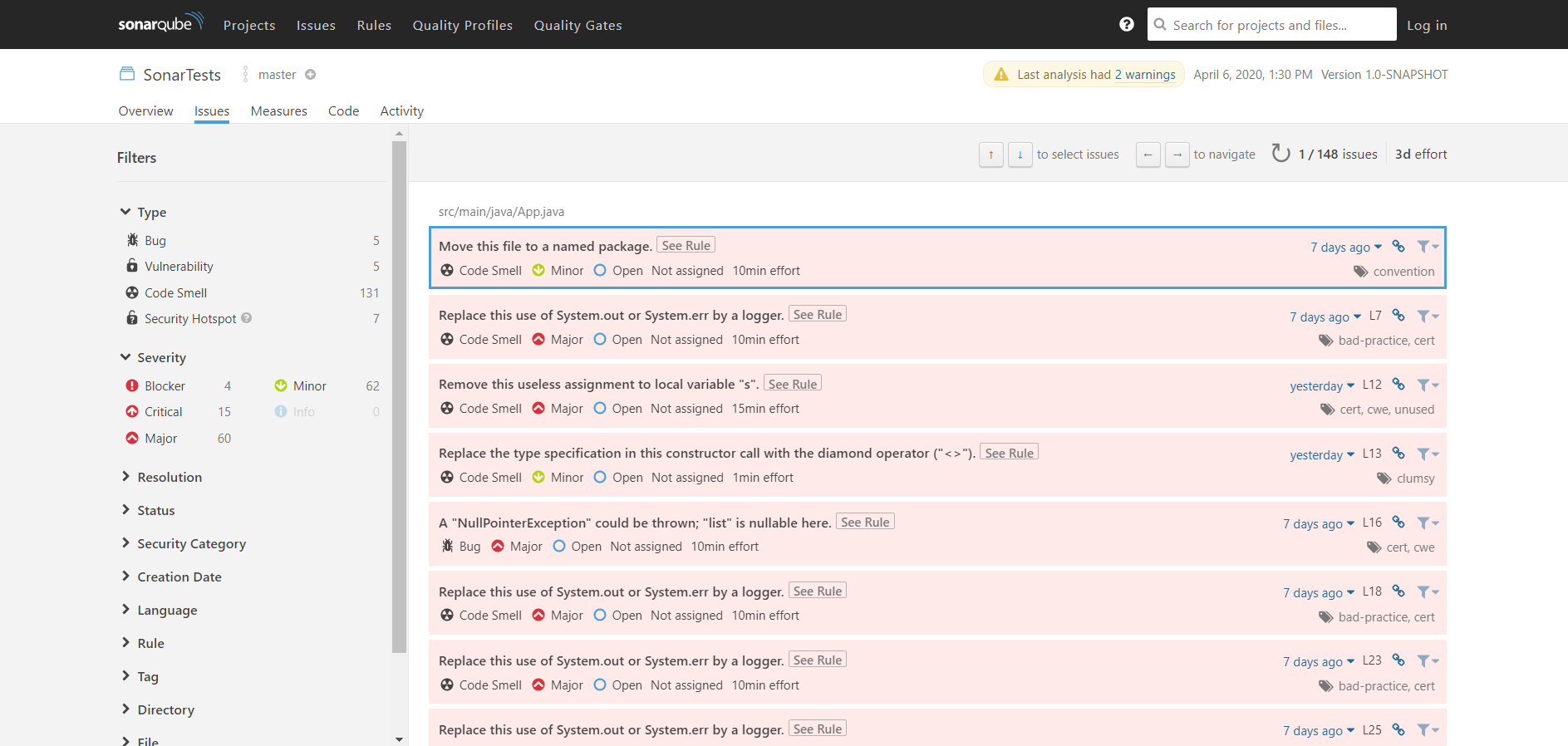


Рисунок 4 – Інтерфейс SonarQube

**3.2.5 Підтримка**

SonarQube має детальну документацію, в якій прописані всі помилки, виправлення, оновлення попередніх версій, а також відповіді на питання, які частіше всього задають про використання аналізатора. SonarQube – дуже популярний засіб статичного аналізу коду, тож він має доволі велику спільноту користувачів, які можливо мають відповіді на питання, що можуть виникнути при роботі з аналізатором. Документація сама по собі детальна, ретельна та охоплює більшість, якщо не всі, аспекти використання аналізатора SonarQube.

**3.3 Засіб DeepCode**

В основі технології DeepCode лежить семантичний аналіз коду укупі з навчанням нейромережі за допомогою Big Data. Найцікавіше в цій розробці те, що в якості бази даних для навчання мережі використовується код публічних репозиторіїв GitHub [17].

Весь процес навчання DeepCode поділяють на кілька етапів. Перший – розмітка бази, тобто парсинг самого GitHub і сортування даних. В якості головного параметра виступає мова програмування, на якій написаний проект. Далі йде розмітка даних і підготовка спаршенного коду до споживання нейромережею. Ну і останній, третій етап – це навчання самого DeepCode.

На виході маємо продукт, який здатний аналізувати завантажений код не тільки з боку синтаксису і наявності фактично помилок, але і з точки зору його корисності. Як приклад наводиться аналіз пул-реквестів в майстер-гілках на тому ж GitHub.

DeepCode здатний відповісти не тільки на питання «скільки в коді помилок», а й видати інформацію щодо кількості нових фіч і потенційних конфліктів з наявною кодовою базою. Тобто розробка здатна проводити як код-рев’ю, так і QA-аудит коду.

На відміну від інших популярних аналізаторів коду DeepCode зосереджений не просто на дотриманні синтаксису і пошуку помилок форматування, але й виявляє і серйозні проблеми. Наприклад, виявлення XSS або SQL-ін’єкцій.

Все це стає доступним саме через початкове джерело даних для навчання системи – завдяки open source проектам на GitHub. Саме завдяки відкритому вихідному коду розробники можуть тренувати мережу не тільки в розрізі того, як повинен виглядати правильний код, але і додавати в процес навчання аналіз даних по внесеним змінам по ходу розвитку проекту. Тобто DeepCode вчиться не на статичних репозиторіях, а аналізує весь коли-небудь написаний в рамках проекту код. Таким чином, система виробляє для себе загальні принципи аналізу і вивчиає логіку розробки, бачить в ході навчання як самі помилки, так і шляхи їх виправлення.

**3.3.1 Системні вимоги**

Для використання DeepCode потрібно мати встановлений будь-який сучасний браузер, такий як Google Chrome, Mozilla Firefox, Opera та ін. А також мати репозиторій на сайті GitHub, BitBucket і т.д., куди потрібно завантажити файли програми з вихідним кодом, щоб аналізатор міг перевірити їх на дефекти.

**3.3.2 Ефективність**

DeepCode успішно виявляє вразливості OS команд, SQL ін’єкції, в також не допускає проникнення вірусів. При тестуванні цього засобу статичного аналізу коду, він повністю виправдав свою роль і продемонстрував здатність виконувати його призначення. Але у цього аналізатора є один недолік, який полягає у тому, що він проводить менш істотну перевірку, адже він виявив меншу кількість помилок у вихідному коді, ніж SonarQube та SpotBugs. DeepCode побачив і показав деякі недоліки, такі, які показали і інші аналізатори, але не всі із них. Також цей аналізатор показав набагато менше попереджень про стиль програмування, ніж SonarQube, але було виявлено більше вразливостей, які впливають на безпеку програми. Хоч він не такий ефективний у лові поганого стилю кодування, однак, виходячи із результатів тестів DeepCode виявляв 100 відсотків помилок, які, як зазначалось в документації, він повинен виявляти. І тому можна сказати, що цей аналізатор має високу ефективність.

**3.3.3 Зручність використання**

Все, що потрібно мати для використання DeepCode – це встановлений браузер та репозиторій на сайті з підтримкою git. Адже DeepCode – це онлайн ресурс, який сканує всі файли, що знаходяться у репозиторії, який потрібно проаналізувати. Це робить процес аналізу помилок у вихідному коді менш стомлюючим, так як всі вразливості, які є в файлах програми, одразу виводяться у одному вікні. Інструменти Vulnerability Source View та Vulnerability Sink View показують короткий опис виявлених помилок, а Provenance Tracker View відстежує кожну вразливість, надає її детальний опис та пропонує шляхи ефективного виправлення такої вразливості, що значно полегшує роботу розробнику.

**3.3.4 Інтерфейс**

Інтерфейс засобу статичного аналізатора коду DeepCode зображений на рисунку 5.

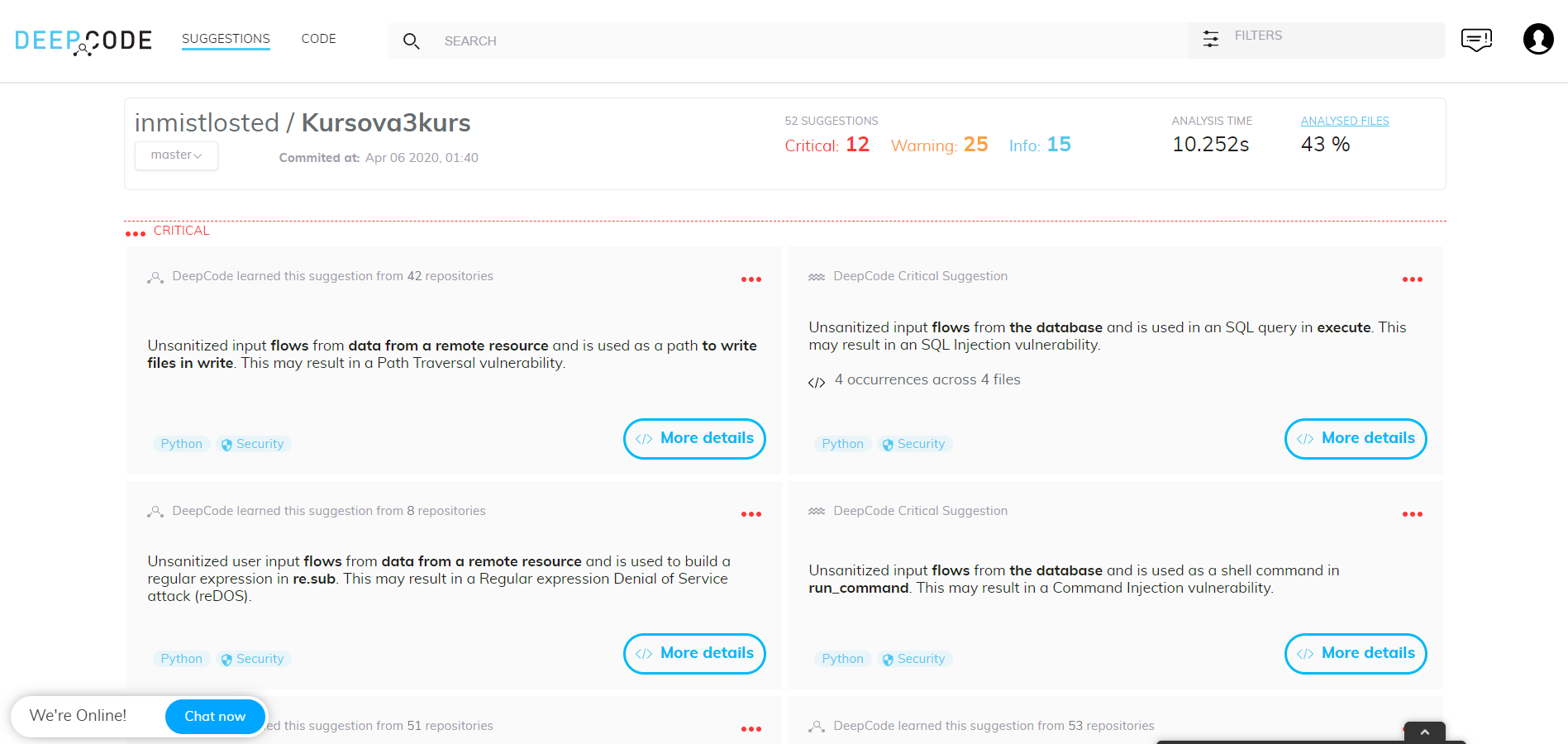


Рисунок 5 – Інтерфейс DeepCode

**3.3.5 Підтримка**

DeepCode має доступну документацію із детальними вказівками щодо використання, приклади як аналізатор працює і скріншоти з описом того як ним слід користуватися. Однак, не так багато існує ресурсів доступних для обговорень щодо використання аналізатора та проблем, як це доступно у випадку з аналізаторами SonarQube і SpotBugs. Хоча, розробник цього засобу статичного аналізу коду також надав електронну адресу на веб-сайті DeepCode для питань щодо використання його інструменту, а також щоб йому повідомляли про проблеми, які виникають.

**3.4 Засіб SpotBugs**

SpotBugs – проект з відкритим вихідним кодом для статичного аналізу коду. Як зрозуміло з назви, він шукає помилки до того, як ви самі їх знайдете. Використовуючи більше 400 шаблонів, SpotBugs поглиблюється в код для виявлення можливих помилок, щоб виявити і виправити їх майже негайно. Потім він дає своїм висновкам один з чотирьох рівнів: «стурбовані», «тривожні», «страшні» і «найстрашніші». SpotBugs є наступником FindBugs, остання версія якого випущена в березні 2015 року. Він допоможе ідентифікувати і класифікувати помилки до того, як вони вплинуть на користувачів [18].

**3.4.1 Системні вимоги**

Spotbugs 3.1.12.2 потребує як мінімум Java сьомої версії і підтримує всі наступні версії. Цей аналізатор буде працювати на платформах Linux, Windows та MacOS X.

**3.4.2 Ефективність**

При аналізі вихідного коду тестового застосунку, SpotBugs вдалося вловити лише дефект NULL показника. Однак в крім цього дефекту він виявив деякі несправності вихідного коду, які не виявлялися іншими аналізаторами. Наприклад, одною поширеною помилкою була помилка «Посилання на кодування за замовчуванням: Знайдено виклик методу, який виконує перетворення byte на String (або String на byte), і буде вважати, що платформа кодування за замовчуванням підходить. Це призведе до зміни поведінки програми в залежності від платформи. Скористайтеся альтернативним API та вкажіть ім’я або об’єкт Charset явно.». Це дуже важливий дефект, який бажано виявити, оскільки розробник може припустити, що за замовчуванням завжди використовується кодування UTF-8. Однак значення строки може бути змінене в залежності від платформи, так як символи ASCII можуть мати різні значення. Наприклад, замість потрібного значення A, на іншій платформі строка може мати значення α. І це лише простий приклад; справа в тому, що призначення рядків коду та їх ефективність можуть відрізнятися в межах різних платформ, якщо значення символів змінюються, коли альтернативний API та ім’я Charset прямо не вказані.

Одним із слабких місць SpotBugs є те, що він вимагає скомпільований код для того, щоб виявити помилки. Якщо код не вдається зібрати, SpotBugs не буде відображати жодні помилки. Однак у FindBugs відображаються помилки, що часто можуть приводити до порушення правильної роботи програми.

**3.4.3 Зручність використання**

Він доступний через інтеграції Ant, Maven, Gradle і Eclipse. Все що потрібно зробити – це підключити плагін SpotBugs у Maven та запустити його. На офіційному сайті SpotBugs доступна проста інструкція з вказівками та скріншотами по налаштуванню і запуску цього аналізатора. Після запуску цього засобу згенерується html-сторінка, відкривши яку, можна побачити всі помилки, які були знайдені у коді програми. На цій стронці у вигляді списку видаються назви файлів, в яких були знайдені помилки, рядки коду з помилками та їх назви, а також вказівки як їх усунути.

* + 1. **Інтерфейс**

Інтерфейс засобу статичного аналізатора коду SpotBugs зображений на рисунку 6.

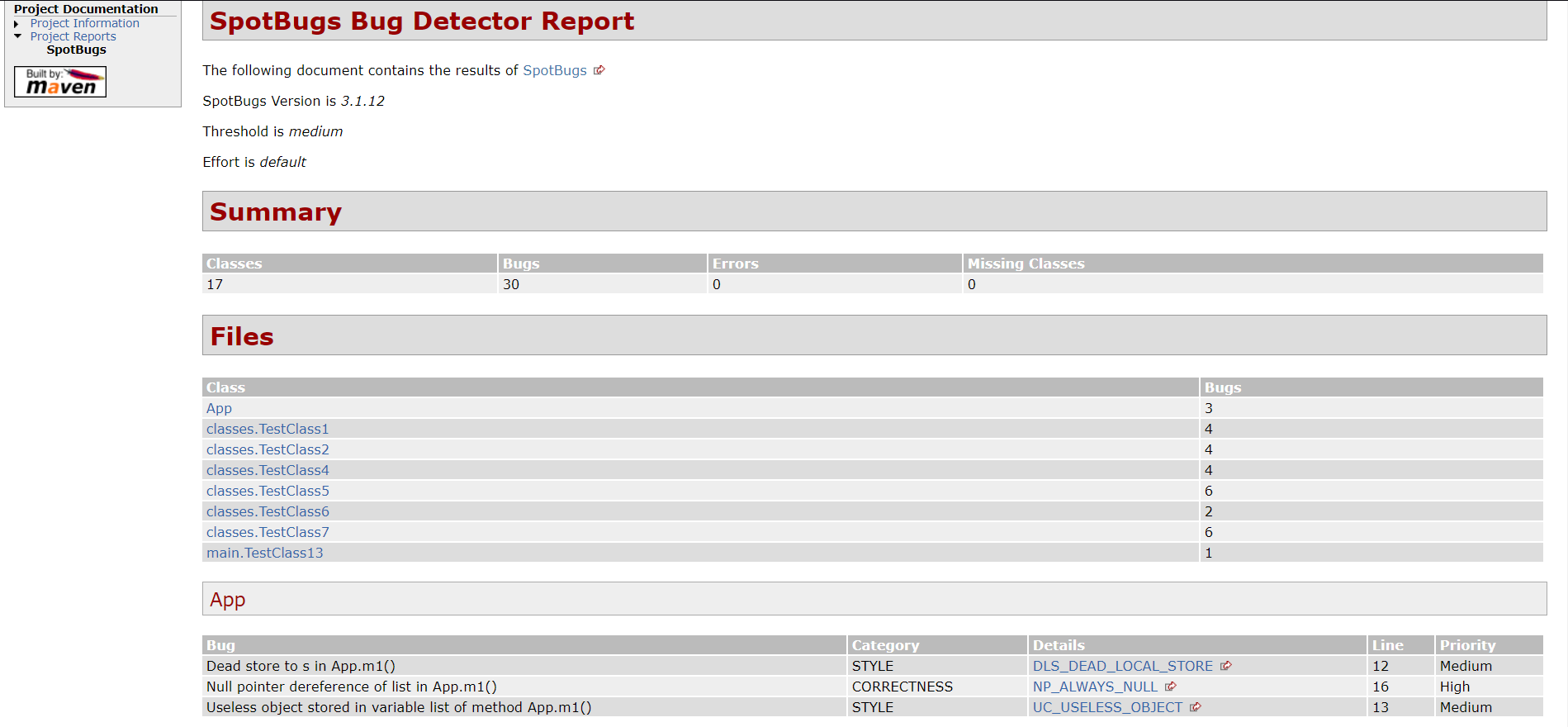


Рисунок 6 – Інтерфейс SpotBugs

**3.4.5 Підтримка**

SpotBugs має доступну документацію з детальними вказівками до використання та типи вразливостей, які аналізатор може виявити. Спільнота розвиває SpotBugs – у нього активна група учасників і супроводжуючів, що просувають проект та допомагають розібратися з помилками, якщо вони виникають.

**3.5 Швидкість роботи засобів**

Швидкість роботи засобу статичного аналізу коду можливо і не головний фактор при порівнянні таких засобів, але точно важливий. Адже іноді буваю дуже важливо якомога швидше проаналізувати код програми та виправити помилки, тому що досить часто програмісти стикаються з всілякими дедлайнами. Саме тому був проаналізований час аналізу засобів SonarQube, DeepCode і SpotBugs, а результати аналізу наведені в таблиці 1.

Таблиця 1 - Час виконання аналізу засобами статичного аналізу коду

|  |  |
| --- | --- |
| Назва аналізатора | Час аналізу, с |
| SonarQube | 24 |
| DeepCode | 10 |
| SpotBugs | 28 |

Як можна побачити із результатів порівняння, найбільш швидким виявився аналізатор DeepCode. Проте як буде видно далі, він видав найменшу кількість попереджень. А найгірший результат показав аналізатор SpotBugs, адже йому для аналізу потрібно було найбільша кількість часу.

**3.6 Аналіз попереджень виданих засобами**

Створений тестовий проект був перевірений без особливих складностей, складніше виявилося продивитись діагностичні попередження і обрати з них найбільш цікаві. Результати такої перевірки наведені в таблиці 2, де показано співвідношення корисних попереджень, що видав кожний аналізатор, до загальної кількості попереджень.

Таблиця 2 - Кількість попереджень, що видали аналізатори

|  |  |  |
| --- | --- | --- |
| Назва аналізатора | Кількість попереджень, шт. | Відсоток корисних попереджень, % |
| SonarQube | 141 | 56 |
| DeepCode | 17 | 65 |
| SpotBugs | 30 | 82 |

Отже, виходячи з результатів аналізу, що наведені в таблиці 2, можна зробити висновок, що найбільшу кількість попереджень видає аналізатор SonarQube, причому, як можна побачити, різниця у кількісті попереджень дуже вагома. Це означає, що саме цей аналізатор проводить найбільш істотну перевірку коду. Хоча з іншого боку відсоток корисних попереджень у SonarQube найменший. Найкраще себе показав аналізатор SpotBugs, тому що він видав середню кількість попереджень і серед цих попереджень відсоток корисних найбільший.

**3.7 Результати тестування засобів**

Якщо проаналізувати всі результати роботи засобів статичного аналізаторів коду, то можна побачити, що засіб SonarQube знайшов найбільшу кількість дефектів з помірною кількістю хибних спрацьовувань. В свою чергу найшвидшим виявився аналізатор DeepCode. Він провів аналіз тестової програми майже втричі швидше за інші аналізатори. Але він видав найменшу кількість попереджень і відсоток корисних складає доволі середнє значення у 65 відсотків. А найбільш корисним виявився аналізатор SpotBugs, адже саме він видав найбільший відсоток корисних попереджень серед всіх засобів статичного аналізу коду. Але недоліком його є те, що засіб SpotBugs затратив найбільше часу для аналізу тестової програми.

Отже, як видно у кожного аналізатора є свої переваги та недоліки. Виявити найкращий з них досить складно. Кожен засіб кращий в залежності від того який аналіз бажає провести програміст. Якщо це істотний детальний аналіз написаної програми, для якої не важливий час та відсоток корисних повідомлень, то краще використовувати аналізатор SonarQube. Якщо потрібно швидко найти найбільш небезпечні помилки, то аналізатор DeepCode в цьому найкращий. А от якщо у розробник не обмежений в часі і йому потрібно якомога більший відсоток корисних серед всіх повідомлень, то найкраще підійде аналізатор SpotBugs.

**ВИСНОВКИ**

Даною роботою були продемонстровані можливості засобів статичного аналізу коду на прикладі розробленого тестової програми. При проведенні дослідження були виявлені як переваги аналізаторів, так і недоліки. Серед переваг можна відзначити:

1. Можливість суттєвого зниження вартості усунення дефектів в програмі. Чим раніше помилку виявлено, тим менша вартість її виправлення. Так згідно з даними, наведеними в книзі Макконнелла «Досконалий Код», виправлення помилки на етапі тестування обійдеться в десять разів дорожче, ніж на етапі конструювання (написання коду).
2. Повне покриття коду. Статичні аналізатори перевіряють навіть ті фрагменти коду, які виконуються вкрай рідко. Такі ділянки коду, як правило, не вдається протестувати іншими методами.
3. Статичний аналіз не залежить від використовуваного компілятора і середовища, в якому буде виконуватися скомпільована програма. Це дозволяє знаходити приховані помилки, які можуть проявити себе тільки через кілька років. Наприклад, це помилки невизначеної поведінки. Такі помилки можуть проявити себе при зміні версії компілятора або при використанні альтернативних джерел для оптимізації коду.

А серед недоліків:

1. Статичний аналіз, як правило, слабкий в діагностиці витоків пам’яті і паралельних помилок. Щоб виявляти подібні помилки, фактично необхідно віртуально виконати частину програми. Це вкрай складно реалізувати. Також подібні алгоритми вимагають дуже багато пам’яті і процесорного часу. Як правило, статичні аналізатори обмежуються діагностикою простих випадків. Більш ефективним способом виявлення витоків пам’яті і паралельних помилок є використання інструментів динамічного аналізу.
2. Програма статичного аналізу попереджає про підозрілі місцях. Це означає, що насправді код, може бути абсолютно коректний. Це називається хибнопозитивними спрацьовуваннями. Зрозуміти, вказує аналізатор на помилку або видав помилкове спрацьовування, може тільки програміст. Необхідність переглядати помилкові спрацьовування забирає робочий час і послаблює увагу до тих ділянок коду, де насправді містяться помилки.

Для порівняння засобів статичного аналізу коду були обрані три аналізатори, що аналізували тестову програму написану на мові Java: SonarQube, DeepCode та SpotBugs. Були порівняні їх ефективність, швилкість роботи, зручність використання, наявність доступної документації та підтримки з боку розробників, а також відсоток корисних попереджень. І в результаті проведеного порівняння не вдалося виявити найкращий з них, адже кожен мав свої переваги та недоліки, і був кращим в залежності від того які завдання перед аналізатором ставив розробник. Виявилося, що DeepCode – найшвидший засіб, SonarQube – проводить найбільш детальну та істотну перевірку та допомагає розробнику покращувати свій стиль програмування, а SpotBugs – надає найбільшу кількість корисних попереджень у відсотковому співвідношенні.

В майбутньому буде проведена робота по дослідженню роботи більшої кількості засобів статичного аналізу коду, які вже будуть аналізувати проекти, що написані на більшості популярних мов програмування, а не тільки Java. Буде розглянута ідея по модифікації існуючого або розробці власного засобу статичного аналізу коду, де був більш широкий набір правил перевірки коду та який би більш ефективно знаходив всі вразливості програм.

**БІБЛІОГРАФІЯ**

1. E. J. Weyuker, T. J. Ostrand. Theories of program testing and the application of revealing subdomains. IEEE Transactions on software engineering, 6(3):236-246. May 1980.
2. E. W. Dijkstra. On the reliability of the programs. https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF
3. Per Runeson, Carian Andersson, Thomas Thelin, Anneliese Andrews, Tomas Berling. What Do We Know about Defect Detection Methods? IEEE Software May/June 2006
4. Dennis M. Ritchie. The development of the C language. Proceedings of HOPL-II The second ACM SIGPLAN conference on History of programming languages. Cambridge, MA, USA - April 20-23, 1993, pp. 201-208
5. S. C. Johnson. A Portable Compiler: Theory and Practice. Proceedings of 5th ACM POPL Symposium, January 1978
6. S. C. Johnson. Lint, a Program Checker. Unix Programmer's manual, Seventh Edition, Vol. 2B, M.D. Mcllroy and B.W. Kernigan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.
7. Benjamine Chelf, Andy Chou. The next generation of Static Analysis. Coverity, March 18, 2008. http://www.coverity.com/library/pdf/Coverity White Paper-SAT-Next Generation Static Analysis.pdf
8. Par Emanuelsson, Ulf Nilsson, A Comparative Study of Industrial Static Analysis Tools. Technical report. Department of Computer and Information Science, Linkoping University. Linkoping, Sweden, 2008.
9. Dawson Engler, Benjamin Chelf, Andy Chou, Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. 0SDI'00 Proceedings of the 4th conference on Symposium on Operating System Design and Implementation, Volume 4, Article No. 1. San Diego, California - October 22-25, 2000
10. Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, Robert Bowdidge. Why don't software developers use static analysis tools to find bugs?. ICSE'13 Proceedings of the 2013 International conference on Software Engineering. San Francisco, CA, USE, May 18-26, 2013
11. John Franco, John Martin. A history of Satisfiability. Handbook of Satisfiability. IOS Press, 2009 doi:10.3233/978-1-58603-929-5-3
12. Компілятори: принципи, технології та інструментарій / Ахо А. В. [та ін.]. М .: ТОВ «І. Д. Вільямс ». 2016. 1184 с.
13. Марков А. С., Фадин А. А. Систематика вразливостей і дефектів безпеки програмних ресурсів // Захист інформації. Інсайд. 2013. № 3 (51). С. 56-61.
14. Жидков І. В., Кадушкін І. В. Прлшо ознаки потенційно небезпечних подій в інформаційних системах // Питання кібербезпеки. 2014. № 1 (2). C. 40-48.
15. Рібері Г., Малмквіст К., Щербаков А. Багаторівневий підхід до оцінки безпеки програмних засобів // Питання кібербезпеки. 2014. № 1 (2). C. 36-39.
16. <https://docs.sonarqube.org/latest/>
17. <https://deepcode.freshdesk.com/support/solutions/articles/60000346607-what-is-deepcode->
18. <https://spotbugs.readthedocs.io/en/stable/introduction.html>