Міністерство освіти і науки України
Національний університет “Києво-Могилянська академія”
Факультет інформатики
Кафедра інформатики

Магістерська робота

на тему: **“ДОСЛІДЖЕННЯ МЕТРИК ЯКОСТІ НА ОСНОВІ СТАТИЧНОГО АНАЛІЗУ КОДУ”**

Виконав: студент 2-го року навчання
Спеціальності
122 Комп’ютерні науки
Моренець Ігор Едуардович

Керівник Гороховский С. С.,
кандидат фізико-математичних наук, доцент

Магістерська робота захищена
з оцінкою _____
Секретар ЕК _____
“____” _____ 2021 р.

Київ 2021

Національний університет “Києво-Могилянська академія”

Факультет інформатики
Кафедра інформатики
Освітній ступінь магістр
Спеціальність 122 Комп’ютерні науки
Освітньо-наукова програма Комп’ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри інформатики

Гороховський С. С.

“____”_____ 2021 року

ЗАВДАННЯ
ДЛЯ МАГІСТЕРСЬКОЇ РОБОТИ СТУДЕНТУ

Моренцю Ігорю Едуардовичу

1. Тема роботи: Дослідження метрик якості на основі статичного аналізу коду
керівник роботи Гороховський Семен Самуїлович, кандидат фізико-математичних наук, доцент
затверджені наказом вищого навчального закладу від “____”_____ 2021 року № ___
2. Строк подання студентом роботи 4 червня 2021 року
3. План роботи

Вступ
1. Аналіз предметної області
2. Опис та аналіз різних метрик та методів їх реалізації
3. Опис архітектури практичної реалізації системи
Висновки
Список джерел
Додатки

ГРАФІК ПІДГОТОВКИ МАГІСТЕРСЬКОЇ РОБОТИ ДО ЗАХИСТУ

№	ПЕРЕЛІК РОБІТ	Термін виконання	Дата ознайомлення наукового керівника	Підпис наукового керівника
1.	Вибір теми, затвердження її на засіданні кафедри та закріплення наукового керівника. Узгодження календарного графіка підготовки магістерської роботи. Ознайомлення студента з критеріями оцінювання магістерської роботи.	жовтень		
2.	Вивчення літератури за темою роботи	жовтень - грудень		
3.	Складання плану роботи та узгодження з науковим керівником	грудень		
4.	Написання розділів роботи	грудень - квітень		
5.	Проміжний контроль виконання роботи	березень		
6.	Написання роботи в цілому, ознайомлення з її першим варіантом наукового керівника	березень-квітень		
7.	Повне завершення написання роботи, оформлення її згідно з вимогами й подання на відгук науковому керівнику	початок травня		
8.	Підготовка до захисту роботи на засіданні кафедри: написання доповіді та виготовлення ілюстративного матеріалу	початок травня		
9.	Попередній захист роботи на засіданні кафедри	середина травня		
10.	Подання роботи на кафедру з усіма супроводжувальними документами	початок червня		
11.	Публічний захист роботи перед екзаменаційною комісією	середина червня		

Графік узгоджено “____”_____ 2021 року

Науковий керівник Гороховський С. С.

Виконавець магістерської роботи Моренець І. Е.

ВІДГУК
про магістерську роботу
на здобуття освітнього ступеня магістра

студента НаУКМА Моренця Ігоря Едуардовича
на тему: ***Дослідження метрик якості на основі статичного аналізу коду***

1. *Актуальність теми*: Складність розробки великих систем програмного забезпечення призводить до великих проблем у їх виконанні. Отже забезпечення можливості раннього виявлення помилок та оцінювання якості проекту і прогнозування рівня якості проєкту ПЗ дають можливість зменшити витрати на розробку ПЗ і уникнути багатьох проблем, які можна минати ще на етапах формулювання вимог. Але є ще один підхід, що полягає в аналізі якості статичного коду на основі метрик, підтриманий засобами автоматизації. Тому тема роботи, де запропоновані, глибоко досліджені й реалізовані методи аналізу якості програмного коду вельми актуальна..
2. *Наявність новизни*: У роботі виконано глибокий аналіз метрик, причому метрики використовуються не лише для пошуку помилок, а також для інформування про складність та інші характеристики коду, Також вироблено систему показників для порівняння методів і підходів, а також спроектована і реалізована ефективна система глибокого аналізу якості програмних проектів.
3. *Відповідність змісту роботи її плану*: Зміст відповідає плану.
4. *Ступінь розкриття теми роботи*: Тема розкрита повною мірою. Є солідна практична реалізація системи.
5. *Ілюстрованість роботи (наявність розрахунків, таблиць, схем, діаграм, тощо)*: Схеми та таблиці показують суть роботи.
6. *Якість оформлення роботи*: висока.

7. *Відповідність роботи спеціальності*: повна відповідність.

8. *Недоліки*: Трапляються синтаксичні огріхи.

9. *Загальний висновок*: допускається до захисту.

10. *На яку оцінку заслуговує робота*: робота заслуговує на оцінку відмінно.

Науковий керівник
кандидат фіз.-мат. наук

5 червня 2021 р. Гороховський С. С.

Зміст

Анотація

В цій роботі були досліджені та описані деякі метрики програмних проектів що визначаються за допомогою статичного аналізу, їх методи реалізації, а також загальні підходи до визначення, імплементації та оцінювання таких метрик. Крім того, була описана система для статичного високорівневого, комплексного та водночас глибокого аналізу програмних проектів.

Перелік термінів та скорочень:

- Null pointer - указник що нікуди не вказує. Розіменування такого указника зазвичай призводить до помилки виконання;
- CI - Continuous Integration;
- VR - Virtual Reality, віртуальна реальність;
- Bug tracker (багтрекер) - система що дозволяє тримати інформацію про помилки в програмному продукті в одному місці;
- AST - Abstract Syntax Tree, Абстрактне синтаксичне дерево;
- Corner case - ситуація яка стається під час особливих чи нестандартних умов використання програмного продукту;
- API - Application Programming Interface;
- TSLint - лінтер для TypeScript;
- Redux - бібліотека для керування станом клієнтських застосувань. Часто використовується у поєднанні з react-redux та redux-thunk;
- SemVer - Semantic Versioning;
- LTS - Long term support, довготривала підтримка;
- TODO - коментар в коді що сигналізує що якась робота була залишена на потім;
- Git - система контролю версій;
- Asset (ассет) - додаткові цифрові файли в проекті, зазвичай медіа;
- CLI - Command-line interface, інтерфейс командного рядка;

- Монорепозиторій (монорепо) - репозиторій який зберігає декілька програмних проектів;
- UI - User Interface, користувацький інтерфейс.

Вступ

Якісний продукт завжди буде більше цінитись користувачами ніж той що має багато помилок та проблем. А якісний програмний код напряму впливає на якість кінцевого продукту [1]. Через це оптимізація якості програмного коду стає однією з першочергових цілей розробників, яка постійно конфліктує з прагненням розробити продукт якомога швидше та дешевше. Особливо складно стає підтримувати якість з ростом кодової бази та збільшенням кількості людей що працюють над нею, через що потрібен постійний ретельний контроль. Типовим та ефективним методом контролю є перегляд коду колегами, але всі люди роблять помилки, включаючи ревьюера (того хто переглядає чужу роботу), і цей метод створює дуже повільний та розтягнутий у часі зворотній зв’язок, що призводить до зниження продуктивності та моралі команди.

Однак дуже багато типових помилок можна відловлювати автоматизовано і надійно за допомогою статичного аналізу коду. Майже жоден проект не обходиться без принаймні якогось виду такого аналізу, найпопулярнішим з яких є статична перевірка типів. Але через багатогранність та складність програмних проектів є дуже багато способів їх статичного аналізу, якісне використання яких може суттєво пришвидшувати роботу всієї команди та одночасно відносно дешево покращувати якість коду та продукту.

Через це є критично необхідним визначати важливі метрики для аналізу та помилки для відловлювання, а також підтримання балансу швидкодії та доречності. Саме це є ціллю даної роботи - проаналізувати та описати деякі важливі метрики для статичного аналізу, їх методи реалізації, а головне - представити загальні підходи та поради до визначення, імплементації та оцінювання таких метрик. Крім того, описати

систему для статичного високорівневого та комплексного аналізу програмних проектів.

Вже існує велика кількість робіт на тему статичного аналізу, але багато з них суттєво відрізняються від цієї принаймні одним з наступних способів:

- Описує лише загальні концепти та терміни;
- Концентрується на конкретній метриці або відносно невеликому наборі метрик (зазвичай до десятка);
- Концентрується на відносно невеликій підмножині проектів, наприклад лише на ООП кодових базах, або тих що написані на Java.
- Намагається зробити далекойдучі висновки на основі метрик;

Принциповим аспектом цієї роботи та результуючого програмного продукту є те що всі описані метрики та підходи надаються користувачам як є і вже користувачі (розробники, менеджери, аналітики, тощо) роблять з цього певні виводи, на приклад на основі порівняння з загальним станом всіх проаналізованих проектів. Далі в роботі цей підхід описується більш детально та обгрунтовується.

1. Аналіз предметної області

1.1. Вступ

Статичний аналіз програмних проектів це дуже обширна галузь і покрити її всю в цій роботі неможливо та не доречно. В цьому розділі будуть описані та проаналізовані основні поняття теми та визначена підмножина теми якій присвячена решта роботи.

1.2. Основи

Для початку, статичний аналіз характеризується тим що, на відміну від динамічного, не потребує виконання програми, а оперує над програмним кодом або пов'язаними артефактами, такими як список залежностей, звіт по тестуванню, тощо.

Найчастіше розробники зустрічаються із статичним аналізом коду у вигляді статичної перевірки типів. Часто вона вбудована у компілятор і тому не вважається як щось окреме, але це не завжди так і тоді стає більш помітно що це лише ще один вид статичного аналізу, просто дуже потужний та корисний. Однак є багато не менш популярних використань цього підходу, наприклад:

- Підтримання кодової бази в одному стилі, що корисно коли над проектом працює багато розробників з різним досвідом;
- Пошук типових помилок (відсутність перевірки на null pointer, умова що завжди є хибною, тощо). Інструменти що виконують таку роботу зазвичай називаються лінтерами (linter);
- Генерація іншого коду або документації. Широко використовуваними прикладами таких інструментів є Protocol buffers [2] та Swagger [3];
- Аналіз проекту на відповідність певним метрикам та загальна статистика, потенційно по множині проектів.

Саме останньому пункту і присвячена ця робота. Цей підхід відрізняється від інших тим що зазвичай фокусується не на спрощенні роботи шляхом відлову помилок або генерації коду, а наданні розробникам нової потенційно корисної інформації про їх проекти. Прикладом такого аналізу може бути графік глибин дерев наслідування, або відсоткова статистика по використанню різних мов програмування в проекті.

1.3. Метричний аналіз

Зазвичай інструменти, підходи та патерни у згаданих вище застосувань статичного аналізу досить схожі, однак в метричному аналізі значно більший простір для експериментів, адже отримані результати можуть як надавати корисну раніше невідому інформацію про проект чи множину проектів, так і бути абсолютно неінформативними, а завчасно передбачити результат не завжди можливо.

Також важливою відмінністю підходів є момент в який виконується аналіз. Лінтери мають видавати результат миттєво під час написання коду щоб створити якомога коротший зворотній зв’язок. Статична перевірка типів, підтримання одного стилю та генерація додаткового коду також зазвичай виконуються на комп’ютері розробника під час роботи або принаймні в середовищі неперервної інтеграції (CI). Але є мало сенсу проводити збирання статистичних даних про проект постійно під час роботи. Кращим варіантом може бути робити це через які-небудь проміжки часу, наприклад кожної ночі, або оновлювати дані на кожну зміну в сховищі коду.

Повертаючись до експериментів, метрики та їх презентація можуть і по можливості мають бути адаптовані під конкретний варіант використання. Потенційно впливати можуть такі речі:

- Потреби користувачів (розробників, менеджерів, тощо).

  Якщо потреби скромні то можливо і не варто робити щось складне. Однак чим менше інформації надавати користувачам тим менш корисним їм буде здаватись інструмент і не рідко якраз отримавши якусь частину інформації виникає попит на більш детальний аналіз. Наприклад, дізнавшись що у проекті середній розмір класів становить близько 200 рядків, користувачі можуть захотіти побачити загальні графіки розподілу розмірів або отримати списки класів з посиланнями прямо на файл та рядок в репозиторії.

- Середовище використання продуктів.

  Розробникам веб-інтерфейсів, космічних шатлів, VR-окуляр та банківських систем часто потрібна різна інформація. Комусь критично важлива швидкодія, для когось безпека, а для більшості проектів зазвичай просто важливі гнучкість та простота підтримки.
- Технології в проектах, наприклад мови програмування.

  Різні мови, фреймворки та бібліотеки мають свої особливості та ідіоми які можна використати щоб зробити не просто загальні метрики, а більш спеціалізовані та доречні. В той же час, деякі загальні метрики просто не вийде використати з будь-яким набором інструментів, наприклад все що стосується класів ймовірно не підійде для аналізу проектів у функціональній парадигмі.
- Спосіб презентації результатів.

  Є велика різниця як представляти результати виконання аналізу: одним числом, таблицею, графіком, інтерактивною діаграмою, чи чимось ще складнішим. Здебільшого це впливає на візуальну частину, а не на сам аналізатор, але від детальності візуалізації залежить те наскільки детальним має бути аналіз та наскільки багато даних потрібно збирати. Нерідко інструменти намагаються звести складні дані до одного числа, наприклад виразити складність розуміння коду у межах від 0 до 100. Така абстракція хоча і буває дуже корисною, на жаль може втрачати багато не менш важливої інформації і бути не дуже точною, через що може бути доречно також надавати користувачам результати в менш абстрагованому вигляді щоб вони робили і свої висновки.
- Ресурси.

  Збирання деяких метрик може займати досить багато часу та обчислювальних ресурсів. Інколи може бути простіше та доречніше

просто не використовувати якусь дуже складну метрику, аніж витрачати ці ресурси.

Всі ці аспекти важливо враховувати при виборі та визначені метрик та вибору інструменту для аналізу або побудові архітектури своєї системи.

1.4. Існуючі інструменти

Вже існує багато різноманітних інструментів для статичного аналізу, в тому числі у вільному доступі та з платною ліцензією. Як відомі та потужні платні системи можна окремо виділити PVS-Studio та SonarQube.

PVS-Studio підтримує C, C++, C# та Java і фокусується на детальному та розумному пошуку помилок, будь то опечатки, логічні помилки, проблеми безпеки та інші. Система позиціонується більше як інструмент що працює одночасно з одним проектом і інтегрується з іншими інструментами (включаючи SonarQube) і виконує аналіз як в CI середовищі так і прямо під час роботи розробника [4].

SonarQube також підтримує інтеграцію з системами контролю версій та CI інструментами, але є більш незалежною системою. Вона підтримує 27 мов програмування, надає звіти відразу по декільком проектам та навіть дозволяє додавати свої правила задані мовою Java [5].

Рисунок 1.1 - Приклад частини звіту аналізу SonarQube [5]

Цей дуже потужний інструмент використовується більше ніж двадцятьма тисячами компаній [6], однак все одно деякі компанії створюють та розвивають свої власні інструменти для статичного аналізу, які краще підходять для саме їх специфічних потреб.

Тому залишається актуальним визначення корисних метрик, опис способів ідентифікації, реалізації та оцінки таких метрик та підходи до реалізації загальної системи для проведення аналізу.

1.5. Методи збирання метрик

Зазвичай статичний аналіз проводиться на вихідному коді проекту та його артефактах, тобто вимагає простого доступу до них, наприклад використовуючи системи управління програмним кодом такі як GitHub, хоча може знадобитися доступ і до інших систем, наприклад до багтрекерів або звітів з CI середовищ.

Підходи ж які використовуються для збирання метрик досить різноманітні, але основні можна розподілити на 4 категорії впорядковані за потужністю та складністю використання:

1. Метадані;
2. Регулярні вирази;
3. Абстрактні синтаксичні дерева (AST);
4. Графи зв’язків.

Важливе зауваження - метадані включають дуже широкий спектр метрик, бо за метадані далі будемо вважати всі артефакти та інші дані про проект які не є самим вихідним кодом. Тобто і список залежностей проекту, і кількість розробників, і кількість дефектів в багтрекері будемо вважати метаданими. Тому якщо в загалому концепт метаданих можна вважати найпростішим, в реальних імплементаціях він може інколи бути і найскладнішим. В одному аналізаторі можуть використовуватись відразу декілька підходів, що можна буде побачити на деяких прикладах.

1.6. Висновки

В цьому розділі було визначено та аргументовано навіщо потрібно використовувати статичний аналіз та на які основні моменти варто звертати увагу (наприклад, потреби користувачів та наявні ресурси) при обиранні одного із розглянутих чи якогось іншого інструменту.

Ці ж фактори впливають на те, чи варто розробляти свою систему для статичного метричного аналізу, так як існуючі інструменти не тільки ніколи не будуть так же тісно інтегруватися з існуючою інфраструктурою як власне рішення, але і можуть просто не надавати бажаний функціонал, особливо якщо це стосується саме отримання нової інформації, а не пошуку помилок.

Також виділені чотири основні підходи до статичного збору метрик та побудови аналізаторів, які в наступному розділі детально розібрані, проаналізовані та продемонстровані на прикладах.

2. Метрики

2.1. Вступ

Не рідко одні й ті самі метрики можна реалізувати за допомогою різних підходів, але рідко коли результат буде однаковим. Розглянемо це на прикладі аналізатора що визначає які бібліотеки використовує модуль, що можна використати для побудови загальної статистики по великій кодовій базі. Отже, основні відмінності полягають в:

- Складність реалізації.

  Так проаналізувати список залежностей заданий в певній схемі в одному файлі (метадані) значно простіше аніж шукати використання цих залежностей в окремих файлах (регулярні вирази або АСД). Але інколи менш простий інструмент може приводити до більш складної реалізації, до чого повернемося пізніше.

- Використання ресурсів.

  Знову таки, аналіз одного файлу буде дуже швидким і потребує максимум завантаження цього одного файла у пам’ять. А от будь-який аналіз всіх файлів у проекті вже апріорі буде незрівнянно повільніше, в той час як побудова та обхід АСД по кожному з них ще на розряд повільніше і може вимагати суттєві об’єми пам’яті, в залежності від способу використання.

- Надійність.

  Чим більше підхід покладається на евристики, вторинні признаки та інші спрощення, тим менш надійним є результат. Часто регулярні вирази є менш надійними за АСД, але через свою простоту та швидкість є кращим вибором для таких задач де точність не критична або отримати хибні результати майже неможливо. Аналіз метаданих може бути як дуже надійним якщо, наприклад, аналізується конфігурація що є єдиним джерелом правди, так і не

надійним якщо аналізуються вторинні признаки або після-ефекти які не можуть стовідсотково гарантувати причину свого виникнення.

- Потужність.

  Цей пункт зазвичай і є основною причиною вибору того чи іншого підходу, тому що деякі метрики дуже складно або просто неможливо зібрати певними видами аналізу, в той час як іншими це тривіальна задача. Крім того, важливо які саме дані можна дістати певним підходом, бо зазвичай чим більша потужність тим більш детальну інформацію можна отримати. І, повертаючись до попередніх пунктів, якщо підхід недостатньо потужний то реалізація певних метрик з його допомогою буде більш складною в розробці та підтримці, а можливо буде і повільніше.

Однак завжди важливо розуміти що саме є ціллю аналізу та яка інформація потрібна. Повертаючись до прикладу з залежностями, аналіз списку залежностей в конфігураційному файлі є тривіальною задачею, але що якщо вказана залежність насправді не використовується в проекті? Тоді можна використати регулярні вирази щоб знайти використання за назвою, але що якщо залежність в якомусь файлі обертається у функцію чи клас і пере-експортується, але далі насправді не використовується? Тоді можна покластися на якусь евристику або використати АСД та зв’язки між файлами щоб відстежити де саме і як використовується залежність. Це доходить до досить складної імплементації яка все одно не максимально надійна, в той час як можливо було доречніше покластися на евристику на якомусь з попередніх кроків або і взагалі залишитись на першому кроці.

В метричному аналізі важливо постійно підтримувати баланс між визначеними вище аспектами і думати ширше, шукаючи простіші, надійніші і водночас ефективні способи вирішення саме поставленої задачі. Далі розглянемо детальніше кожний підхід з прикладами на Node.js та TypeScript, однак це все може бути реалізоване будь-якою мовою.

2.2. Метадані

2.2.1. Вступ

Як було зазначено раніше, метадані це дуже обширне поняття яке включає в себе як найпростіші так і досить складні аналізатори. Через це будемо розглядати методики та приклади у порядку зростання складності.

2.2.2. Перевірка наявності файлу

Одним з найпростіших варіантів використання метаданих є перевірка наявності якогось важливого файлу. Часто такі файли як README, LICENSE, gitignore, pom.xml, package.json та інші конфігураційні елементи знаходяться в корені проекту, через що доступ до них за наявності доступу до проекту є тривіальним і навіть може не потребувати завантаження всього проекту, якщо є можливість зробити запит по такий файл окремо. Однак якщо проект все одно потрібно завантажувати для інших аналізаторів то має сенс вже працювати з завантаженими даними а не робити зайвий мережевий запит.

Прикладом такого простого але корисного аналізатора є перевірка чи існує в проекті README.md, який зазвичай використовується для зберігання основної інформації про проект для розробників:

```
import fs from 'fs';

const readmeFilePresent = (path: string): boolean =>
  [`${path}/README.md`, `${path}/readme.md`, `${path}/Readme.md`].some(
    fs.existsSync,
  );
```

Рисунок 2.1 - Реалізація аналізатора перевірки наявності README

В цьому розділі аналізатор це функція що приймає на вхід шлях до кореневої папки проекту (перед цим завантаженого) та повертає якийсь

результат, в цьому випадку бульове значення. В наступному розділі (присвяченому архітектурі реальної системи для статичного метричного аналізу проектів) ця структура буде ускладнена.

Отже, цей простий аналізатор за допомогою файлової системи (`fs`) перевіряє наявність принаймні одного з варіантів файлів. Ця реалізація проста, працює швидко (враховуючи використані технології) та є досить надійною, хоча і покладається на евристику, що не буде такого файлу з нестандартною назвою, наприклад “ReadMe”.

Можна сказати що цей аналізатор не є дуже корисним, так як сама наявність файлу нічого не говорить про його зміст, тому і порожній файл, і той що має хибну інформацію пройде аналіз. Однак, сама присутність такої метрики вже покаже розробникам що від них очікується і підштовхне до створення доречного README.

2.2.3. Аналіз вмісту файлу з метаданими

Логічним продовженням є перехід від перевірки наявності файлу до перевірки його вмісту. Якщо у файлу є задана стандартна структура, це значно спрощує задачу. Так для парсингу json та xml файлів вже існує безліч інструментів і працювати з їх представленням як з об’єктами значно простіше ніж з оригінальним текстом. Такі інструменти можна знайти і для інших розширень, наприклад того же md, що дозволить просто перевірити те що файл README.md відповідає певній структурі. Але чим складнішим може бути вміст файлу, тим складніше та повільніше буде й інструмент для його обробки, тому до аналізу файлів з програмним кодом ми повернемося в підрозділі про регулярні вирази.

Корисною метрикою та сигналом для розробників може бути перевірка на те як налаштований компілятор або інший інструмент. В контексті TypeScript, це може бути перевірка не те чи увімкнений strict режим, що значно покращує типо-безпеку:

```
const TSStrict = (path: string): boolean =>
  readFile(`${path}/tsconfig.json`).compilerOptions.strict === true;
```

Рисунок 2.2 - Перша реалізація аналізатора перевірки на strict режим

В цій першій наївній реалізації ми зчитуємо конфігурацію в корені проекту та згідно до схеми конфігураційного файлу [7] перевіряємо чи стоїть прапорець `strict`. Реалізація допоміжної функції `readFile` наведена в Додатку А.

Однак цей аналізатор це дуже гарний приклад того що в метричному аналізі треба багато і інколи нестандартно думати про corner case'и і розуміти предметну область. Такою наївною реалізацією можна покрити як мало так і багато випадків, в залежності від конкретного сценарію використання.

Важливо враховувати що TypeScript став популярним набагато пізніше ніж люди почали писати проекти на JavaScript, тому у великій кодовій базі не всі модулі можуть бути розроблені з TypeScript. Отже треба врахувати що конфігураційного файлу може просто не бути. Водночас, може бути не доречним відображати такий варіант як негативний, бо якщо проект взагалі не використовує певну технологію то можливо у розробників є вагомі причини на це і їм не сподобається отримувати негативний відгук про проект через це. Тому можна з двійкової логіки перейти у трійкову і ввести поняття недоречного (irrelevant) результату, який і використовувати у таких випадках.

Але якщо файл і є, в ньому може не бути `compilerOptions`, адже конфігурація може розширювати іншу і нічого більше:

```
{
 "extends": "./tsconfig.base.json"
}
```

Рисунок 2.3 - Приклад TypeScript конфігурації без `compilerOptions`

Однак, цей варіант, на відміну від попереднього, не обов'язково веде до недоречного результату, адже батьківська конфігурація може вмикати strict режим, отже треба перевіряти і її. Але тут додається ще більше складностей:

- Батьківська конфігурація може бути не локальна, а встановлюватись з залежностями, отже треба ще ідентифікувати та встановити ту залежність.
- TypeScript дозволяє не вказувати розширення json при визначенні батьківської конфігурації, тому треба його додавати коли потрібно.
- Батьківська конфігурація може мати strict увімкненим, але потім в дочірній конфігурації його можна вимкнути.

Таким чином від задачі що могла здатись тривіальною ми прийшли до досить комплексної і невідомо скільки ще corner case'ів не покрито. Фінальна реалізація аналізатору наведена в Додатку Б.

Гарним варіантом перевірити чи все було передбачено в реалізації аналізатора є просто "прогнати" його на множині проектів, бажано різних. Зазвичай практично неможливо передбачити всі можливі нюанси, а такий варіант точно покаже наскільки аналізатор працює принаймні в поточному стані кодової бази. Надалі, якщо система активно використовується, користувачі можуть повідомити про те що якийсь аналізатор перестав коректно працювати, наприклад після того як проекти перейшли на нову версію якоїсь бібліотеки, але краще до такого не доводити, а тримати руку на пульсі кодової бази та слідкувати за змінами та трендами.

2.2.4. Перевірка правил вже заданих в інших інструментах

Інколи якась метрика або правило вже задані в іншому інструменті, наприклад лінтері, і важливо знати які модулі підтримують код в задовільному стані по цій метриці, що має бути можливість побачити в системі метричного аналізу. Тоді є три основні шляхи:

- З нуля реалізувати цю метрику.

  Складний та не дуже надійний варіант, так як реалізації в інших інструментах зазвичай вже гарно протестовані і оптимізовані. Однак це найбільш гнучкий варіант і інколи він єдиний, якщо не підходять наступні.
- “Проганяти” той же інструмент та використовувати результати.

  Якщо інструмент надає підходяще API для виконання та отримання результатів, це може бути найкращий варіант, так як він дозволяє максимально перевикористати код, в ідеалі, не втративши якості результату. Якщо ж інструмент недостатньо гнучкий для такого, цей варіант відпадає.
- Перевіряти використання цього інструменту в проектах.

  Якщо ми знаємо що інструмент вже перевіряє те що нам потрібно, можна просто перевірити те що він використовується саме так як потрібно. Цей підхід дуже схожий на попередній приклад з аналізом вмісту одного файлу і є відносно простим і надійним і дуже швидким, але значно менш потужним за інші варіанти, так як він не зможе надати саме звіт по аналізу, наприклад, скільки порушень правила є в проекті і де.

Отже, знову вибір залежить від конкретних вимог. Інколи справді достатньо третього варіанту і так як він фактично є окремим аналізатором, його і розглянемо детальніше на прикладі правил лінтера.

Зазвичай в лінтерах дуже багато правил і нерідко велика частина з них є стилістичними, а кожний розробник полюбляє свій стиль коду.

Можна визначити множину правил які очікуються що будуть увімкнені на кожному проекті і це перевіряти, але зазвичай лінтери дозволяють наслідувати правила з іншої конфігурації. Отже можна створити одну - або декілька, якщо необхідно - конфігурацій, які будуть наслідувати всі інші проекти. Таким чином, якщо потрібно буде додати, прибрати або налаштувати якесь правило, це можна буде зробити по всій кодовій базі в одному місці. Звичайно, в такому випадку модулям потрібен буде час на те щоб привести код до очікуваного стану, тому вони можуть вимикати певні правила на певний час, що можна віднести до технічного боргу.

Базуючись на цьому, можна перевіряти не всі правила окремо, а, по-перше, те що конфігурація в модулі наслідує певну стандартну і, по-друге, скільки правил відключені. Реалізацію такого аналізатору на прикладі TSLint можна знайти в Додатку В. Основна ідея така:

- Задається список конфігурацій які очікується що конфігурація модуля буде наслідувати;
- Зчитується власне конфігурація модуля. Якщо її немає, можна це вважати як за негативний результат, так і за недоречний;
- Перевіряється наявність кожної з очікуваних батьківських конфігурацій;
- Підраховується кількість вимкнених правил.

Досить просто, але тут не врахований важливий момент - в дочірній конфігурації можна не вимкнути правило, а просто змінити його налаштування так щоб воно вже перевіряло не зовсім те що задавалося в батьківській конфігурації. Отже, як розширення цього аналізатору, можна було б ще перевіряти і цей аспект, але більшість випадків покрито і наведеною реалізацією.

2.2.5. Популярність залежностей

У великих кодових базах, де в навіть схожих модулях можуть використовуватись різні технології та підходи, може виникнути питання - а що ж використовувати? Тобто хотілося б розуміти які технології де використовуються і що популярне в компанії.

Інша ситуація - інфраструктурна команда створила новий інструмент і представила розробникам. Всім сподобалося, але через час не зрозуміло чи хтось насправді використовує його. Тут можна зробити просто і імплементувати бінарну перевірку на те чи використовується ця залежність. Однак, можуть з’явитися ще такі інструменти і якщо взяти до уваги першу згадану ситуацію, може мати сенс створити аналізатор що формував би статистику по всім залежностям по всім модулям.

Таким чином, при продуманій візуалізації, до якої повернемося в третьому розділі, можна зробити інструмент в якому буде просто та зручно подивитись які технології користуються популярністю у розробників і які навпаки мало хто використовує.

Реалізацію такого аналізатору можна знайти в Додатку Г. Логіка досить проста, але це гарна демонстрація того що аналізатори можна робити не лише по формулі “Проект - статус”, а і використовувати те знання що модулі існують не у вакуумі та агрегувати дані. Цей аналізатор сильно покладається на гарну візуалізацію, але, загалом, має великий потенціал для розширення та додавання нових елементів. Наприклад, можна використовувати один колір для технологій що зазвичай йдуть разом, таких як комбінація Redux + react-redux + redux-thunk, що дозволить краще бачити напрями, а не лише окремі технології.

2.2.6. Версії залежностей

Залежності проекту це широка тема, по якій можна придумати не одну потенційно корисну метрику і далі ми ще розглянемо декілька таких.

Наприклад, нерідко в організаціях є певний стандарт по тому які версії технологій мають використовуватись розробниками. Цей стандарт може визначатись обмеженнями пов’язаними з іншими технологіями або цільовими платформами (тобто може вимагатись використання не останніх версій), або навпаки недоліками попередніх версій в напрямку безпеки, швидкодії, тощо (тобто старі версії не рекомендуються до використання).

Виходячи з цього, може бути корисним бачити постійно актуальний звіт по тому які версії яких технологій використовуються в яких проектах. Це дозволяє:

- Явно показати розробникам що від них очікується;
- Побачити високорівневу ситуацію по всій кодовій базі;
- Відстежити окремі модулі що відклоняються від норми та поспілкуватись з розробниками щоб зрозуміти причини.

Якщо говорити про версії залежностей, зазвичай їх задають в одному з двох форматів:

1. Просто числом - простий варіант, який легко перевірити на відповідність очікуванням, але в той же час не дуже гнучкий, якщо, на приклад, хочеться задати якусь множину доступних версій;
2. Semantic versioning - дуже гнучкий формат який включає багато варіантів задання версій, наприклад “~1.2.3”, що включає всі версії від 1.2.3 до (але не включаючи) 1.3.0 [8], або “>=1.2.3 <1.3.0 || =1.3.2”, що знову включає всі версії від 1.2.3 до 1.3.0, не включаючи 1.3.0, але ще додає 1.3.2. Через потужність та популярність цього підходу його і розглянемо далі.

Реалізацію аналізатора версій залежностей наведено в Додатку Д. Ідея така:

1. Задається асоціативний масив залежностей та їх очікуваних версій (у форматі SemVer, що дозволяє задати множину значень);

2. Для кожного проекту з його конфігурації дістаються всі залежності з версіями;
3. По кожній з затребуваних залежностей для кожного проекту перевіряється чи максимальна версія залежності що може бути встановлена за визначеною версію в конфігурації проекта входить у множину очікуваних версій залежності. Таким чином, якщо остання версія залежності 5.0, очікується множина >=4 і в проекті вказана множина >2, проект пройде перевірку, адже 5.0 підходить під умову >2 і одночасно входить в множину >=4.

Цей аналізатор вже може вимагати роботи із сторонніми сервісами для того щоб отримати список версій що відповідають умовам. особливо це може ускладнюватись якщо є бажання перевіряти не лише прості залежності вказані в загальній конфігурації а і ще щось. Тоді, з великою ймовірністю, доведеться окремо обробляти ці випадки і, можливо, покладатися на евристики.

Наприклад, в JavaScript проектах може бути корисним контролювати версії Node.js в проектах, тому що в більш нових версіях можуть бути функції що ще не підтримується цільовими платформами, а в більш старих можуть бути проблеми з використанням ресурсів чи безпекою. Однак взагалі в таких проектах ніде не вимагається вказувати версію Node, отже треба, наприклад, покладатися на те що у Вашій організації або команді всі використовують Node Version Manager, який дозволяє додавати файли .nvmrc, в яких вказувати бажану версію Node.js. Тоді можна спробувати зчитати версію з цього файлу, після чого перевірити чи вона відповідає очікуваній. Однак, важливо враховувати те що у Node.js є окремі теги для деяких версій, наприклад LTS для тої що рекомендується для використання та latest для останньої, отже це також треба врахувати. Це все не є великою проблемою, але такі нюанси є у кожної технології і треба їх мати на увазі

та вчасно ідентифікувати. Реалізація аналізатора версії Node.js не наводиться і не розбирається заради стислості.

2.2.7. Застарілі залежності

Схожим але все ж таки відмінним аналізатором є перевірка застарілих залежностей. Для того щоб спонукати розробників оновлювати свої залежності і не страждати від вразливостей, відсутності підтримки, поганої оптимізації та інших проблем, можна для кожного модуля виводити список його застарілих залежностей з пропозицією оновити їх.

Тут можна перевикористати попередній аналізатор для найскладнішої частини але важливо зважити чи потрібна перевірка залежностей що використовуються лише як допомога розробці, наприклад фреймворки для тестування. Зазвичай, підтримання цих залежностей в актуальному стані не так важливо бо вони не впливають на швидкодію та безпеку кінцевих користувачів, тому в наведеній далі реалізації вони опущені:

```
import _ from 'lodash';
import latestVersion from 'latest-version';

async function outdatedDependencies(path: string) {
 const packageJson = readFile(`${path}/package.json`);
 if (!packageJson?.dependencies)
   return { status: Status.IRRELEVANT };

 const dependencies = await Promise.all(
   _.map(_.keys(packageJson.dependencies), async (name) => {
     return { name, wantedVersion: await latestVersion(name) };
   }),
 );

 const result = _.first(
   await analyzeVersion(
     [{ path, packageJsonPath: path, name: packageJson.name }],
     { dependencies },
   ),
 );

 const packageVersionDiffs = _.reduce(
   result,
   (acc, { status, text, targetValue }, name) => {
     if (status !== Status.GOOD)
       acc.push(`${name} ${text} -> ${targetValue}`);

     return acc;
   },
   [],
 );

 return packageVersionDiffs.length
   ? { status: Status.BAD, text: packageVersionDiffs.join('\n') }
   : { status: Status.GOOD };
}
```

Рисунок 2.4 - Аналізатор застарілих залежностей

2.2.8. Динамічне в статичному

Статичний аналіз хоча і покриває велику кількість аспектів програмних продуктів, все ж не всесильний і деякі важливі напрями йому недоступні, чому і використовують також динамічний аналіз. Прикладами є заміри використання ресурсів та швидкодії, відсоток покриття тестами та власне тестування коректності, тощо. Статичним аналізом такі речі можна виміряти зазвичай лише емпірично, але статичний аналіз метаданих інколи все ж може допомогти. Справа в тому, що не завжди обов'язково надавати нову інформацію користувачам саме у вигляді результату власного аналізатора - інколи достатньо просто правильно відобразити вже відому інформацію, розкидану по багатьом інструментам, в одному місці для побудови загальної картини.

Отже, якщо у нас вже налаштоване потужне CI середовище що виконує тести, збирає інформацію по покриттю коду, швидкодії, тощо, і це середовище надає можливість отримати ці результати з іншого застосунку, можна побудувати аналізатор що буде просто збирати ці дані, парсити та зручно відображати. Може виникнути запитання - "навіщо це потрібно якщо і так ця інформація вже доступна", однак дуже важливо те, що вона розкидана по логам збірок кожного проекту і на практиці нікому не цікаво їх досліджувати щоб зрозуміти в якому стані знаходиться кодова база. Набагато зручніше побачити витримку у вигляді таблиці чи іншого звіту з конкретними числами по кожному модулю.

Приклад такого аналізатору, що збирає кількість попереджень (warnings) та відсоток покриття тестами, можна побачити в Додатку Е. Хоча він і покладається на досить багато евристик, найважливіша частина все ж залишиться одна, незалежно від інструменту для збірки та тестування.

Цей аналізатор складається з двох основних частин, хоча може бути розширений чи звужений до бажаної кількості:

1. Кількість попереджень.

   Для цього може бути достатньо власне тексту збірки, який буде проаналізований на певний патерн, який вже залежить від конкретних інструментів, тому тут не наводиться.

2. Відсоток покриття коду.

   Ця частина покладається на те що в проекті взагалі увімкнено збирання покриття. Якщо це так, то далі ця задача, як і попередня, зводиться до парсингу отриманого логу збірки. Такий вид задач досить швидко вирішується регулярними виразами, які ми детально розглянемо в наступному розділі.

2.3. Регулярні вирази

2.3.1. Вступ

Регулярні вирази це послідовність символів що описують певний патерн в тексті. Наприклад, патерн \b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b можна використати як відносно наївний опис електронної адреси [9]. Загалом, вони дозволяють коротко і декларативно описати навіть дуже складні патерни в тексті, що приходить до нагоди в аналізаторах що мають щось шукати у вмістах файлів, наприклад коді.

Як аналіз метаданих може включати використання регулярних виразів, так і аналізатор оснований на регулярних виразах може включати якесь використання метаданих. Так само в наступних розділах основні інструменти можуть використовувати інші, але все ж вони знаходяться там де є через те який підхід є основою аналізатора.

2.3.2. Пошук патернів в коді

Одним з найпростіших способів використати регулярні вирази в підрахунку метрик є пошук певних рядків у файлах. Якщо рядок або рядки завжди однакові і їх не можна сплутати з чимось іншим, то взагалі регулярні вирази можуть бути і не потрібні, бо можна просто виконати пошук підрядка у файлі. Однак зазвичай потрібно знайти більш складний патерн ніж унікальний рядок і тут вже корисні регулярні вирази.

Можна привести дуже багато прикладів того що можна шукати. Цей підхід простий, відносно гнучкий і швидкий, і в залежності від середовища, технологій та потреб можна придумати багато використань. Наприклад, деякі інші технології для статичного аналізу дозволяють додавати коментарі в коді щоб наступний рядок, блок, або весь файл не проходив аналіз на якесь правило, або взагалі. Хоча інколи це справді

необхідно, зазвичай використання таких мір каже про недосвідченість розробника з певними технологіями, або небажання витрачати час на вирішення проблеми у момент написання коду, що фактично створює технічний борг. Отже, можна підрахувати кількість таких ігнорувань правил в коді модуля.

Іншим прикладом є підрахунок кількості TODO та схожих за семантикою коментарів:

```
async function todosCounter(path: string): Promise<AnalysisResult> {
 const todoRegExp = /( |\/\/)(TODO|FIXME)( .*)?$/gim;

 const count = countRegExpInFiles(await getCodeFiles(path), todoRegExp);

 return { text: count, status: count === 0 ? Status.GOOD : Status.BAD };
}

interface AnalysisResult {
 status: Status;
 text?: number | string;
}

enum Status {
 GOOD = 'GOOD',
 BAD = 'BAD',
 MIXED = 'MIXED',
 IRRELEVANT = 'IRRELEVANT',
}
```

Рисунок 2.5 - Аналізатор кількості TODO в проекті

Як видно, у всіх файлах з кодом в проекті йде пошук такого патерну:

1. Пробіл або `//` (початок коментара в JavaScript, для інших мов може виглядати інакше);
2. TODO або FIXME;
3. Опціональні пробіл та будь-яка послідовність символів після нього;
4. Кінець рядка.

Варто також згадати про встановлені прапорці:

- `g` - пошук всіх підходящий підрядків, не лише першого;
- `i` - регістр символів ігнорується;
- `m` - так як весь вміст файлу зберігається в одному рядку, патерн `$` відповідав би кінцю файла. Однак з цим прапорцем він відповідає кінцю рядка, тобто послідовності символів що закінчуються символом \n.

Такий патерн дозволяє шукати різні способи використання TODO, водночас відсіюючі рядки на кшталт “todosCounter”, що може бути назвою функції. Реалізація допоміжних функцій `getCodeFiles` та `countRegExpInFiles` наводиться в Додатку Ж.

2.3.3. Детальні звіти

Однак інколи одного числа проблем в проекті недостатньо, а є потреба в більш детальному звіті по ситуації. Наприклад, може бути корисно побачити які саме TODO стоять в проекті, де вони знаходяться і хто і коли востаннє їх оновлював. Вся ця інформація може допомогти побачити корисні патерни і визначити вектори руху для покращення проекту:

```
async function todoReport(projects: Project[]) {
 return (await Promise.all(projects.map(analyzeProject)))
   .flat()
   .filter(notUndefined)
   .filter(
     (line, _index, array) =>
       !array.some(({ filePath }) => filePath.indexOf(line.filePath) > 0),
   );
}
```

```
async function analyzeProject({ path }: Project) {
 const regexp = `[ //](TODO|FIXME)[(: _-]`;
 const targetFiles = await gitGrepByRegex(path, regexp, true);

 return Promise.all(
   targetFiles.map(async ({ filePath, lineNumber }) => {
     const blameResult = await gitBlameLine(path, filePath, lineNumber);
     if (!blameResult) {
       return;
     }

     const todoIndex = blameResult.code.search(/(todo|fixme)/i);

     return {
       filePath,
       lineNumber,
       columnNumber: String(todoIndex + 1),
       codeWithTodo: blameResult.code.substring(todoIndex).trim(),
       lastChangedBy: blameResult.lastChangedBy,
       lastChangedDate: blameResult.lastChangedDate,
     };
   }),
 );
}
```

Рисунок 2.6 - Звіт по TODO в проекті

В цьому аналізаторі для кожного проекту виконується такий алгоритм:

1. За допомогою git та регулярних виразів шукаються всі файли що включають TODO або FIXME коментарі;
2. В цих файлах для цих рядків, знову за допомогою git, отримується інформація про те коли та ким була зроблена остання зміна цього рядка;
3. Визначаються конкретні колонки та вміст TODO коментарів.

Реалізація допоміжних функцій `notUndefined`, `gitGrepByRegex` та `gitBlameLine` наведена в Додатку И. Git є одним з дуже потужних

інструментів, які можна використовувати для складних метрик, однак конкретно з git варто пам’ятати, що зміни які нас цікавлять можуть бути і не в останньому коміті, і не в десятку останніх, тому доведеться завантажувати всю історію, що набагато повільніше ніж використовувати лише останній коміт, чого вистачає для більшості аналізаторів.

2.3.4. Використання технологій

Раніше в роботі вже було розглянуто аналізатор популярності залежностей, але його проблема в тому, що простий аналіз списку залежностей не гарантує те що ті залежності справді використовуються в проекті. Залежності можуть бути у конфігураціях по різним причинам і якщо потрібно дізнатись що насправді використовується, доводиться звертатись як мінімум до регулярних виразів.

Розглянемо приклад з інструментами для керування станом клієнтських застосунків:

```
import { compact, sortBy, uniq } from 'lodash';

async function stateManagementUsage(path: string) {
 const libraries = await getUsedStateManagementLibraries(path);

 return libraries.length === 0
   ? { status: Status.IRRELEVANT }
   : {
       text: libraries.join('\n'),
       status: libraries.length === 1 ? Status.GOOD : Status.BAD,
     };
}
```

```
function getUsedStateManagementLibraries(path: string) {
 return Promise.all(
   knownStateManagementLibs.map(async ([regexp, lib]) =>
     countRegExpInFiles(await getCodeFiles(`${path}/src`), regexp) === 0
       ? undefined
       : lib,
   ),
 )
   .then(compact)
   .then(uniq)
   .then((libs) => sortBy(libs));
}

const makeLibRegex = (name: string) =>
 new RegExp(`from ['"]${name}['"]|require\\(['"]${name}['"]\\)`, 'g');

const knownStateManagementLibs: [RegExp, string][] = [
 [makeLibRegex('remx'), 'remx'],
 [makeLibRegex('redux'), 'redux'],
 [makeLibRegex('@reduxjs/toolkit'), 'redux'],
 [makeLibRegex('react-redux'), 'redux'],
 [makeLibRegex('mobx'), 'mobx'],
 [makeLibRegex('mobx-state-tree'), 'mobx-state-tree'],
];
```

Рисунок 2.7 - Аналізатор використання клієнтських інструментів для керування станом

Спочатку будується співвідношення “бібліотека -> регулярні вирази”. Для одного інструменту може бути декілька виразів, тому що буває що інструмент використовується не напряму, а через іншу бібліотеку, через що важливо перевіряти як пряме імпортування, так і використання інших бібліотек, що свідчить про непряме використання інструменту.

Після чого, по кожному кодовому файлу проганяється кожний з регулярних виразів і збирається інформація які інструменти використовуються в проекті. Очікується що на одному проекті буде використовуватись лише один такий інструмент.

Можна звернути увагу на те, що аналізуються файли лише в директорії `src`. Ця евристика додана для того щоб враховувати справжній код, а не якісь допоміжні скрипти, файли для тестів, тощо. Це не є обов'язковою частиною, але в залежності від конкретної кодової бази такий прийом може бути доречним.

2.3.4. Невикористання чогось

Хоча часто аналізатори перевіряють проект на наявність якихось шкідливих аспектів, інколи доречно пошукати відсутність корисних. Наприклад, застосунки зазвичай перекладають на десятки мов, нехай M, отже кожен з N рядків тексту в проекті має бути перекладений, що призводить до N * M рядків. Але що якщо K рядків вже не використовуються в проекті? Можливо код який їх включав видалили, або з самого початку їх додали випадково. Тоді виходить, що користувачі отримують M*K рядків абсолютно зайвого тексту, який могли б не завантажувати. Крім того, це може означати помилку в застосунку, бо можливо рядки додали але забули використати.

Зазвичай всі переклади задаються в одній папці в різних файлах і кожний рядок має унікальний ідентифікатор, тому перевірити чи всі рядки використовуються не складно:

1. Зчитати список ідентифікаторів рядків;
2. Отримати всі кодові файли в проекті;
3. Для кожного ідентифікатора перевірити чи знаходиться він принаймні в одному файлі.

Через те що рядки унікальні, тут навіть не потрібні регулярні вирази - достатньо просто пошуку підрядка в рядку (файлі).

Однак ассети, такі як медіа файли (наприклад зображення, відео, звукові записи та шрифти), не обов'язково мають унікальний ідентифікатор і також можуть бути додані але не використані. Тут вже

потрібне правильне використання регулярних виразів, що продемонстровано в Додатку К.

Цей аналізатор є прикладом не тільки складного але все ж підходящого завдання для регулярних виразів, але і того що завжди при розробці таких систем важливо тримати в голові предметну область та конкретну кодову базу для якої інструмент розробляється:

- Ассети можуть підключатися різними способами і з різних місць в проекті, тому потрібно перевіряти всі можливі шляхи. Так, наприклад, якщо зображення лежить за шляхом “src/assets/img.png”, воно може бути підключено наступними способами: “./src/assets/img.png”, “./assets/img.png”, “./img.png”. Крім того, шлях до всіх проміжних папок може включати виходи на рівень вище “../”, що також треба враховувати.
- В деяких середовищах, наприклад мобільних застосунках, в проекті може бути декілька версій одного зображення для дисплеїв різної роздільної здатності. Тоді назви таких файлів можуть бути у форматі “img@2x.png”, де “@2x” це суфікс який не використовується при підключенні зображення в коді, але вказує на яких дисплеях використовувати саме цей варіант. Для аналізу важливо відкидати ці суфікси.
- Також в деяких середовищах при збірці проекту є можливість підключати різні версії файлів в залежності від додаткових розширень файлів. Наприклад, в React Native застосунках це могла бути платформа - img.android.png, img.ios.png, тощо. Отже, потрібно ігнорувати такі розширення.

В аналізаторі спочатку збираються та нормалізуються всі ассети, а потім в кожному кодовому файлі проекту за допомогою регулярних виразів шукаються ці ассети. Важливо зауважити, що може бути доречним перевіряти не лише кодові файли, а взагалі всі, тому що деякі ассети

можуть використовуватись, наприклад, в документації. Також, в різних середовищах та з різними технологіями можуть бути й інші способи приховати від аналізатора що файл справді використовується - важливо їх ідентифікувати та адаптувати рішення, але не варто намагатись передбачити все завчасно, краще робити ітеративні тести.

2.3.5. Вкладені патерни

Хоча регулярні вирази та парсинг тексту це потужні інструменти, вони не всесильні і деякі аналізатори реалізовувати за допомогою них може бути дуже складно або зовсім неможливо.

Прикладом може бути аналізатор пропущених тестів в проекті. Тут складність заключається в тому що в сучасних фреймворках для тестування може бути багато різних способів пропустити тест або, що особливо важливо, групу тестів. Наприклад, Jest - фреймворк для тестування JavaScript застосувань - дозволяє пропускати окремі тести, пропускати блоки тестів, а також навпаки позначати що лише певні тести з блоку треба виконувати. Отже, аналізатор має максимально враховувати різні способи пропуску, а також шукати не тільки сам тест, а ще й бажано ідентифікувати його назву:

```
import path from 'path';

const analyzeSkippedTests = async (projects: Project[]) =>
 (await Promise.all(projects.map(analyzeProject))).flat();

const analyzeProject = async (project: Project) => {
 const matches = await gitGrepByRegex(
   project.path,
   '(^|\\s)((it|test|describe).skip\\(|(xit|xtest|xdescribe)\\()',
 );

 return Promise.all(matches.map((match) => analyzeGrepMatch(project, match)));
};

const analyzeGrepMatch = async (
 project: Project,
 { filePath, lineNumber }: GitGrepMatch,
) => {
 const fileContent = readFile(path.join(project.path, filePath), false);
 const blameResult = await gitBlameLine(project.path, filePath, lineNumber);

 return {
   filePath,
   lineNumber,
   testName: getTestTitle(fileContent, Number(lineNumber) - 1),
   lastChangedBy: blameResult?.lastChangedBy,
   lastChangedDate: blameResult?.lastChangedDate,
 };
};

const getTestTitle = (fileContent: string, lineNumber: number) => {
 const skip = fileContent.split('\n', lineNumber).join('\n').length;
 const testTitleRegexp =
/(it|test|describe|xit|xdescribe)(\.skip|)\([\s]*['"`]{1}(.+?)['"`]{1}/gim;

 return testTitleRegexp.exec(fileContent.substr(skip))?.[3] ?? '';
};
```

Рисунок 2.8 - Звіт по пропущеним тестам в проекті

Тут знову використовується потужність git та регулярних виразів щоб зібрати всі файли в проекті що мають пропущені тести або блоки. Після цього, отримується інформація про останні зміни та власне назва тесту, для якої вже використовується досить складний регулярний вираз в функції `getTestTitle`. В цій реалізації взагалі не враховані ситуації коли для виконання виділені лише окремі тести, а інші неявно стають пропущеними. Крім того, якби ми хотіли отримати точну кількість пропущених тестів, ця реалізація б не підійшла ще й тому що цілі пропущені блоки вважаються за один тест, але треба було б рахувати тести в цих блоках, та рекурсивно у вкладених блоках.

Якщо нам потрібна така точна інформація, можна або знову використовувати аналіз збірки, тобто результатів динамічного аналізу, або спробувати використати інструмент потужніший за регулярні вирази - абстрактні синтаксичні дерева.

2.4. Абстрактні синтаксичні дерева

2.4.1. Вступ

Абстрактні синтаксичні дерева, або скорочено АСД, це деревовидні представлення синтаксичної структури програмного коду. Вузли дерева це елементи програми і кожний з них може мати різні важливі властивості.

Розглянемо наступну реалізацію Евклідового алгоритму як приклад:

```
while b ≠ 0
 if a > b
   a := a − b
 else
   b := b − a
return a
```

Рисунок 2.9 - Псевдокод для алгоритму Евкліда

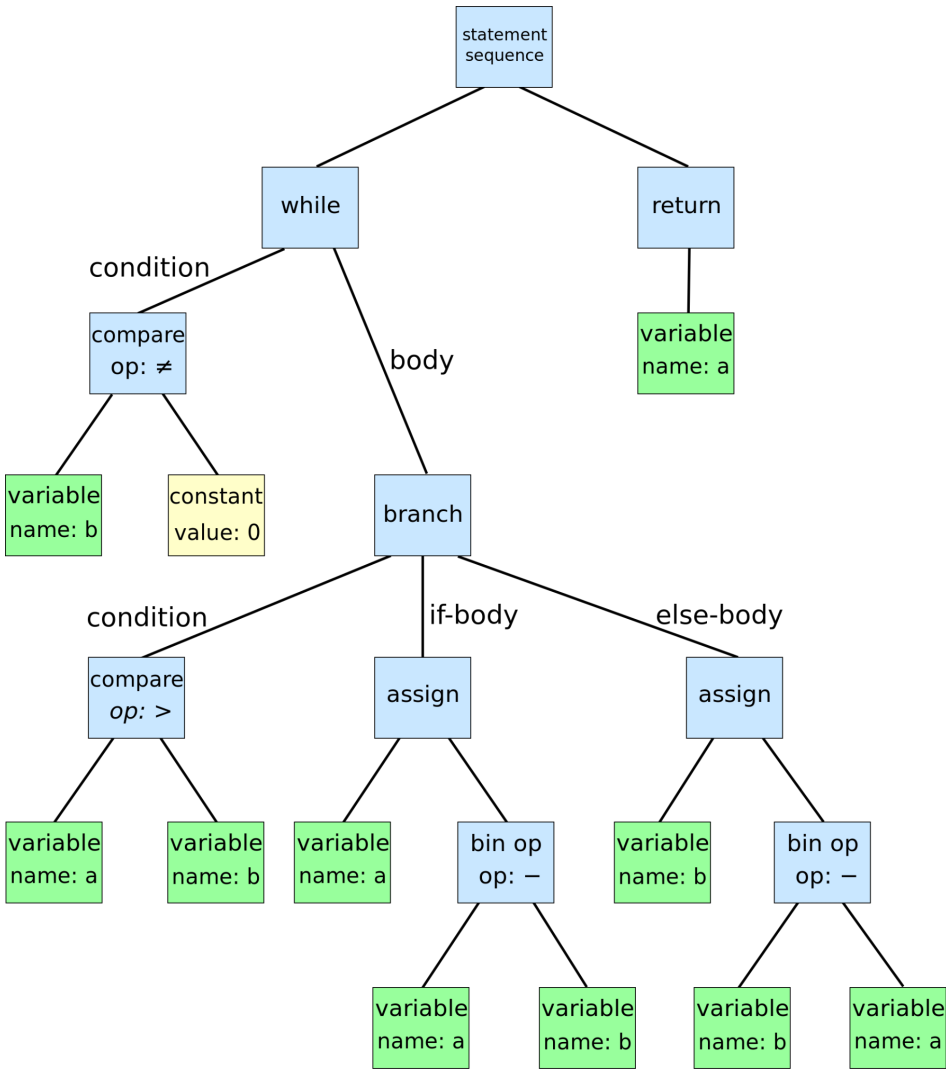

Рисунок 2.10 - АСД псевдокоду алгоритму Евкліда [10]

АСД зазвичай будуються для блоку коду або цілого файлу і є основним інструментом для представлення та роботи з програмним кодом таких інструментів як компілятори, транспілятори, лінтери, пакувальники, статичні аналізатори та інших. Це представлення є одночасно зручним в роботі та дуже потужним, тому що дозволяє використати весь функціонал сучасних мов програмування для роботи з кодом як з програмним об’єктом. Водночас, побудова та робота з АСД є на розряд повільнішою за простий парсинг тексту з регулярними виразами, тому якщо можна щось надійно зробити без АСД, краще піти цим шляхом.

Хоча такі дерева можна обходити і вручну, не рідко інструменти що будують АСД надають зручний спосіб їх обходу - патерн Відвідувач (Visitor). Розробник вказує лише які види вузлів йому цікаві і яку логіку виконувати для них. Тоді, при обході дерева, кожного разу як будуть зустрічатися вузли певних типів, буде виконуватися ця логіка. Однак не завжди різні інструменти будуть видавати однакове дерево для одного коду. Навіть для мов для яких визначений стандарт АСД (наприклад ESTree для EcmaScript [11]) різні, навіть дуже популярні інструменти можуть мати суттєві відмінності. Отже, вибір інструменту для побудови АСД є кроком який може значно спростити чи ускладнити всю подальшу розробку. Всі наступні приклади використовують Babel, який гарно підходив для роботи з EcmaScript, реалізаціями якого є JavaScript та TypeScript.

2.4.2. Кількість елементів

Так як Babel надає згаданий вище патерн Відвідувач, найпростішим його використанням може бути підрахунок кількості певних елементів в коді. Однак в чому проблема їх порахувати регулярними виразами?

Розглянемо приклад з оператором ! з мови TypeScript. Як і в багатьох інших мовах програмування, він зазвичай виступає як булеве “ні”, однак точно так само виглядає оператор ствердження що об’єкт після якого він стоїть не є невизначеним, через що і його назва - non null assertion operator. Отже, якщо з регулярними виразами теоретично і можна відрізнити ці оператори базуючись на певних евристиках, набагто надійніше використати Абстрактні синтаксичні дерева, тому що в них ці два оператори явно є вузлами різних типів:

```
import traverse from '@babel/traverse';
import _ from 'lodash';

async function nonNullAssertionCount(path: string): Promise<AnalysisResult> {
 const files = await getCodeFiles(path, 'ts|tsx');
 if (files.length === 0) {
   return { status: Status.IRRELEVANT };
 }

 const count = _.sumBy(files, countAssertionsInFile);

 return {
   text: count,
   status: count === 0 ? Status.GOOD : Status.BAD,
 };
}

function countAssertionsInFile(path: string): number {
 let count = 0;

 traverse(AST.parseFile(path), {
   TSNonNullExpression() {
     ++count;
   },
 });

 return count;
}
```

Рисунок 2.11 - Аналізатор кількості операторів `!` в TypeScript

Реалізація допоміжної функції `AST.parseFile` наведена в Додатку Л.

В даній імплементації для кожного кодового файлу в проекті (в цьому випадку для всіх TypeScript файлів) виконується такий алгоритм:

1. Зчитати код файлу як текст;
2. Розпарсити текст в АСД за допомогою Babel;
3. Обійти дерево, інкрементуючи лічильник кожного разу як зустрічається вузол певного типу, в цьому випадку non null assertion;

4. Підсумувати результати по всім файлам та видати результат.

Такий патерн можна перевикористати для підрахунку кількості будь-якого типу вузлів в проекті і він може надійно приносити досить корисну інформацію при дуже низькій складності реалізації. В ідеалі також перевикористати побудовані АСД між аналізаторами, адже код не мав би змінитися, а от побудова дерев це досить повільна операція.

2.4.3. Статистичний аналіз

Однак вузли АСД це далеко не лише їх типи, а набагато більше, адже вони також мають багато корисних властивостей, наприклад ідентифікатор (назва функції, змінної, класу, тощо), номер рядків та стовпчиків початку та кінця блоку, та багато іншого.

Цю інформацію можна спробувати використати щоб зібрати статистику по проектам, наприклад, середню довжину функцій та класів. Однак, хоча в теорії це звучить як корисна метрика, на практиці треба бути дуже уважним до середовища та технологій проектів. Якщо мова йде про EcmaScript, то там в проектах зазвичай наявно дуже багато функцій довжиною від одного до трьох рядків. Через це середня довжина всіх функцій та класів навіть в проектах сумнозвісних своїми великими класами та функціями буде зазвичай не більше 20, а частіше взагалі біля 10 рядків. Через це більш корисним може бути такий сценарій:

1. Зібрати інформацію по довжинам всіх функцій та класів;
2. Надати список цих функцій та класів з назвами, розмірами та бажано посиланнями на місце в репозиторії, щоб відразу побачити що це;
3. Порахувати середні довжини найбільших, наприклад, десяти відсотків;
4. Відобразити загальний графік по всім функціям та класам в проекті та їх довжинам.

Розглянемо аналізатор що виконує перші два пункти:

```
import { Class, Function } from '@babel/types';
import get from 'lodash/get';
import take from 'lodash/take';

type Details = { name: string; size: number; githubLink: string };

const analyzeFunctionsAndClassesListBySize = (projects: Project[]) =>
 Promise.all(projects.map(analyzeProject));

const analyzeProject = async (project: Project) =>
 (await getCodeFilesWithoutTests(project.path))
   .flatMap(getContext(project))
   .sort((a, b) => b.size - a.size);

const getContext = (project: Project) => (path: string): Details[] => {
 const details: Details[] = [];

 const countLines = ({ node }: { node: Function | Class }): void => {
   const size = AST.getNodeLinesOfCode(node);
   const name = get(node, 'id.name') || get(node, 'key.name') || 'anonymous';
   const githubLink =
     project.packageRemoteUrl +
     path.slice(project.path.length + 1) +
     `#L${node.loc?.start.line}-L${node.loc?.end.line}`;

   details.push({ name, size, githubLink });
 };

 traverse(AST.parseFile(path), { Function: countLines, Class: countLines });

 return details;
};
```

Рисунок 2.12 - Аналізатор довжин функцій та класів

Цього разу аналізуються два типи вузлів, хоча, насправді, `Function` та `Class` включають в себе всі можливі види функцій та класів. З вузла отримується довжина блоку (реалізація в Додатку Л), назва та посилання на GitHub. Конкретно в цьому варіанті повертаються всі функції

відсортовані за спаданням розміру, що надає потім можливість взяти найбільші десять відсотків та порахувати середню довжину, або взяти всі і відобразити на графіку, що може бути особливо корисним для пошуку аномалій при порівнянні різних проектів у великій кодовій базі.

2.4.4. Обхід вручну

Досі ми не виходили за межі патерну Відвідувач, але інколи задачі вимагають аналізу не тільки окремих вузлів а і їх контексту. Якщо інструмент дозволяє, наприклад, отримати батьків та дітей вузла в межах патерну Відвідувач, це вже значно збільшує можливості до реалізації складних аналізаторів. Однак, навіть якщо такої можливості немає, завжди можна обійти дерево повністю вручну, що, звичайно, значно складніше в реалізації.

Прикладом метрики для якої такий підхід може бути корисним є підрахунок кількості пропущених тестів, який був згаданий в попередньому підрозділі. Сучасні фреймворки для тестування дозволяють пропускати окремі тести, блоки тестів, виділяти окремі тести та блоки так щоб виконувались лише вони та інші так виділені, а також робити це з різним синтаксисом. Через це при аналізі вузла важливо враховувати його оточення, тобто інші тести та згруповування, що продемонстровано в реалізації аналізатора в додатку М.

Алгоритм для АСД кожного файлу можна коротко описати так:

1. Для всіх видів тестів інкрементується лічильник тестів;
2. Якщо тест пропущений, інкрементується лічильник пропущених. Те саме для виділених на виключне виконання;
3. Якщо тест звичайний, серед батьків шукається згруповування яке або пропускає всі дочірні тести, або навпаки їх виділяє на виключне виконання;

4. Якщо такий батько знайдений, інкрементується відповідно лічильник пропущених або виділених тестів;
5. Якщо такий батько не знайдений і в файлі є інші тести що позначені на виключне виконання, тест вважається пропущеним;
6. Якщо були знайдені тести що виділені на виключне виконання, результат рахується як різниця між загальною кількістю тестів та кількістю таких виділених;
7. Інакше, результат це кількість пропущених тестів (лічильник).

У поєднанні зручного та просто патерну Відвідувач та потужного ручного обходу, АСД є дуже корисним та багатозадачним інструментом. Однак, при його використанні варто пам’ятати про швидкодію, адже це не сильна сторона цього підходу, особливо при ручному обході. Ще одна вроджена вада АСД це те що вони будуються лише на одний файл, але інколи потрібно аналізувати зв’язки між файлами та сутностями в цих файлах. Для цього (найскладнішого розглянутого в цій роботі) типу задач нам знадобляться графи зв’язків.

2.5. Графи зв’язків

Графи зв’язків можна будувати для різних сутностей: директорій, файлів, класів, функцій, тощо. В залежності від задачі може бути потрібен різний граф, але найпопулярнішим видом буде граф зв’язків файлів. Пакувальники коду зазвичай вже будують такі графи і деякі з них можуть навіть надавати API для використання цих графів користувачами у власних цілях. Наприклад Metro bundler, який використовувався для React Native застосувань, надавав такий функціонал:

```
const IncrementalBundler = require('metro/src/IncrementalBundler');
const loadDefaultConfig =
require('@react-native-community/cli/build/tools/config/index.js').default;
const loadMetroConfig =
require('@react-native-community/cli/build/tools/loadMetroConfig.js').default;

const options = {
 entryFile: 'index.js',
 platform: 'android',
 dev: true,
 minify: false,
};

(async () => {
 const buildGraphRes = await new IncrementalBundler(
   await loadMetroConfig(loadDefaultConfig()),
   options,
 ).buildGraph(options.entryFile, options);

 buildGraphRes.graph.dependencies.forEach((dependency) => {
   console.log('GraphNode:', dependency.path);
   console.log('SourceCode:', dependency.getSource().toString());
   console.log('Dependencies', dependency.dependecies);
   console.log('Inverse dependencies', dependency.inverseDependencies);
 });
})();
```

Рисунок 2.13 - Побудова графу залежностей за допомогою Metro

Пакувальник починає з заданого файлу і збирає граф зв’язків з шляхом, вихідним кодом, залежностями, зворотніми залежностями та іншими корисними атрибутами для кожного файлу.

Такий граф можна використати для будь-якого аналізу зв’язків між файлами в проекті, наприклад, пошуку циклічних залежностей. Більш комплексною задачею є візуалізація залежностей в проекті з метою високорівнево оцінити ситуацію та побачити, наприклад, наскільки тісно пов’язані модулі, чи не перетинаються доменні межі, чи не “протікають” абстракції, тощо.

Якщо ж є бажання отримати те саме але для зв’язків між класами, типами, функціями, або іншими сутностями в проекті, доведеться або шукати інший підходящий інструмент, або реалізовувати це вручну. Взагалому, достатньо мати можливість побудувати, зберегти та проаналізувати АСД по всім файлам в проекті. Тоді можна для кожної сутності зберегти основну інформацію про неї (назву, локацію в проекті, тощо) і її прямі та, за потреби, зворотні залежності. В результаті вийде також граф, лише у вигляді списку суміжності. Код для побудови та аналізу таких структур не наводиться через величезний об’єм та складність.

В реальності зазвичай все утикається в те як зручно та зрозуміло дані будуть представлені кінцевому користувачу, тобто у реалізацію власне системи метричного статичного аналізу проектів.

2.6. Висновки

В цьому розділі були виділені основні критерії оцінювання реалізації аналізатора: складність реалізації, використання ресурсів, надійність та потужність.

Було детально розглянуто чотири раніше виділені підходи до збору метрик та по ним сумарно виділено, описано та проаналізовано вісімнадцять різноманітних аналізаторів, починаючи перевіркою наявності README файлу в проекті, продовжуючи пошуком невикористаних ассетів, та закінчуючи детальним звітом по розмірам функцій та класів.

Аналіз метаданих, регулярні вирази та абстрактні синтаксичні дерева за результатами дослідження було виділені як найбільш корисні та прикладні у реальній розробці, в той час як графи залежностей підходять лише для невеликої підмножини задач, які, однак, ніяким іншим методом не вирішити.

В наступному розділі розглядається реалізація реальної веб системи для статичного метричного аналізу проектів.

3. Архітектура прикладної системи

3.1. Вступ

Ціль використання або побудови системи для метричного статичного аналізу програмних проектів може бути різна в різних компаніях та окремих командах. Але, звичайно, від цілі та конкретних потреб залежать фундаментальні концепти системи, при чому як продуктові так і технічні. Від того які деталі має надавати система, в якому виді, як надійно, як актуально, в якому об'ємі та багатьох інших питань система може бути побудована у абсолютно різний спосіб. Розглянута в цій роботі реалізація такої системи базується на наведених далі принципах:

- Аналіз має відбуватись відразу на багатьох проектах, в тому числі монорепозиторіях, що надає можливість збирати дані по всій кодовій базі, або на її визначених підмножинах;
- Кожна команда має мати змогу додати проект або множину проектів в систему і окремо відслідковувати результати по цій множині по бажаним метрикам, не обов'язково всім доступним;
- Результати аналізу мають бути доступні через зручний та інтуїтивно зрозумілий веб-інтерфейс, доступний через сучасний браузер;
- Результати мають оновлюватись що найрідше раз на добу або за вимогою. Має бути чітко відомо коли саме було останнє оновлення;
- На основі результатів аналізів мають надаватися лише мінімальні висновки та рекомендації, а не агреговані вердикти. Основна цінність результатів у них самих, робити висновки здебільшого залишається користувачам;
- Аналізатор має надавати нову, не критичну інформацію про проекти. Це протиставляється іншим інструментам які спеціалізуються на пошуку саме помилок та якомога швидшому повідомленю про них. Цей інструмент не призначений для таких задач.

На основі цих вимог, було визначено що система має складатися з двох частин: фронт-енд застосунок та бек-енд частина. Так як фронт-енд частині достатньо бути звичайним клієнтським веб-застосунком, вибір технологій, архітектури та кращих практик базується майже виключно на підходах індустрії, а не на особливостях предметної області, через що не представляє багато інтересу в цій роботі. Бек-енд компонент є більш складним та загалом цікавим для дослідження через те що на нього покладається власне аналіз та надання його результатів. Обидві частини в результаті розроблялися з допомогою мови програмування TypeScript та суміжних технологій, таких як React, Node.js, Jest, Lodash та інших.

В плані архітектури, бек-енд був побудований як відносно звичайне CLI Node.js застосування, яке запускається кожної ночі в CI середовищі, виконує аналіз по заданим проектам і зберігає результат. Після чого запускається збірка клієнтської частини під час якої результати аналізатора статично пакуються разом з клієнтом. Такий підхід загалом досить простий, однак часом стало очевидно що кращим, хоча і значно складнішим, рішенням було б все ж мати бек-енд як повноцінний веб-сервіс з базою даних, API та іншими суміжними властивостями. Статичні результати аналізу значно обмежували можливості для розширення функціоналу системи в напрямку покращення зручності. Наприклад, що веб-сервіс зробив би набагато простішим:

- Динамічне оновлення даних, наприклад через інтерфейс;
- Інтеграцію з іншими сервісами, наприклад репозиторіями та комунікаційними системами;
- Надання історії результатів - як змінювались результати метрик з часов, чи є тренд на покращення, чи стає гірше і в яких проміжках часу;

- Швидше перше завантаження фронт-енд частини, так як не потрібно відразу вантажити всі результати.

Однак, незважаючи на все це, варіант з CLI є простішим в реалізації і все одно досить потужним для більшості задач. Далі буде розглянуто три основних частини системи:

1. Підготовка до аналізу - завантаження та перед-обробка даних;
2. Аналіз - статичний аналіз проектів по визначеним метрикам;
3. Відображення - фронт-енд частина.

3.2. Підготовка до аналізу

3.2.1. Конфігурація

Все починається з конфігурації. Однією з початкових вимог була можливість аналізувати відразу багато проектів, отже маємо надавати можливість якось задавати їх список. Іншою важливою вимогою було надати можливість окремим людям або командам створювати свої конфігурації із обраними проектами та аналізаторами, не обов'язково всіма. Таким чином, приходимо до такої структури:

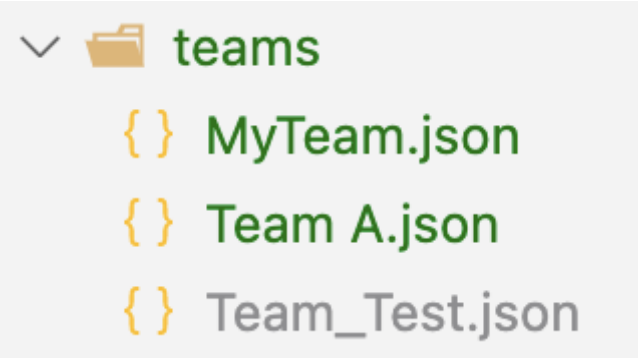

Рисунок 3.1 - Список "команд"

Введемо поняття "команди" або "компанії", яке визначає окрему конфігурацію аналізатора для певної множини проектів. Така конфігурація включає список проектів, метрики та інші налаштування. Під кожну команду на етапі візуалізації відводиться окрема сторінка. Для зручності розробки також додана можливість створювати тестові конфігурації, які відрізняються префіксом "_Test", які ігноруються системами контролю версій і таким чином дозволяють перевіряти локально будь-які гіпотези або зміни в програмі без потреби редагувати та потім повертати до початкового стану якусь з існуючих конфігурацій.

Розглянемо простий приклад такої конфігурації:

```
{
 "repos": [
   {
     "gitName": "VernonHawk/Flattr"
   },
   {
     "gitName": "VernonHawk/Biblio",
     "includedPaths": ["main-project"]
   },
   {
     "gitName": "VernonHawk/Brewtal",
     "ignoredPaths": ["/__tests__/"]
   }
 ],
 "analyzers": {
   "custom": ["todosCounter", "TSStrict"],
   "todoReport": true,
   "versions": {
     "dependencies": [
       {
         "name": "typescript",
         "wantedVersion": "latest"
       }
     ]
   }
 }
}
```

Рисунок 3.2 - Приклад конфігурації команди

Тут можна виділити дві основні частини:

1. Репозиторії.

   В цьому прикладі вказаний список GitHub репозиторіїв які потрібно зклонувати для аналізу. Для кожного з них за потреби можна вказати `includedPaths` та `ignoredPaths`. Однією з початкових вимог була можливість роботи з монорепозиторіями, тобто система має вміти ідентифікувати всі окремі підпроекти репозиторію і аналізувати їх. Але проблема в тому, що інколи потрібно взяти лише частину такого

монорепозиторію, в чому і допомагає ігнорування, або навпаки виключне обрання певних шляхів у репозиторії.

Інколи ж є потреба більш гнучкого задання списку проектів, для чого використовуються так звані генератори репозиторіїв - функції що повертають такий самий список як в прикладі, але генерують його в програмі, використовуючи всю потужність високорівневої мови програмування. Приклад такого генератора буде розглянуто далі.

2. Аналізатори.

   Аналізатори задаються як мапа де кожній назві аналізатора відповідає його конфігурація. Найпростіший варіант це просто булеве значення що відповідало б тому чи включений аналізатор (однак немає сенсу вказувати false, адже тоді можна просто не додавати цей аналізатор), але для різних метрик це можуть бути більш складні структури, такі як списки або об’єкти. Важливо що аналізатори задані у такий спосіб будуть в результаті всі відображені на одні сторінці на сайті.

   Якщо ж є бажання або потреба відокремити аналізатори, можна використати конфігурацію з вкладками (tabs), яка дозволяє задати список вкладок, кожна зі своєю назвою та списком аналізаторів:

```
{
 "reposGenerator": "repoGeneratorExample",
 "tabs": [
   {
     "name": "Miscellaneous",
     "analyzers": {
       "custom": ["todosCounter", "readmePresent", "skippedTestsCount"]
     }
   },
   {
     "name": "Dependencies",
     "analyzers": {
       "custom": ["TSStrict", "outdatedDependencies"],
       "versions": {
         "dependencies": [
           {
             "name": "react-native",
             "wantedVersion": "18"
           },
           {
             "name": "typescript",
             "wantedVersion": ">=6"
           }
         ]
       }
     }
   }
 ]
}
```

Рисунок 3.3 - Приклад конфігурації із генератором репозиторіїв та двома вкладками

Розглянемо приклад генератора репозиторіїв який отримує список з конфігурації іншого програмного проекту. Отже, потрібно склонувати той репозиторій, дістати список та перетворити його у бажаний вигляд:

```
import gitInfo from 'hosted-git-info';
import { compact, uniqBy } from 'lodash';

async function repoGeneratorExample() {
 const urls = await Promise.all(
   (await cloneAppsManager()).map(async (artifactId: string) => {
     const metadata = (await getArtifact(artifactId))?.[0]?.metadata;
     if (!metadata) {
       return false;
     }

     return {
       gitName: gitInfo.fromUrl(metadata.gitUrl)!.path(),
       ignoredPaths: ['example', 'modulesMock'],
       ...(metadata?.moduleRelativePath !== '.' && {
         includedPaths: [`/${metadata.moduleRelativePath}/package.json`],
       }),
     };
   }),
 );

 return uniqBy(compact(urls), ({ includedPaths, gitName }) =>
   includedPaths ? gitName + includedPaths[0] : gitName,
 );
}

async function cloneAppsManager(): Promise<string[]> {
 const { path } = await createTempDir('appsManager');
 await execAsync(
   `git clone git@github.com:account/repo.git ${path} --depth 1`,
 );

 return (await readFile(`${path}/package.json`)).modules.map(
   (moduleName: string) => `com.companyName.npm:${moduleName}`,
 );
}
```

Рисунок 3.4 - Приклад генератору репозиторіїв

В цьому прикладі в Apps manager репозиторії зберігається список бажаних модулів у вигляді npm залежностей. Ці залежності потрібно ще

перетворити на шляхи до GitHub репозиторіїв, в чому допомагає умовна функція `getArtifact`, яка отримує дані про модуль з відповідного CI середовища. Маючи такий список можна переходити до клонування проектів для аналізу.

3.2.2. Завантаження

Конфігурація кожної команди обробляється повністю окремо і послідовно, хоча цей процес і можна не складно паралелізувати. Для кожної потрібно отримати список репозиторіїв, зклонувати їх та застосувати вказані фільтри:

```
async function scanTeam(team: string, configuration: TeamConfiguration) {
 const teamConfig = configuration[team];
 const fetchFullHistory = needFullHistory(teamConfig);

 const projects = Promise.all(
   (await getTeamRepos(teamConfig)).map(async (repo) =>
     getRepoProjects((await createTempDir('')).path, repo, fetchFullHistory),
   ),
 );

 // ...
}

async function getTeamRepos(config: CommonTeamConfiguration) {
 return [...(config.repos ?? []), ...(await executeReposGenerator(config))];
}

async function getRepoProjects(
 tempPath: string,
 { gitName, ignoredPaths, includedPaths }: Repo,
 fetchFullHistory: boolean,
) {
 const allProjects = await getMonorepoProjects(
   tempPath,
   gitName,
   fetchFullHistory,
 );

 return removeIgnoredProjects(allProjects, ignoredPaths, includedPaths);
}
```

Рисунок 3.5 - Високорівнево завантаження проектів

Окремо розглянемо параметр `fetchFullHistory`, що відповідає за те чи потрібно клонувати всю історію репозиторія, що значно повільніше ніж завантажувати лише останній коміт. Деяким аналізаторам може бути потрібна ця історія, наприклад розглянутим раніше звіту по TODO, який використовує її для отримання інформації про те хто та коли востаннє змінював певні частини коду.

Далі функція `executeReposGenerator` просто виконує генератор репозиторіїв, якщо він заданий. `getMonorepoProjects` власне клонує репозиторії, виділяє серед них окремі проекти (в одному репозиторії їх може бути багато) та повертає оброблений локальний шлях до них. `removeIgnoredProjects` прибирає з проектів ті що не відповідають заданим в конфігурації критеріям (`ignoredPaths` та `includedPaths`). Реалізації цих функцій не наводяться та не розбираються так як вони є не складними, але є не дуже цікавими в контексті основної теми роботи.

Після того як всі проекти командної конфігурації склоновані та підготовлені, час переходити до власне метричного статичного аналізу.

3.3. Аналіз

Під час роботи над аналізаторами стало помітно що деякі з них слідують більш простій логіці ніж інші і не вимагають якоїсь інформації про проекти більш ніж кореневий локальний шлях та не потребують якоїсь особливої візуалізації. Такі прості аналізатори будемо в проекті називати “технічними боргами” через те що спочатку це були лише, власне, борги, а потім додалися і інші метрики. Такі “борги” всі відносяться до одного аналізатору `analyzeCustomDebts`, який відображає одну таблицю для всіх боргів. Це зручно тому що можна:

- Спростити розробку - від розробника вимагається лише визначити та реалізувати функцію що приймає на вхід шлях до проекту та повертає результат у визначеному форматі (в розглянутій системі це вже продемонстрований раніше `AnalysisResult`);
- Перевикористати максимум коду - знову таки, розробник лише задає одну функцію і автоматично отримує її підключення до аналізатору та відображення на фронт-енд частині в загальній таблиці;
- Ефективно використати місце на UI - замість того щоб вигадувати як відобразити великі списки даних по кожному проекту, можна їх всі згрупувати у одну таблицю де їх буде всім користувачам просто побачити і порівняти.

Всі ж інші аналізатори працюють зі списками проектів і мають спеціальні власні компоненти на фронт-енд частині, які не обов'язково є таблицями. Розглянемо високорівнево як проходить аналіз:

```
async function getAnalysisRes(config: TeamConfiguration, projects: Project[]) {
 if (isPlainTeamConfig(config)) {
   return {
     metaInfo: { reportCreatedAt: new Date().toISOString() },
     ...(await analyze(config.analyzers, projects)),
   };
 }

 return {
   metaInfo: { reportCreatedAt: new Date().toISOString() },
   tabs: await Promise.all(
     config.tabs.map(async ({ name, analyzers }) => ({
       name,
       analyzers: await analyze(analyzers, projects),
     })),
   ),
 };
}

async function analyze(analyzers: AnalyzersConfiguration, projects: Project[]) {
 const { custom, buildLog, todoReport } = analyzers;
 const result: AnalyzersResults = {};

 if (custom) {
   result.customDebts = await analyzeCustomDebts(projects, custom);
 }
 if (buildLog) {
   result.warnings = await analyzeBuildLog(projects);
 }
 if (todoReport) {
   result.todoReport = await analyzeTodo(
     projects,
     typeof todoReport === 'object' ? todoReport : {},
   );
 }
 // other analyzers...
 return result;
}
```

Рисунок 3.6 - Високорівнево аналіз проектів

Функція `getAnalysisRes` викликається з `scanTeam` після завантаження та обробки всіх проектів. В першу чергу, вона перевіряє чи конфігурація з вкладками, чи без, бо тоді відрізняється і формат результату і логіка отримання списку аналізаторів. Також окремо виділимо властивість `reportCreatedAt` яка пізніше буде використана на клієнті для задоволення вимоги про те, що має бути відомо коли дані були востаннє оновлені.

Далі кожний аналізатор окремо викликається (якщо він заданий) з відповідними налаштуваннями. При збільшенні кількості аналізаторів є сенс ввести механізм що дозволив би не додавати код в функцію `analyze`, а, наприклад, лише додавати елемент до асоціативного масиву, або взагалі покластись на відповідність назви аналізатора назві файла. Приклад такого динамічного підходу розглянемо у реалізації аналізатора `analyzeCustomDebts` який якраз відповідає за згадані трохи раніше “борги”:

```
type CustomAnalyzers =
 | 'avgSizeOfBiggestFunctionsAndClasses'
 | 'nonNullAssertionCount'
 | 'outdatedDependencies';

const analyzeCustomDebts = (projects: Project[], conf: CustomAnalyzers[]) =>
 Promise.all(
   projects.map(async (project) => ({
     name: project.name,
     packageRemoteUrl: project.packageRemoteUrl,
     ...(await analyzeProject(project, conf)),
   })),
 );

async function analyzeProject(project: Project, analyzers: CustomAnalyzers[]) {
 const results: { [analyzerName: string]: AnalysisResult } = {};

 for (const functionName of analyzers) {
   const functionModule = require(`../customDebts/${functionName}`);
   const analyzerName = functionModule.columnName;
   results[analyzerName] = await functionModule[functionName](project.path);
 }

 return results;
}
```

Рисунок 3.7 - Аналізатор технічних боргів

В цьому прикладі `require` динамічно імпортує функцію за назвою та викликає її із шляхом до проекту. За потреби та бажання цей функціонал можна розширити можливістю задавати налаштування окремим боргам, що може бути корисно тому що різні команди або окремі користувачі можуть хотіти бачити дещо різні метрики.

Коли всі аналізатори були виконані і дані по всім командам зібрані у відповідних json файлах, результат зберігається і є готовим до відображення на UI.

3.4. Відображення

Фронт-енд частина побудована як звичайне і досить просте React застосування, навіть без використання інструментів керування станом. Відразу розглянемо вигляд сторінки типової команди та на основі цього розберемо всю візуалізаційну частину системи:

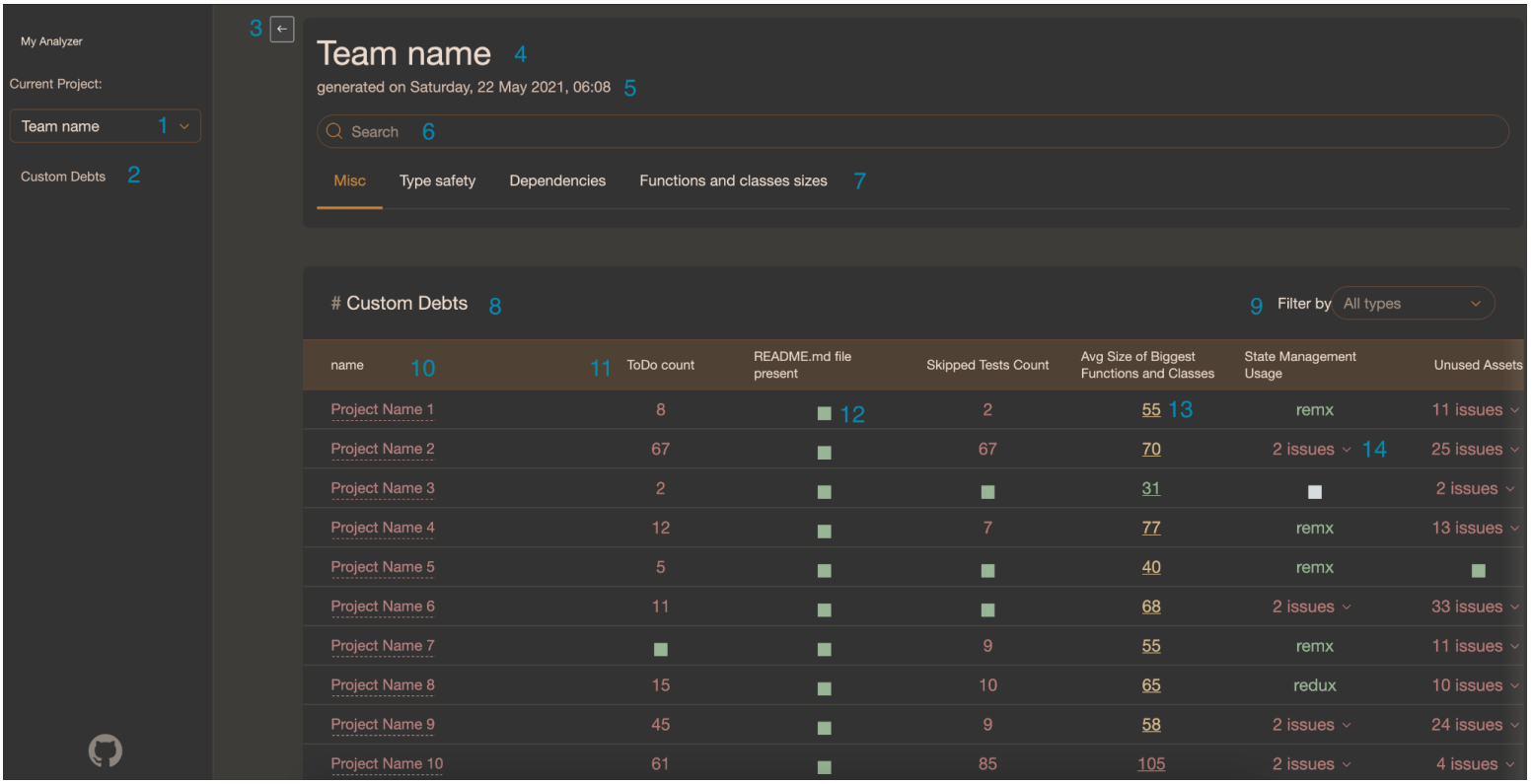

Рисунок 3.8 - UI фронт-енд частини системи

На зображенні можна побачити багато елементів:

1. Вибір команди. Так як команд може бути багато, надається можливість обирати бажану для перегляду. Вибір також відображається у пошуковому рядку веб-браузера, що дозволяє зберегти посилання на конкретну команду. Якщо на вибір є лише одна команда, вона автоматично обирається та цей список не відображається (що можливо, якщо якась команда створює свою окрему версію фронт-енд відображення).
2. Список аналізаторів. Кожен з аналізаторів на сторінці відображається у цьому списку, даючи користувачам уявлення про те

що вони можуть знайти якщо пролистають нижче, адже можна побачити перший аналізатор і подумати що це все. Якщо натиснути на аналізатор в даному списку, сторінка пролистається до цього аналізатора. Це також відображається в пошуковому рядку браузера.

3. Перемикач бокової панелі. Дозволяє закрити або навпаки відкрити бокову панель із вибором команди та списком аналізаторів. Якщо доступна лише одна команда і на сторінці наявний лише один аналізатор, панель за замовчуванням закрита. В інших випадках вона відкрита.
4. Назва команди. Точно визначається назвою json конфігурації команди на етапі аналізу.
5. Час і дата останнього оновлення даних.
6. Пошук по сторінці. Пошук можна визначити окремо для кожного аналізатора, але зазвичай він буде фільтрувати дані за наявністю якогось підрядка в назвах проектів. Відображається в пошуковому рядку браузера.
7. Вкладки. Якщо конфігурація проекту задає вкладки, тут можливе переключення між ними. Пошук працює в межах однієї вкладки, але не очищається при переході на іншу, що дозволяє подивитись всі результати по підмножині проектів. Відображається в пошуковому рядку браузера.
8. Аналізатор. Кожний аналізатор має свій компонент, який зазвичай є таблицею, але може бути і списком, графіком, їх комбінацією чи будь-яким іншим доречним способом представлення результатів аналізу. На даному зображені представлений аналізатор боргів.

   Як потенційне покращення можна було б зливати заголовок сторінки та заголовок аналізатора для максимізації корисної інформації на екрані.

9. Фільтри. Дозволяють фільтрувати дані за певними критеріями, наприклад відобразити лише ті проекти у яких все добре, або навпаки є хоча б щось погане. Іншим прикладом може бути фільтрування даних за автором чи датою змін (у випадку звіту по TODO). Кожний аналізатор може мати свої специфічні фільтри або не мати зовсім.

   В цій системі були також додані фільтри по проекту, але потім стало зрозуміло, що з наявністю пошуку вони не потрібні.
10. Стовпчик назв. Закріплений стовпчик що містить назви проектів, які визначаються назвою проекту в конфігурації (package.json) шляхом розбиття на слова по знаку “-” і капіталізації слів. Кожна назва є також посиланням на кореневу папку проекту в GitHub репозиторії.

    Це є корисним навіть не враховуючи весь інший функціонал аналізатора, тому що у великій кодовій базі різні модулі можуть бути розкидані по багатьом різним репозиторіям та моно-репозиторіям, а такий список дозволяє всім швидко отримати доступ до цих модулів.

    Як покращення, за потреби до цього стовбчику можна додати також посилання на інші доречні ресурси, наприклад на окрему сторінку проекту або останню вдалу збірку на CI. Однак, варто враховувати те що назви проектів можуть бути досить довгими і не влізати у клітинку, через що можуть не влізати і посилання.
11. Технічні борги. В цьому прикладі задані технічні борги відображаються як інші стовпчики таблиці. Результатом може бути текст, число, варіація попередніх двох варіантом з посиланням, список, або просто статус.
12. Статус. Якщо в результаті присутній лише статус, він відображається квадратом з кольором. Статус присутній і в інших варіантах результату і забарвлює їх у відповідний колір:
    a. GOOD (добре) - зелений;

b. MIXED (змішаний/середній) - жовтий
c. BAD (погано) - червоний;
d. IRRELEVANT (недоречно) - сірий. Використовується там де метрика не є доречною для проекту, наприклад в аналізаторі TS Strict для проекту що не використовує TypeScript.

13. Посилання. Додає інтерактивності у систему, дозволяючи переходити з таблиці на інші сторінки, вкладки або таблиці. В цьому прикладі натискання на результат боргу “Avg Size of Biggest Functions and Classes” для якогось з проектів переводить користувача на вкладку “Functions and classes sizes” із звітом по розмірам функцій та класів в проекті і встановлює пошук на назву проекта, таким чином показуючи дані лише по цьому проекту.
14. Список. Якщо даних більше ніж один рядок, вони перетворюються на згортуємий список, який відображає загальну кількість елементів та зберігає дуже багато місця. Без такого автоматичного згортання деякі аналізатори робили навігацію по таблиці дуже складною.

Раніше вже було згадано що дані статично зберігаються з модулем аналізатора і потім імпортуються фронт-енд частиною. Результат кожної команди зберігається у окремому json файлі з назвою TeamName-result.json (де замість TeamName стоїть власне назва команди) і окремо також зберігається файл resultsRegistry.json, який містить мапу проектів. В клієнтському коді завантаження даних виглядає так:

```
import resultsRegistry from 'tech-debts-analyzer/resultsRegistry.json';

function App() {
 const [results, setResults] = useState<TechDebtsResults>({});

 useEffect(() => {
   const importResults = async () => {
     const resultsContent: TechDebtsResults = {};

     for (const teamName in resultsRegistry) {
       const result = await import(
         `tech-debts-analyzer/${resultsRegistry[teamName]}-result.json`
       );
       resultsContent[teamName] = result.default;
     }

     setResults(resultsContent);
   };

   void importResults();
 }, []);

 // ...
}
```

Рисунок 3.9 - Завантаження результатів аналізу на клієнті

Отже, спочатку статично зчитується регістр, а потім по кожному його елементу динамічно імпортується результат кожної команди.

Після цього користувач обирає бажану команду з випадаючого списку і потрібно відобразити командну сторінку що складається з двох частин: заголовку та тіла. Саме в тілі знаходяться аналізатори:

```
interface Props {
 results: AnalyzersResults;
 searchValue: string | null;
 onClearSearchClick: () => void;
 onItemClickConfig:{ [key: string]: (row: any) => void };
}

const TeamPageBody = ({ results, ...commonProps }: Props) => (
 <>
   {Object.keys(results).map((analysisName) => (
     <div className={s.card} key={analysisName}>
       {analysisName === 'skippedTestsReport' ? (
         <SkippedTestsReportTable
           {...commonProps}
           results={results[analysisName]!}
         />
       ) : analysisName === 'todoReport' ? (
         <TodoReportTable {...commonProps} results={results[analysisName]!} />
       ) : analysisName === 'experimentsReport' ? (
         <ExperimentsReport {...commonProps} results={results[analysisName]!} />
       ) : analysisName === 'functionsAndClassesListBySize' ? (
         <FunctionsListTable {...commonProps} results={results[analysisName]!} />
       ) : (
         <AnalyzerTable
           {...commonProps}
           analysisName={analysisName}
           results={results[analysisName]!}
         />
       )}
     </div>
   ))}
 </>
);
```

Рисунок 3.10 - Тіло сторінки команди

Тут можна побачити що деякі аналізатори мають свої спеціальні компоненти, а деяким вистачає гнучкої таблиці. Властивість `onItemClickConfig` використовується для переходів між аналізаторами.

3.5. Висновки

В цьому розділі було розглянуто які вимоги можуть ставитись до реальної системи для статичного метричного аналізу проектів, та як відповідати цим вимогам. Коротко описано підготовку до аналізу, власне виконання аналізаторів та відображення результатів.

Висновки

В результаті роботи було досліджено, описано, реалізовано та проаналізовано різноманітні метрики програмних проектів що визначаються за допомогою статичного аналізу. Були розглянути та порівняні їх методи реалізації, такі як аналіз метаданих, парсинг тексту та регулярні вирази, абстрактні синтаксичні дерева та графи залежностей між сутностями. Що не менш важливо, були детально описані загальні підходи до визначення, імплементації та оцінювання таких метрик. Крім того, була описана реальна програмна веб-система для статичного високорівневого, комплексного та водночас глибокого аналізу багатьох програмних проектів.

Результати дослідження можуть бути використані, по-перше, як сукупність знань та досвіду для подальших досліджень в галузі та побудови власних систем метричного статичного аналізу. І по-друге, сама наведена практична реалізація вже може бути використана для аналізу проектів в організації чи окремою командою.

Як шляхи розвитку роботи можна виділити:

- Детальніше дослідження найменш розглянутого в роботі підвиду аналізаторів - тих що потребують графи зв’язків;
- Визначення інших основних підходів до реалізації або переосмислення виділених в роботі;
- Дизайн, розробка та аналіз нових аналізаторів;
- Розширення практичної системи як новими аналізаторами, так і інтеграціями, методами візуалізації, інтерактивністю, тощо.

Список використаних джерел

1. Bagheri E. Assessing the Maintainability of Software Product Line Feature Models using Structural Metric / Bagheri E., Gasevic D. // Software Quality Journal. – 2011. – № 19. – С. 579-612.
2. Language Guide (proto3): What's Generated From Your .proto? [Електронний ресурс]: Матеріал з офіційної документації Protocol Buffers. Перевірено 15 травня 2021. Режим доступу: https://developers.google.com/protocol-buffers/docs/proto3#whats_generated_from_your_proto.
3. Swagger UI [Електронний ресурс]: Матеріал з офіційної документації Swagger. Перевірено 15 травня 2021. Режим доступу: https://swagger.io/tools/swagger-ui.
4. Офіційний сайт PVS-Studio [Електронний ресурс]: Перевірено 15 травня 2021. Режим доступу: https://pvs-studio.com.
5. Офіційний сайт SonarQube [Електронний ресурс]: Перевірено 15 травня 2021. Режим доступу: https://www.sonarqube.org.
6. List of Customers [Електронний ресурс]: Матеріал з офіційної документації SonarSource. Перевірено 15 травня 2021. Режим доступу: https://www.sonarsource.com/customers.
7. TSConfig Reference [Електронний ресурс]: Матеріал з офіційної документації TypeScript. Перевірено 15 травня 2021. Режим доступу: https://www.typescriptlang.org/tsconfig.
8. Semver: Tilde and Caret [Електронний ресурс]: Tim Oxley. *The NodeSource Blog* - 2014. Перевірено 15 травня 2021. Режим доступу: https://nodesource.com/blog/semver-tilde-and-caret.

9. Regular Expressions Tutorial - Learn How to Use and Get The Most out of Regular Expressions [Електронний ресурс]: Jan Goyvaerts. *www.regular-expressions.info* - 2019. Перевірено 15 травня 2021. Режим доступу: http://www.regular-expressions.info/tutorial.html.
10. Abstract syntax tree [Електронний ресурс]: *Wikipedia* - 2021. Перевірено 17 травня 2021. Режим доступу: https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=1016693387.
11. The ESTree Spec [Електронний ресурс]: *GitHub* - 2021. Перевірено 17 травня 2021. Режим доступу: https://github.com/estree/estree.