Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка програмного засобу для ефективного рефакторингу та контрою якості архітектури проєкту

Виконав: студент IV курсу, групи СПс-42

спеціальності 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

(підпис) Ганчук Н.А. (прізвище та ініціали)

Керівник (підпис) Гащин Н.Б. (прізвище та ініціали)

Нормоконтроль (підпис) Стоянов Ю.М. (прізвище та ініціали)

Завідувач кафедри (підпис) Петрик М.Р. (прізвище та ініціали)

Рецензент (підпис) (прізвище та ініціали)

Тернопіль - 2024

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М.Р.
(підпис) (прізвище та ініціали)
«__» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

Студенту Ганчуку Назарію Андрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка програмного засобу для ефективного рефакторингу та контрою якості архітектури проєкту

Керівник роботи Гащин Надія Богданівна, к.т.н., доц.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «15» 04 2024 року № 4/7-344
2. Термін подання студентом завершеної роботи 24.06.2024 р.
3. Вихідні дані до роботи наукові літературні джерела

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ
1. Аналіз предметної області.
2. Постановка завдань і концепція інструменту
3. Реалізація та впровадження рішення.
4. Безпека життєдіяльності, основи охорони праці
Висновки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Титулка. 2. Актуальність 3. Мета, задачі дослідження. 4. Порівняльна таблиця аналогів
5. Вимоги до рішення. 6. Архітектура програмного інструменту
7. Архітектурні компоненти, Технологічний стек. 8. Граф залежностей тестового проекту.
9. Діаграма компонентів, «Чиста архітектура». 10. Результат підрахунку метрик
Instability та Abstractness для модулів. 11. Результат кластеризації модулів, Адаптація алгоритму. 12. Приклади тестових даних із різних проєктів.
13. Тестування інструменту для різних гіперпараметрів алгоритму. 14. Висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи хорони праці			

7. Дата видачі завдання 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	16.04 – 18.04.24	*Виконано*
2.	Підбір джерел про засоби аналізу компонентної		
	архітектури програмного забезпечення	19.04 – 22.04.24	*Виконано*
3.	Опрацювання джерел про інструменти для рефакторингу		
	та контролю якості архітектури проєктів	23.04 – 29.04.24	*Виконано*
4.	Проведення дослідження щодо розробки програмного		
	засобу для рефакторингу та контролю якості	30.04 – 04.05.24	*Виконано*
5	Розроблення програмного коду	05.05 – 12.05.24	*Виконано*
6.	Оформлення розділу «Аналіз предметної області»	13.05 – 18.05.24	*Виконано*
7.	Оформлення розділу «Постановка завдань і концепція		
	інструменту»	19.05 – 24.05.24	*Виконано*
8.	Оформлення розділу «Реалізація та впровадження		
	рішення»	25.05 – 31.05.24	*Виконано*
9.	Виконання завдання до підрозділу «Безпека		
	життєдіяльності, основи хорони праці»	01.06 – 05.06.24	*Виконано*
10.	Оформлення кваліфікаційної роботи	06.06 – 09.06.24	*Виконано*
11.	Нормоконтроль	10.06 – 13.06.24	*Виконано*
12.	Перевірка на плагіат	12.06 – 16.06.24	*Виконано*
13.	Попередній захист кваліфікаційної роботи	17.06 – 20.06.24	*Виконано*
14.	Захист кваліфікаційної роботи	25.06.24	

Студент _____ Ганчук Н.А.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Гащин Н.Б.
(підпис) (прізвище та ініціали)

РЕФЕРАТ / ABSTRACT

Кваліфікаційна робота бакалавра містить: 52 сторінки, 20 рисунків, 26 джерел та 3 додатки.

АРХІТЕКТУРНА МЕТРИКА, МАШИННЕ НАВЧАННЯ, ПАРСИНГ, РЕФАКТОРИНГ, ЯКІСТЬ АРХІТЕКТУРИ

У кваліфікаційної роботі бакалавра було розроблено допоміжний інструмент для проведення аналізу компонентної архітектури на основі механізмів машинного навчання.

Вивчено архітектурні проблеми у розробці, досліджено існуючі аналоги. Описана концепція програмного інструменту, продумано алгоритми виявлення архітектурних проблем і труднощі у реалізації, вивчено підходи машинного навчання та вибрано оптимальний. При проведенні дослідження підтвердилася гіпотеза про роль зворотніх зв'язків користувачів для модулів для покращення архітектури. Розроблений програмний інструмент дає змогу виявляти розбіжності у структурі схожих за абстрактністю та стійкістю модулів. Це дозволить ефективніше рефакторити та контролювати якість архітектури проєкту.

ARCHITECTURE METRICS, MACHINE LEARNING, PARSING, REFACTORING, ARCHITECTURE QUALITY

In the bachelor's thesis, an auxiliary tool was developed for the analysis of component architecture based on machine learning mechanisms.

Architectural problems in development were studied, existing analogues were studied. The concept of the software tool is described, algorithms for identifying architectural problems and difficulties in implementation are thought out, machine learning approaches are studied and the optimal one is selected. The research confirmed the hypothesis about the role of user feedback for modules to improve the architecture. The developed software tool makes it possible to detect differences in the structure of modules that are similar in terms of abstractness and stability. This will allow more effective refactoring and quality control of the project architecture.

ПЕРЕЛІК СКОРОЧЕНЬ І ТЕРМІНІВ

API (application programming interface) – набір визначень підпрограм, протоколів взаємодії та засобів для створення програмного забезпечення.

Cohesion (пов’язаність) – умовна міра того наскільки внутрішньо пов'язаним є код в одному модулі програми (його функції, змінні, класи).

Coupling (зв’язність) – умовна міра, у якій модуль програми зовнішньо залежить від кожного іншого модуля (використовує сторонні функції, змінні, класи).

IDE (integrated development environment) – інтегроване середовище розробки.

Instability (міра нестійкості модуля) – відношення вихідних залежностей до модуля до суми вхідних.

LCOM (Lack of Cohesion of Methods) – непов’язність коду (архітектурна метрика).

Парсинг (синтаксичний анааліз) (parsing) – процес аналізу вхідної послідовності символів, з метою розбору граматичної структури згідно із заданою формальною граматикою.

ПЗ – програмне забезпечення.

Рефакторинг — процес редагування програмного коду, внутрішньої структури програмного забезпечення для полегшення розуміння коду та внесення подальших правок без зміни зовнішньої поведінки самої системи.

Технічний борг (TechDebt) – відображає додаткову роботу розробника, яка виникає, коли код, котрий легко втілити в короткотерміновій перспективі, використовують замість застосування найкращого рішення.

ЗМІСТ

ВСТУП

Актуальність теми дослідження. Проектування надійної, гнучкої та масштабованої архітектури систем є одним із основних завдань у галузі розробки ПЗ. Добре спроектований код явно описує систему та її предметну область, а також дозволяє простіше орієнтуватися в кодовій базі, що прискорює розробку ПЗ. При поганій архітектурі, навпаки, складніше знаходити зв'язки між модулями, грамотно їх перевикористовувати і змінювати без непередбачених наслідків і ланцюгової реакції змін.

Тому значимість масштабованої архітектури проекту залишається актуальною й у наші дні, оскільки системи, що розробляються, значно збільшилися за складністю і за обсягом команд [1,2]. Архітектура впливає і на час впровадження нових людей у команду, на швидкість розробки нової функціональності (Time-to-Market, комунікацію всередині команди та загальну стабільність продукту).

При цьому проекти розвиваються, доповнюються вимогами, а розробники ПЗ намагаються адаптувати кодову базу проекту під цю постійно зростаючу складність. Але ключова суперечність у тому, при цьому ми не маємо навичок передбачення майбутнього, щоб вгадати, в якому напрямку піде розвиток проекту і як він розширюватиметься [1,2]. Тому вкрай важливо - спочатку закладати таку архітектуру, яку можна було б гнучко адаптувати під нові умови та вимоги, та актуалізувати її з розвитком проекту з мінімальними витратами.

Загальновідомі патерни проектування та методології [3] актуальні і зараз, але їх наявність не дуже допомагає ситуації - на жаль, вони дуже абстрактні та конфліктні між собою. Через це виникає безліч різних інтерпретацій та проблем у практичній застосовності цих практик [4]. Деякі розробники нехтують ними у моменті на користь прискорення розробки, а десь застосовують надмірно, чим погіршують розвиток проекту. Різноманітність різних парадигм і відсутність єдиного стандарту лише ускладнює ситуацію.

Мета роботи: вирішити проблему підтримки архітектури JavaScript-проєктів у актуальному стані, за допомогою допоміжного інструментарію.

Для досягнення мети потрібно вирішити наступні завдання:

– вивчити літературні джерела та інші матеріали з обраної предметної галузі;

– вивчити існуючі аналоги на вирішення поставленої проблеми;

– зібрати ряд доступних проектів і дослідити практики проектування, що використовуються в них;

– перевірити поставлені припущення, виявити загальні закономірності та уточнити заявлені проблеми;

– зібрати базовий прототип рішення на підтвердження припущень;

– перевірити дієвість прототипу та уточнити вимоги до інструменту, що розробляється;

– дослідити ідеї для покращення інструменту в аспектах аналізу та пропонованого рефакторингу декомпозиції модулів;

– вивчити отримані результати та скоригувати подальшу діяльність;

– впровадити в експериментальному порядку розроблене рішення до низки доступних проектів та зафіксувати результати.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

В рамках роботи була поставлена проблема контролю якості архітектури модулів при розробці проектів, без всього необхідного контексту розвитку проекту і без можливості через це досягти ідеальної архітектури в моменті (just-in-time) [5]. Стратегія роботи зі складністю "в моменті" полягає в тому, щоб не намагатися прорахувати та оптимізувати заздалегідь те, що невідомо.

1.1 Вивчення архітектурних проблем у розробці

Проектування систем та роль архітектури. Під час проектування системи розробник будує її передбачувану модель відносин між сутностями у цій системі. При цьому бізнес-вимоги не статичні, вони постійно змінюються. Добре спроектована система здатна витримати та адаптуватися під ці зміни та продовжить функціонувати (зі збереженням простоти навігації в кодобазі, перевикористання логіки, пошуку зв'язків між модулями). Тому важливо скласти бажані, але досяжні вимоги у рамках розробки допоміжного інструменту проектування.

Поточний стан справ та існуючі підходи. Особистий досвід розробки та спілкування з командами з різних проектів лише підтверджує це. Погіршує становище наявна мала кількість сучасного інструментарію, що допомогло б відстежувати архітектуру проектів у динаміці. Доводиться покладатися на код-рев'ю, процеси, конвенції у команді - тобто на людський чинник. Є інструменти, що допомагають перевіряти конкретні правила, прив'язані до мови або бібліотеки, але ніде немає перевірок і порад щодо того, як декомпозувати компоненти архітектури та пов'язувати між собою із збереженням погіршення уніфікованості та перевикористовуваності.

Вимоги до архітектури. Якщо агрегувати всі вивчені матеріали [4,6,7,8,9] та досвід корпоративної розробки, то можна виділити очікування від гарної архітектури:

– явність і простота - зв'язки модулів, їх перевикористання та дублювання логіки, рівень абстрактності відповідної складності проекту;

– контроль та передбачуваність - безпроблемне розширення, модифікація та видалення коду в проекті, ізоляція логіки;

– адаптованість та стійкість - збереження складності розробки при зміні бізнес-вимог, масштабуванні проекту та команди.

Архітектурні проблеми у проектах. Спираючись на доступні реальні проекти, було розпочато вивчення практичних архітектурних проблем. З історії коммітів у репозиторії [10], були відстежені основні зміни при рефакторинг проекту. Потім було побудовано орієнтований граф залежностей [11], яким можна проаналізувати наявні у проекті проблеми.

При ретельному вивченні проектів, власного досвіду та інших розробників були помічені такі архітектурні проблеми:

– Normalizing Public API. Багато модулів не контролюють те, як використовуються їхні публічні контракти та наскільки глибоко використовується внутрішні частини реалізації, які не повинні бути доступними для зовнішнього використання;

– Decomposition & Isolation of responsibility. Багато модулів накопичують у собі багато відповідальності, через що часто зміни наростають в одній точці проекту та ускладнюється перевикористання;

– Unification & Sharing of responsibility. Багато модулів повторюють реалізацію тих самих аспектів, через що зростає деуніфованість логіки в кодобазі з ускладненням подальшого перевикористання;

– Grouping similar. Модулі, сильно пов'язані один з одним, лежать у різних місцях.

Усі перелічені вище проблеми мають схожу загальну рису: розсіювання відповідальності та її кордонів разом із порушенням доступу та управлінням

відповідальності у модулях з недостатнім розвитком проекту. Як правило, це виражається у кількості зв'язків, частоті змін та структурному розташуванні. Впливає це все, відповідно, на небезпеку змін, труднощі у перекористуванні та пошуку логіки в кодобазі.

Відповідальність складно формалізувати виміряти, але так чи інакше рішення можна реалізувати на основі:

– орієнтованого графа залежностей модулів та його юнітів;
– загальних архітектурних метрик (LCOM-like, Instability & Abstractness);
– граничних значень в ієрархії залежностей (за відповідальністю/абстрактностю/розміром).

1.2 Огляд існуючих аналогів

Перед розробкою власного рішення потрібно було дослідити вже існуючі альтернативи для вирішення поставлених проблем.

1.2.1 Інструменти рефакторингу IDE від JetBrains

Вбудовані в IDE від компанії JetBrains інструменти для розробки допомагають рефакторити тільки тривіальні ділянки коду, і можуть виявити лише найпростіші проблеми в коді. IDE від JetBrains вміє знаходити дублювання коду, але якщо змінити кілька символів - такий код JetBrains вже не вважатиметься дублюючим, хоча за структурою йде явне порушення принципів DRY.

1.2.2 SonarQube

Функціональність цього інструменту сконцентрована на синтаксичних помилках коду, можливих проблемах безпеки, типів та антипатернів у реалізації. І хоч реалізовано це для багатьох мов та на глибокому рівні, SonarQube все одно не вирішує проблеми поганої архітектури модулів.

1.2.3 TechDebt Rate / StepSize

Пропонує для команд цілу екосистему по роботі з технічним боргом проекту, де можна відзначати "погані" місця в кодовій базі проекту [12]. І хоч у рішення висока інтеграція з таск-трекерами, але StepSize теж не передбачає жодного аналізу та роботи з якістю декомпозиції та уніфікації модулів.

1.2.4 Метрики архітектури кодової бази

Для контролю якості коду є багато реалізацій лінтингу синтаксису, а також є низка фундаментальних метрик (наприклад LCOM, HoC, MMAC, SCOM, Instability & Abstractness), у тому числі архітектурних. Якісь з них більше сконцентровані на Coupling , якісь на Cohesion , але в цілому знову ж таки багато різних, але одного загальноприйнятого і скрізь використовуваного – ні [13,14]. У тому числі немає прикладних рекомендацій до рефакторингу.

1.2.5 Dependency Cruiser / ESLint

Обидва рішення є статистичні аналізатори коду, що допомагають відстежувати помилки JavaScript -коду до запуску самої програми. ESLint зав'язаний у реалізації на синтаксис мови [15], а Dependency Cruiser - більше зв'язку між модулями. Однак обидва ці інструменти вимагають ручного налаштування та реалізації правил, які частіше диктуються предметною областю проекту, ніж будь-якими загальними закономірностями та практиками. При цьому, як правило, на старті розробки проекту, у розробників недостатньо знань про ці правила, щоб явно вказати їх у лінтерах, крім загальновідомих правил

1.2.6 Github Copilot (OpenAI)

Розширення, що ґрунтується на штучному інтелекті. За коментарем у коді чи назві функції, допомагає написати реалізацію, на підставі нейромережі, навченої на відкритих прикладах коду [16]. Показує себе ефективно при написанні локальних ділянок коду, але складно масштабується при найпростішому ускладненні модульності, і допомагає контролювати достатній рівень якості архітектури коду.

1.2.7 Результати порівняння аналогів

В результаті дослідження аналогів стало зрозуміло, що інструменти є, проте кожен з них вирішує свою приватну проблему і немає загального рішення, яке на достатньому рівні контролювало б якість архітектури (табл. 1.1).

Таблиця 1.1 – Порівняльна таблиця аналогів

	Jetbrains IDE tools	Sonar Qube	StepSize/ TechDebt	Software Design metrics	ESLint / Dep.Cruiser	GitHub Copilot
Контроль якості архітектури	~	–	~	+	~	–
Загальноприйняті практики проектування	~	–	–	+	~	~
Практичні рекомендації	+	+	+	~	~	+
Адаптивність до масштабу проекту	–	–	~	+	–	–
Аналіз проекту	+	+	–	–	+	–
Розширюваність та налаштовуваність	–	–	+	–	+	–

Таким чином, постає проблема у розробці програмного засобу, який би ввібрав в себе всі переваги розглянутих аналогів.

2 ПОСТАНОВКА ЗАВДАНЬ І КОНЦЕПЦІЯ ІНСТРУМЕНТУ

2.1 Концепція програмного інструменту

2.1.1 Аспекти використання

Перш за все необхідно забезпечити максимальне врахування таких аспектів використання:

– врахування особливостей проекту. Головна проблема - це враховувати розвиток проекту в динаміці (як змінюється декомпозиція модулів, їх зв'язок у кодовій базі). При цьому важливо враховувати і поточний обсяг і складність проекту для пошуку покращень в архітектурі модулів;

– своєчасний і зрозумілий зворотній зв'язок. Також важливо дізнаватися про основні проблеми при проектуванні заздалегідь, а не на етапі код-рев'ю. А тому також важливо, щоб ці зауваження були зрозумілі, і розробник міг одразу виправити їх у кодовій базі проекту. При цьому у досвідчених розробників на проекті є й накопичена досвідом експертиза щодо проекту. Приблизно того ж ефекту можна домогтися при статичному аналізі зв'язків модулів;

– налаштовуваність рішення. Враховуючи всю складність завдань інструменту, важливо, щоб користувач мав можливість налаштовувати особливості його роботи під себе та команду (суворість, чутливість декомпозиції та інші аспекти роботи алгоритму);

– простота та ефективність у використанні. Вкрай важливо, щоб інструмент не був надмірно складним у використанні та внутрішній будові. В рамках кваліфікаційної роботи можна обмежитися форматом статичного аналізу, без додавання надмірно допоміжних асистентів та інтеграцій у IDE. Інструмент не повинен прискіпливо змушувати прагнути до еталонності, якої дуже складно досягти в корпоративних умовах. Він має допомагати покращувати рівно настільки, наскільки це прискорить розробку та покращить комунікацію в

команді.

2.1.2 Вимоги до рішення

З урахуванням аспектів використання, розпишемо вимоги до рішення, що розробляється:

– customizing. Можливість налаштовувати гіперпараметри алгоритму, для уточнення роботи його логіки та суворості, а також розширюваність плагінами;

– quick feedback. Швидкий цикл зворотного зв'язку при використанні інструменту (що раніше користувач дізнається про потенційні проблеми в кодовій базі – тим краще);

– easy, effective, reliable. Інструмент не повинен бути надто нав'язливим у допомозі. При цьому, повинен показувати свою ефективність і простоту, щоб будь-який користувач міг довірити кодову базу проекту;

– metrics dased. Алгоритм повинен враховувати відомі метрики архітектури модулів (LCOM, Instability & Abstractness, Unification & Sharing, Decomposition & Isolation);

– justified enforcing. Алгоритм повинен спонукати до виправданого і погіршує використання кращих практик, без абстрактних “добре - погано” і прагнення недосяжної еталонності.

2.2 Архітектура рішення

2.2.1 Схема використання

Враховуючи всі вимоги та сценарії використання, було складено верхньорівневу архітектуру рішення (рис. 2.1):

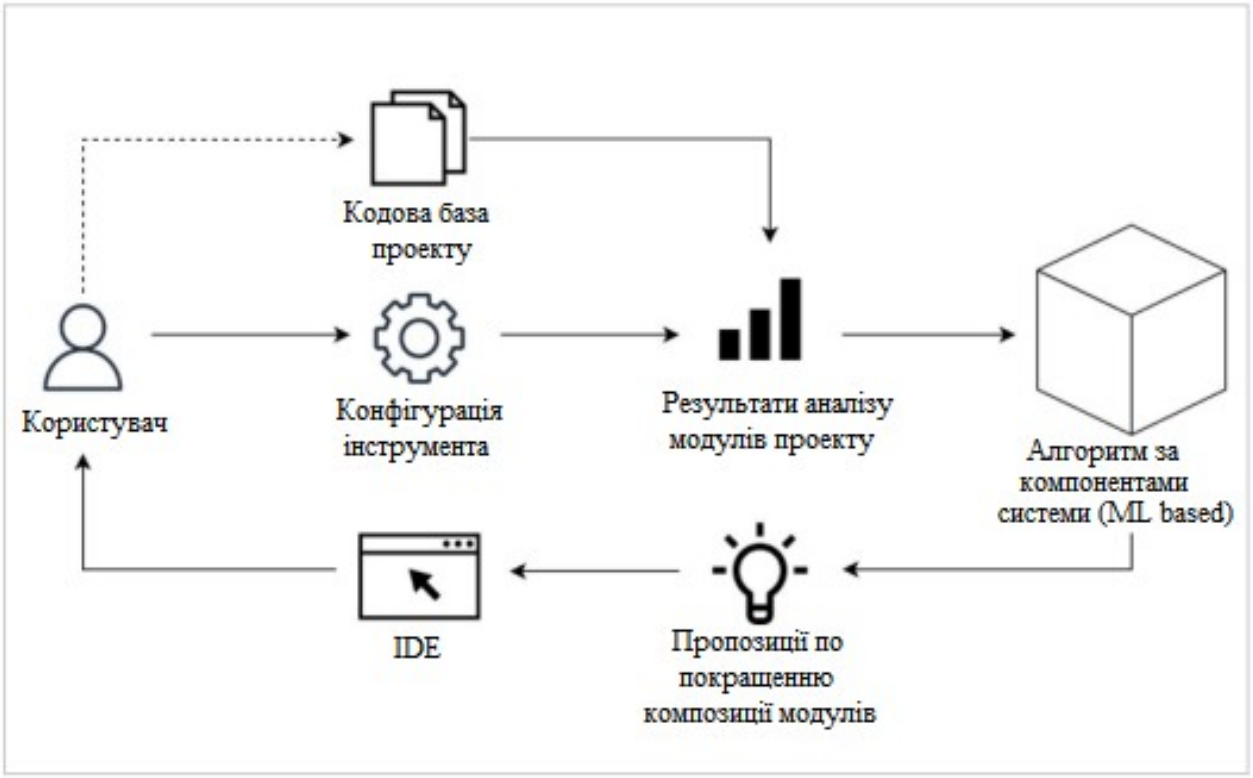

Рисунок 2.1 – Архітектура рішення під час використання

2.2.2 Архітектурні компоненти

Концептуально архітектура програмного інструменту містить такі складові частини:

– UserConfig - конфігурація користувача налаштувань для інструмента;

– Analyzer - парсинг та аналіз модулів з підрахунком метрик;

– Clusterizer - підготовка та кластеризація модулів;

– Feedback - пошук архітектурних проблем та донесення зворотного зв'язку.

Користувач задає гіперпараметри алгоритму, його строгість, чутливість та глибину аналізу.

Модуль Analyzer проводить парсинг кодової бази проекту, аналізує модулі та їх зв'язки, а також нормалізує до загальної структури даних. Крім цього модуль

надає методи для підрахунку архітектурних метрик для подальшої обробки.

Модуль Clusterizer готує датасет ознак, спираючись на підраховані метрики для модулів та проводить кластеризацію.

Модуль Feedback знаходить основні проблеми, спираючись на результати кластеризації та доносить до користувача у зрозумілому та людиночитаному форматі.

Користувач отримує зрозумілий зворотний зв'язок із проблемними модулями з погляду архітектури та приймає рішення, що з ними робити далі.

2.2.3 Технологічний стек

Стек: TypeScript.

Бібліотеки машинного навчання: density-clustering.

Допоміжні бібліотеки: lodash.

Утиліти для розробки: ts-node, ts-node-dev, eslint, cross-env.

Інфраструктура: npm, github.

Візуалізація результатів: CSS, HTML, chartjs.

2.3 План робіт

Оформлені вимоги та архітектура дозволяють спланувати подальшу роботу над проектом та оцінити основні ризики.

2.3.1 Список завдань

Зібрати базовий прототип рішення на підтвердження гіпотези.

Перевірити дієвість прототипу та зафіксувати результати.

Дослідити ідеї для покращення інструменту в аспектах аналізу та пропонованого рефакторингу декомпозиції модулів.

Ітеративно покращувати інструмент до виправлення критичних помилок.

Перевірити розроблене рішення на інших доступних проектах та зафіксувати результати.

2.3.2 Ризики та аспекти

Складно враховувати мінливість проекту з його історією, складністю та розміром команди (використовувати git - надмірно, можуть допомогти і налаштування користувача).

Важко перевіряти ефективність програми без великої вибірки проектів і команд (на локальних прикладах, але цього потрібні конкретні критерії).

Усі аспекти логіки алгоритму мають бути добре розділені, для можливості налаштовувати та розширювати поведінку під себе.

Аналізу імпортів може не вистачити, можливо, доведеться аналізувати зв'язність на більш глибокому рівні.

Зауваження можуть бути нав'язливими для даної стадії розвитку проекту, прагнення еталонності.

Важко виміряти рівень відповідальності модулів для правильного розподілу модулів.

3 РЕАЛІЗАЦІЯ ТА ВПРОВАДЖЕННЯ РІШЕННЯ

3.1 Парсинг кодової бази

Для проведення аналізу мало знати існуючі файли у проекті та їх розташування у структурі. Крім цього необхідно знати і залежність з-поміж них, тобто від яких інших модулів залежить кожен файл. Робота алгоритму починається із цього етапу.

Було вирішено скористатися готовими рішеннями: madge [11] для парсингу файлів та їх зв'язків та graphviz [17] для візуалізації графа файлів (рис. 3.1). В результаті роботи цих інструментів виходить json -файл зі зв'язками, який вже буде оброблятися іншими модулями розробленого інструменту.

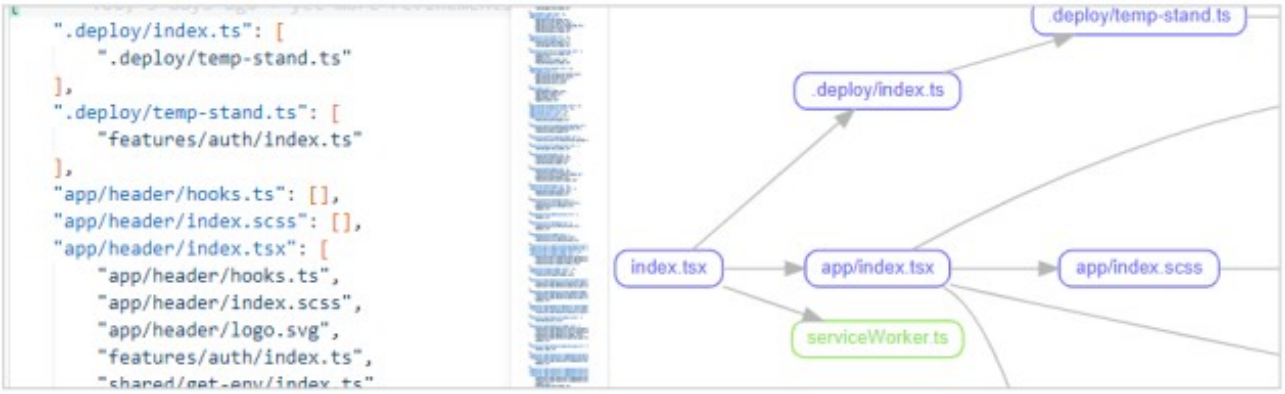

Рисунок 3.1 – Граф залежностей тестового проекту

Вже під час розробки було помічено, що етап парсингу є одним із найважливіших, оскільки якщо при парсингу станеться якась помилка, вона вплине надалі і на решту алгоритму і його точність. Наприклад, були проблеми при вказівці відносних шляхів, що змушувало писати обхідні рішення, хоча проблема була у парсингу. Також, якщо намагатися збагатити граф імпортів після парсингу, то початкова неповнота графа теж дасть себе знати (наприклад під час розробки, коли потрібно було явно вказати зв'язок між модулями, хоча в графі її не було). При цьому початковий граф мав на увазі зв'язок між будь-якими

файлами, статичними файлами стилів/картинок - хоча при аналізі такі повинні ігноруватися.

3.2 Реалізація модуля Analyzer

Наступний компонент в архітектурі інструмента – модуль Analyzer. Він повинен виявити архітектурні ознаки для кожного модуля для подальшої кластеризації в модулі Clusterizer.

3.2.1 Теоретична база для розрахунків

Для розрахунків ознак було вирішено використати архітектурні метрики. Але ключова проблема в тому, що всі відомі метрики (LCOM, MMAC, NHD, SCOM і т.п.) - слабо поєднуються одна з одною, і в цілому немає загальноприйнятих з них. При цьому вони враховують лише зв'язність модулів, але не їх складність, абстрактність реалізації та стійкість до змін. Під такі вимоги підходили лише метрики Instability & Abstractness, які докладно викладені у книзі Роберта Мартіна “Чиста Архітектура” [6,18] з обгрунтуванням прикладної розробки у компаніях та командах. Ці метрики і було вирішено задіяти у роботі алгоритму.

Роберт Мартін приходить до необхідності запровадити принципи стійкості та абстрактності, спираючись на реальні проблеми в розробці (рис. 3.2):

– мінливість модулів;

– складності з інтеграцією версій у команді;

– швидкість та зручність викочування нових фіч з перевикористанням загальної кодової бази.

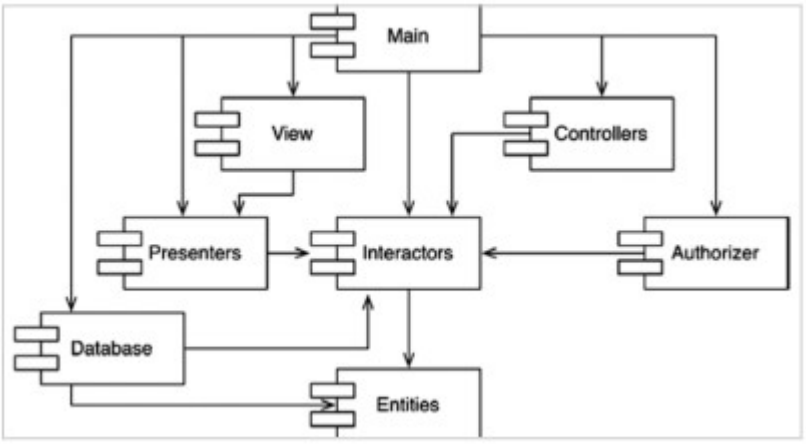

Рисунок 3.2 – Діаграма компонентів, «Чиста архітектура»

Спираючись на ці міркування можна визначити поняття стійкості і абстрактності модулів в архітектурі.

Стійкість - здатність [модуля] зберігати свій стан, при зовнішніх впливах [від нових бізнес-вимог].

Абстрактність - високорівневість правил та рішень, закладених у модуль.

Далі Роберт Мартін запроваджує принципи стійкості та абстрактності, щоб формалізувати критерії для подальшого підрахунку цих метрик.

Принцип стійкості - модуль стає стійким (рис. 3.3), коли від нього залежать кілька інших модулів, при цьому сам він залежить від більш стійких модулів, або не залежить ні від кого (в ідеальній системі (рис. 3.4) повинні бути як стійкі, так і нестійкі модулі, щоб залишалася можливість вносити зміни до кодобази).

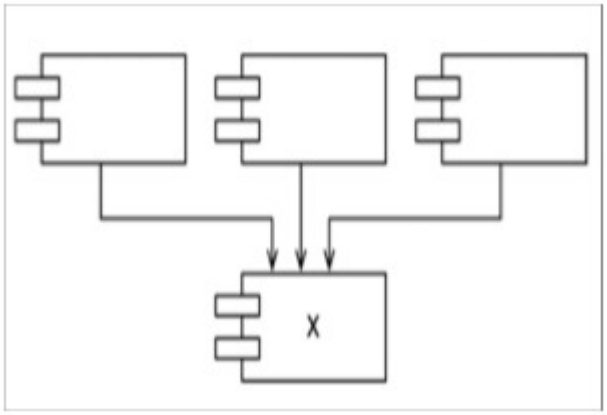

Рисунок 3.3 – Стійкий компонент

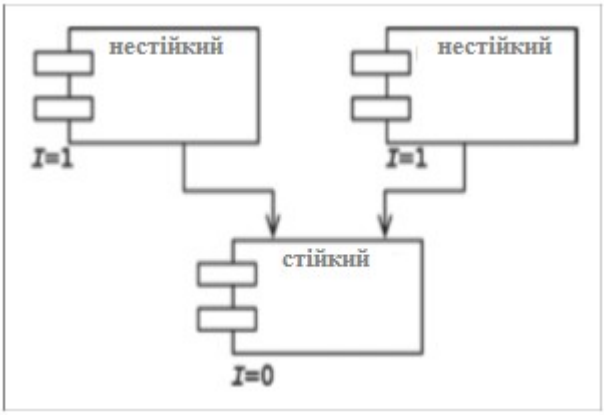

Рисунок 3.4 – Ідеальна організація системи

Принцип абстрактності – абстрактність пропорційна стійкості – тобто, високорівневі правила мають бути закладені у стійкі модулі, щоб їх було складніше змінювати (в ідеальній системі повинні залишатися як абстрактні, так і конкретні, щоб залишалася можливість вносити зміни на різних рівнях абстрактності).

Подальші міркування Роберта Мартіна зводяться до вивчення відносин між цими двома метриками, для аналізу архітектурних проблем у кодовій базі (рис. 3.5). Але в рамках роботи зупинятись на цьому не будемо. Важливо те, що вже є давно вивчені метрики, які можуть бути фундаментом для роботи алгоритму для дотримання загальних вимог до інструменту.

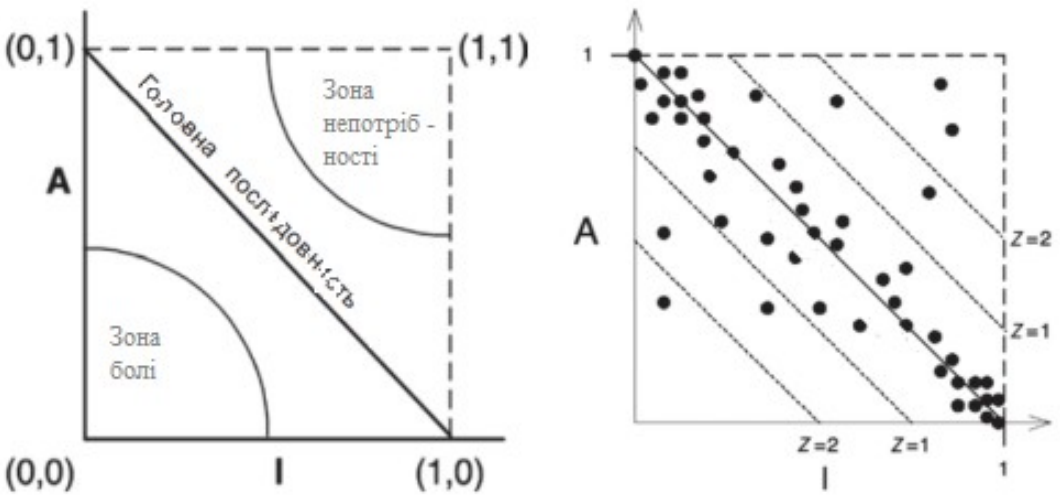

Рисунок 3.5 – «Зони виключення» і «Діаграма розсіювання компонентів», графіку відношення Стійкості (I) від Абстрактності (A)

3.2.2 Допоміжний модуль fs та клас Project

Кожна з метрик хоч і має свої нюанси та труднощі у реалізації, але все одно вимагає більш багатогранної інформації про зв'язність модулів. Простого графа не вистачить, потрібно мати доступ і до загального переліку модулів (з визначенням кордонів кожного модуля), і до дерева файлової структури, і до модульного графа в цілому. Для цього потрібно створити не просто структуру даних, але повноцінний клас Project (див. Додаток А), зі своєю мінімальною поведінкою для зовнішнього використання.

Спочатку модуль analyzer/fs містив лише допоміжні утиліти для перетворень графа зв'язків проекту. Однак при рефакторингу інструменту стало помітно, що одна і та ж логіка по роботі з графом зв'язків починає деуніфікуватися в проекті. Тому був необхідний доступ до інстансу проекту, з повною інформацією про зв'язки модулів у різних структури даних. Цей інстанс дозволить спростити підрахунок метрик, а також їх подальшу кластеризацію у модулі Clusterizer.

Розглянемо інтерфейс класу Project докладніше:

– imports: Map<string, any> - початковий граф зв'язків, отримуваний у процесі парсингу. У процесі обробки не змінюється, крім фільтрації сторонніх файлів (наприклад файли стилів, статики тощо. - залежно від налаштувань користувача);

– files: string[] - загальний перелік файлів проекту;

– structure: Map<string, any> - дерево файлової структури, побудоване виходячи з графа зв'язків;

– modules: string[] - узагальнені модулі проекту, побудовані на основі структури проекту для більш високорівневого аналізу (наприклад, файли components/button/button.tsx та components/button/style.css складаються в модуль

components/button);

– modulesGraph: Map<string, string[]>- узагальнений граф залежностей лише на рівні модулів, побудований на початковому графі файлів.

3.2.3 Підрахунок метрики Instability

Звернемося знову до “Чистої архітектури” Роберта Мартіна, де він запроваджує метрику Instability, яка обчислюється за формулою (3.1):

$$I = \frac{out}{¿ + out}, \tag{3.1}$$

де *out* - кількість залежностей від зовнішніх модулів, *in* - кількість модулів, що залежать від поточного.

Якщо I=0, отже компонент максимально стійкий, оскільки не залежить від жодних інших модулів

Якщо I=1, отже максимально нестійкий залежить тільки від інших, при цьому від нього не залежить жоден інший модуль.

Що стосується розроблюваного рішення, для файлів це обчислюється без особливих труднощів (Додаток Б). Результати представлені на рис. 3.6 (out – кількість прямих залежностей файлу у графі імпортів, in - кількість файлів з поточним файлом у залежностях):

Рисунок 3.6 – Результат підрахунку Instability для модулів

3.2.4 Підрахунок метрики Abstractness

Роберт Мартін вводить також визначення та заходи абстрактності.

Міра абстрактності модуля (Abstractness) - ставлення абстрактних юнітів (абстрактні класи, інтерфейси) до загальної кількості юнітів у компоненті, обчислюється за формулою (3.3):

$$A=\frac{N_a}{N_c}, \tag{3.3}$$

де N_a – число абстрактних юнітів у компоненті (абстрактні класи, інтерфейси), N_c-загальна кількість юнітів у компоненті.

Якщо A=0, отже компонент абсолютно “конкретний у реалізації”, так як у ньому відсутні абстрактні юніти.

Якщо A=1, отже компонент абсолютно “абстрактний”, оскільки повністю складається із абстрактних юнітів.

Проте з моменту написання книги минуло багато років і час робить свої корективи. Наприклад, поточна розробка будується далеко не тільки на абстрактних класах та інтерфейсах (особливо у світі javascript -проектів). У наші

дні модулі можуть бути різного ступеня абстрактності (рис. 3.7):

– helpers/components - абстрактні службові утиліти/компоненти з UIKit, які нічого не знають про предметну область проекту;

– entities – базова доменна логіка сутностей проекту, що використовують під собою службову логіку;

– features/usecases - базові сценарії користувача, що регулярно змінюються з новими бізнес-вимогами;

– widgets/pages - конкретні сторінки та віджети, які постійно змінюються з новими завданнями.

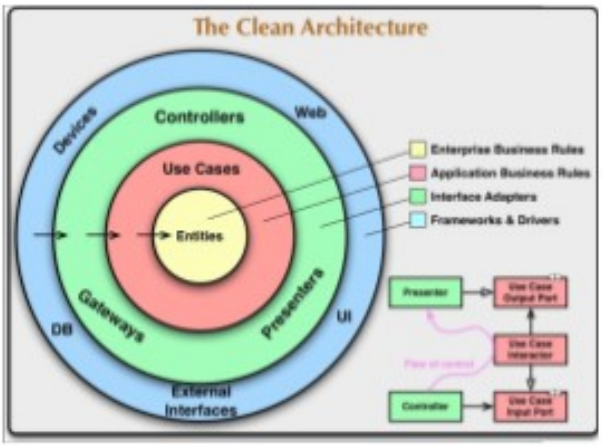

Рисунок 3.7 – Абстрактність модулів

Все це підводить до того, що абстрактність у наші дні вважається не так кількістю абстрактних юнітом, як “конкретністю реалізації” у загальній структурі модулів. Тобто ступенем реалізації інших абстрактних контрактів та ступенем оголошення контрактів для використання у більш високорівневих модулях. До цього ж висновку приходить автор і в [19] (рис. 3.8)

Рисунок 3.8 – Переосмислення міри абстрактності

Цей же поділ на абстрактність ще більш помітно у світі Frontend -розробки, наприклад, у контексті методології Atomic Design [7]. На рис. 3.9 кожен наступний шар відповідальності використовує під собою попередній та уточнює його контракт для використання у шарах вище:

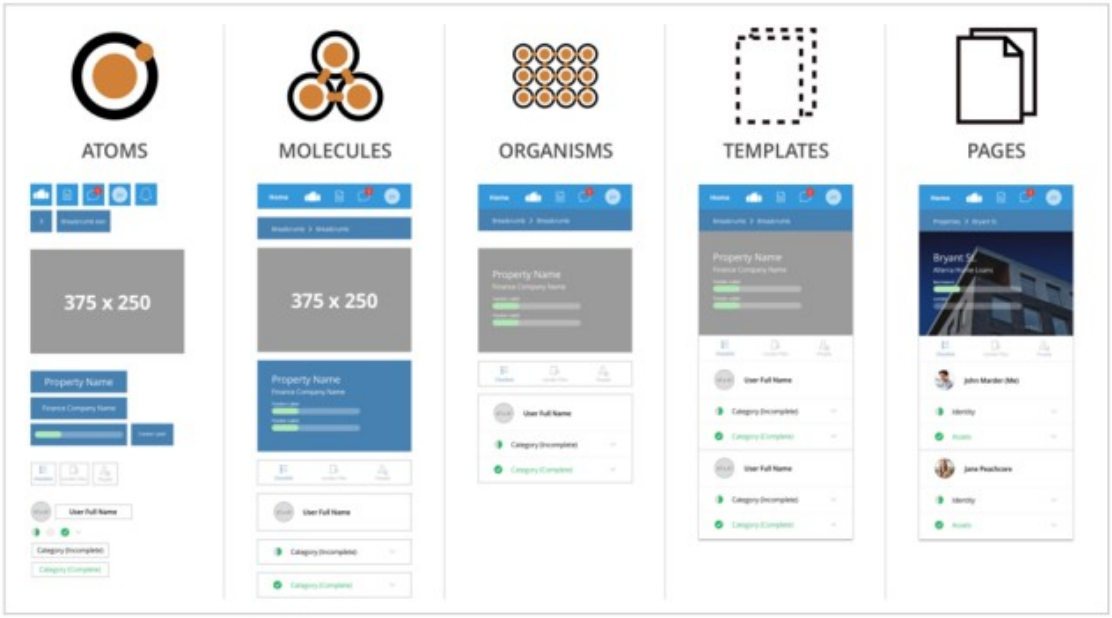

Рисунок 3.9 – Методологія Atomic Design

У випадку з інструментом, що розробляється, можна за аналогією підрахувати ступені абстрактності кожного конкретного модуля в загальному графі. Причому чим “нижче” розташований модуль у тому графі, тим паче він абстрактний, так як його використовують усі вище розташовані модулі (рис. 3.10).

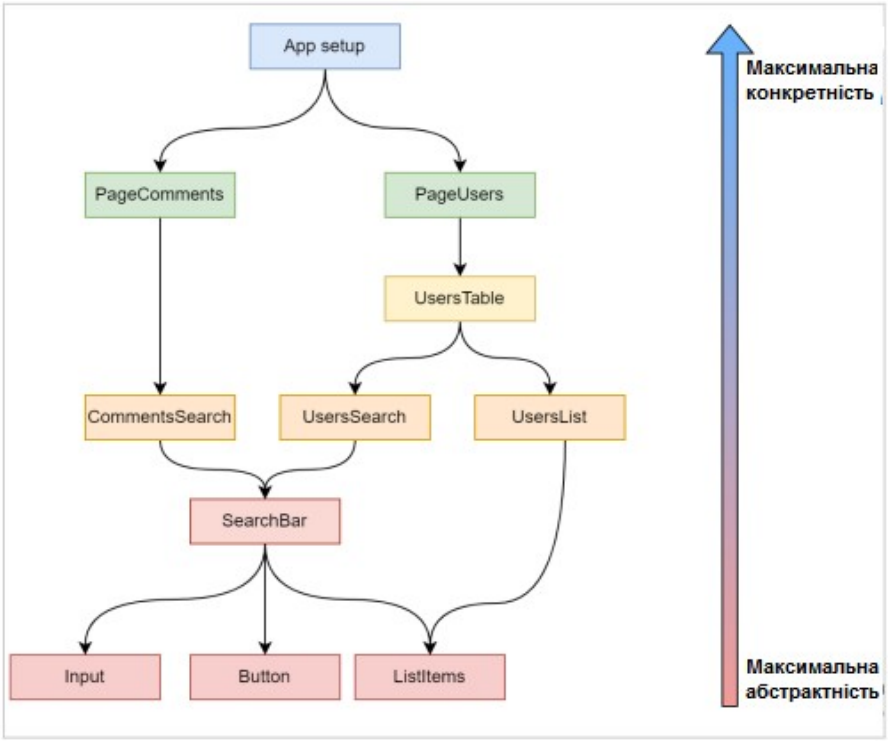

Рисунок 3.10 – Модульний граф, розділений за рівнями абстрактності

І хоч на спрощеному прикладі з “витягнутим графом” все зрозуміло, але навіть на ньому видно певні складнощі для реалізації в алгоритмі:

– важко однозначно визначити порядок модуля у загальному ланцюжку залежностей. Наприклад, SearchBar модуль використовується відразу у двох модулях, і незрозуміло по якому з ланцюжків проводить розрахунок нагору. Також для модуля UsersTaЬIe, який залежить від двох інших;

– не завжди є повні дані про зв'язки між модулями. У реальному житті деякі ділянки графа не будуть настільки заповненими, оскільки проекти постійно розвиваються та нові частини пишуться з нуля. А це означає, що далеко не для всіх модулів ми можемо визначити, хто з них “абстрактніший”. Це один із прикладів частково впорядкованої множини, де ми не можемо з абсолютною точністю судити про порядок елемента в загальній множині, оскільки множина зв'язків між ними обмежена. Тому ми можемо лише частково впорядкувати її, але не повністю;

– складно однозначно визначити межі модуля у файловій системі. Як зрозуміти, що components/button/button.tsx відноситься до модуля components,

якщо в модулі немає index -файлу і в батьківській директорії модулів містяться хелпери, які будуть збивати з пантелику при визначенні?

Всі ці нюанси призводять до того, що буде важко однозначно визначити абстрактність модулів, але можна скласти базовий алгоритм, який працюватиме в більшості випадків! А зазначені невизначеності можна виразити в гіперпараметрах, які задаватимуться користувачем. При цьому важливо не зав'язуватися сильно на "використовуваності" модулів, оскільки для цього вже є метрика Instability (а ці метрики мають бути найменш залежними один від одного).

Для підрахунку абстрактностей знадобиться спеціальна структура даних, за допомогою якої можна підрахувати порядок абстрактності модуля у множині модулів проекту. Для цього скористаємось нашим допоміжним класом Project, де реалізуємо підрахунок ступеня абстрактності для кожного модуля у методі getModulesWeights.

Під час розробки перевірялися різні варіанти упорядкування графа модулів, але кожен із них містив у собі багато недоліків. Було вирішено зупинитися на спрощеному алгоритмі, який полягає в ітеративному розгортанні графа залежностей до повного списку залежностей кожного модуля і їх підрахунку. З таким спрощеним підходом ступінь абстрактності визначається хоч і не з повною точністю, але достатньо для інструменту, що розробляється. При цьому, глибина розгортання може задаватися користувачем (бо це впливає і на точність, і на швидкодію інструменту).

Ці ваги дозволяють визначити міру конкретності для модуля як (3.4):

$$C' = \frac{w}{w_{max}}, \tag{3.4}$$

де w - вага конкретності для конкретного модуля (тобто число його залежностей у розгорнутому графі), w_{max} — максимальне значення конкретності у графі модулів.

Якщо C'=0, то модуль максимально абстрактний, так як не залежить від

жодних більш низькорівневих модулів у розгорнутому графі.

Якщо C'=1, отже модуль максимально конкретний, оскільки містить найбільшу кількість залежностей у розгорнутому графі.

Відповідно, міра абстрактності обчислюватиметься за формулою (3.5):

$$A' = 1 - C', \tag{3.5}$$

де C' - міра конкретності модуля

Формула має ряд недоліків, але в контексті інструменту її буде достатньо для подальшої кластеризації та обробки (рис. 3.11).

```
.deploy: 0.85
app: 0
features/auth: 0.9
features/error: 0.9
features/hero-sheet: 1
features/origin: 1
features/repo-details: 0.9
features/repo-explorer: 0.85
features/repo-list: 0
features/search: 0
features/user-info: 0.85
index.tsx: 0
models.gen.ts: 1
models.ts: 0.95
pages: 0
serviceWorker.ts: 1
shared/components/card: 0
shared/components/org: 0
```

Рисунок 3.11 – Результат підрахунку Abstractness для модулів

3.3 Реалізація модуля Clusterizer

3.3.1 Вибір механізмів машинного навчання

Якщо поглянути на спрощену карту машинного навчання (рис. 3.12), стає зрозумілим, що не підходять рішення з ускладнених категорій “Навчання з підкріпленням”, “Нейромережі” і “Ансамблеві методи”, тому що ми оперуємо найпростішими ознаками без заготовленої навчальної вибірки. Тому з “Класичного навчання” підходять лише механізми “Без вчителя”, саме:

– кластеризація;
– асоціація;
– зменшення розмірності (узагальнення).

Рисунок 3.12 – Спрощена схема класичного навчання

Розглянемо можливе використання цих підходів:

– Association (збір даних про модулі) - для виділення зв'язків між модулями, і визначення неявних зв'язків за модулями, що часто використовуються разом;

– Generalization (підготовка даних про модулі) - узагальнення даних з графа імпортів, метрик та асоціації для подальшої кластеризації;

– Clusterization (ядро алгоритму) - для виділення груп модулів на підставі заданих ознак та функції відстані (наприклад, кількість зовнішніх / внутрішніх залежностей та глибина у файловій ієрархії як features, і ступінь стійкості/абстрактності як distance).

Таким чином, кластерний аналіз дозволить на підставі множини ознак (features) модулів, виявити їхнє оптимальне розташування в архітектурі на підставі Instability & Abstractness метрик (distance function). Це дозволить зберегти баланс стійкості та абстрактності при декомпозиції відповідальності в модульній архітектурі (в майбутньому для збагачення даних можна впровадити і механізми сімейства Association та Generalization*).*

Розглянемо алгоритми кластеризації для впровадження в ядро алгоритму:

– K-Means - випадковий розкид точок кластерів із центроїдами не зовсім підійде, ми не можемо бути впевнені, що отримані features утворюватимуть кругоподібні кластери - при яких і добре діє цей алгоритм [20];

– C-Means - на відміну K-Means враховує ступінь/імовірність приналежності кластеру, але з тих же причин [20];

– DBScan - якраз будує кластери на підставі "найближчого сусідства" при різних формах кластерів, що якраз більше підходить для інструменту, що розробляється, хоч і працює більш загально і універсально [20].

3.3.2 Реалізація кластеризації модулів

Щоб уникнути "зоопарку технологій" для простого інструменту, було вирішено реалізовувати модуль кластеризації теж на TypeScript, без python.

Як пам'ятаємо, кожна точка в отриманому датасеті складається з двох ознак – Instability та Abstractness, що лежать в інтервалі [0; 1]. Отже, відстань між точками визначення кластера буде обчислюватися за формулою (3.6):

$$dist(i,j)=d(I,A)=\sqrt{(I_i-I_j)^2+(A_i-A_j)^2}, \quad (3.6)$$

де *d(I,A)* - Евклідова відстань між метриками I, A для двох точок.

Надалі цю формулу можна доповнити новими ознаками та ускладнити функцію відстані. Оскільки функція відстані загальноприйнята, було вирішено задіяти npm -бібліотеку “density-clustering” [21] для використання методу dbscan у контексті інструмента, що розробляється.

Спочатку підготуємо датасет на основі метрик Instability & Abstractness, потім отримаємо угруповання за кластерами для наших модулів (див. Додаток Б), яке можна буде використовувати надалі для пошуку проблем в архітектурі (рис. 3.13).

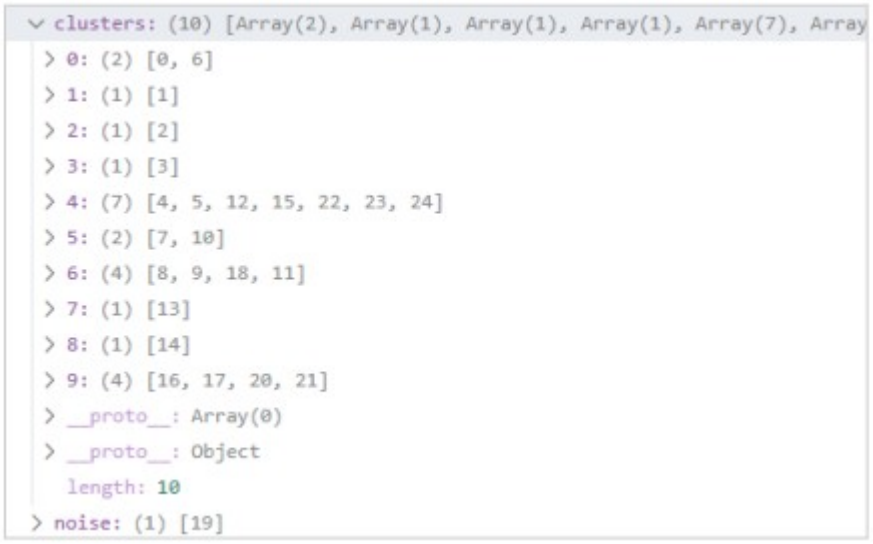

Рисунок 3.13 – Результат кластеризації модулів

3.4 Реалізація модуля зворотного зв'язку для користувача

Далі було необхідно адаптувати інструмент для зовнішнього використання в проєктах. Користувач повинен розуміти, що робити з результатами цієї кластеризації, бажано у візуальному простому вигляді.

Пошук проблем спираючись на результати кластеризації Результати кластеризації - це те, як мали б бути розташовані модулі "при ідеальному розкладі". При цьому є і фактична відстань між модулями у файловій структурі (звідси наприклад, беруться відносні імпорти, коли ми намагаємося дістатися до модуля вище за батьківську директорію).

Це означає, що в рамках одного кластера ми можемо визначити, які модулі найбільш віддалені від інших, хоча за метриками Instability & Abstractness вони повинні знаходитися поруч у файловій структурі. Ми не можемо сказати гарантовано, чи варто їх розташовувати поруч, але підсвітити користувачеві, щоб він вирішив сам - цілком. Для цієї логіки потрібно реалізувати:

– службову утиліту getFSDist для розрахунку відстані між двома модулями;

– функцію findClusterIssues для пошуку проблем у межах одного кластера;

– функцію-оболонку findProjectIssues , яка ітеративно збирає проблеми для кожного кластера і виводить результат аналізу у зручному для людини вигляді.

Візуалізація результатів. Для спрощеного розуміння результатів було зверстано базовий UI-дашборд за допомогою бібліотеки chartjs [22], що генерується в результаті роботи інструменту (надалі може використовуватися в CI/CD у корпоративній розробці).

Було проблемуно помістити на дашборд усі дані, що хотілося б, оскільки це відразу перевантажувало б інтерфейс та ускладнювало використання. Тому було залишено лише основне:

– відображення кластерів у модульній сітці ознак Instability & Abstractness;

– інформація про знайдені проблемні модулі, що потребують уваги;

– інформація про проігноровані модулі, які потрапили в шум при кластеризації.

На дашборд можна додати інтерактивні елементи для зміни опцій інструменту, але все це в планах на майбутній розвиток інструменту.

3.5 Реалізація налаштувань користувача

Оскільки в алгоритмі є багато невизначеностей, і немає точних знань про проект, він повинен бути налаштовуємо зовні опціями користувача. Для цього потрібно було адаптувати алгоритм і подекуди відрефакторити цілі модулі. Фрагмент програмного коду наведено на рис. 3.14.

```
{
    /** Стратегія аналізу: за модулями/за файлами */
    "strategy": "files",
    /** Опції аналізу (фільтрації файлів, глибина розгортання графа для
    підрахунку абстрактності */
    "analyzer": {
        "extensions": ["tsx", "ts", "jsx", "js"],
        "abstractnessDepth": 3
    },
    /** Опції кластеризації (число найближчих сусідів і радіус пошуку) */
    "clustering": {
        "neighNum": 1,
        "neighRadius": 0.05
    }
}
```

Рисунок 3.14 – Адаптація алгоритму

Оскільки в контексті аналізу ми працюємо одночасно і з файлами та

узагальненими модулями, треба точно знати, що використовувати для кластеризації та підрахунку метрик. Тому для опція strategy значення за замовчуванням: “modules ”.

Оскільки в проектах зустрічаються різні файли, потрібно мати можливість задавати фільтр для видів видів файлів, що цікавлять користувача. В рамках аналізу, користувачеві навряд чи цікаві файли стилів/статики тощо, тому потрібно мати можливість задавати такий фільтр зовні. Для опція analyzer.extensions значення за замовчуванням: [“tsx”, “ts”, “jsx”, “js”].

Оскільки для наближеного підрахунку абстрактності, ми вибрали розгортання графа модулів, це може досить сильно позначитися на швидкодії інструменту (особливо у великих проектах). Тому користувач може сам вибирати, наскільки глибоко треба розгортати цей граф для свого проекту (що позначиться на точності, але кожен зможе вибрати відповідно до своєї ситуації). Опція analyzer.abstractnessDepth, значення за замовчуванням: 5.

Оскільки кластеризація нічого не знає про контекст аналізу, а оперує лише датасетом ознак, опції пошуку сусідів повинні задаватися ззовні. Так, існують евристичні способи автоматично оцінити ці значення, але так чи інакше точніше буде результат при ручному налаштуванні користувачем. Для опції dustermg.neighNum, dustermg.neighRadшs значення за замовчуванням: neighNum=3, neighRadius=0.2.

3.6 Тестування

3.6.1 Підготовка тестових даних

Для фіксації ефективності алгоритму необхідно було регулярно тестувати інструмент на різних тестових даних і проєктах (рис. 3.15). Для цього було взято

особисті проєкти, а також ті, що були у вільному доступі.

```
import PRESET_GH_FSD from "./gh-fsd.imports.json";
import PRESET_GH_FDD from "./gh-fdd.imports.json";
import PRESET_GH_FDD_SPEC from "./gh-fdd-spec.imports.json";
import PRESET_GH_FDD_APP from "./gh-fdd-app.imports.json";
import PRESET_FAVEIN from "./favein.imports.json";
```

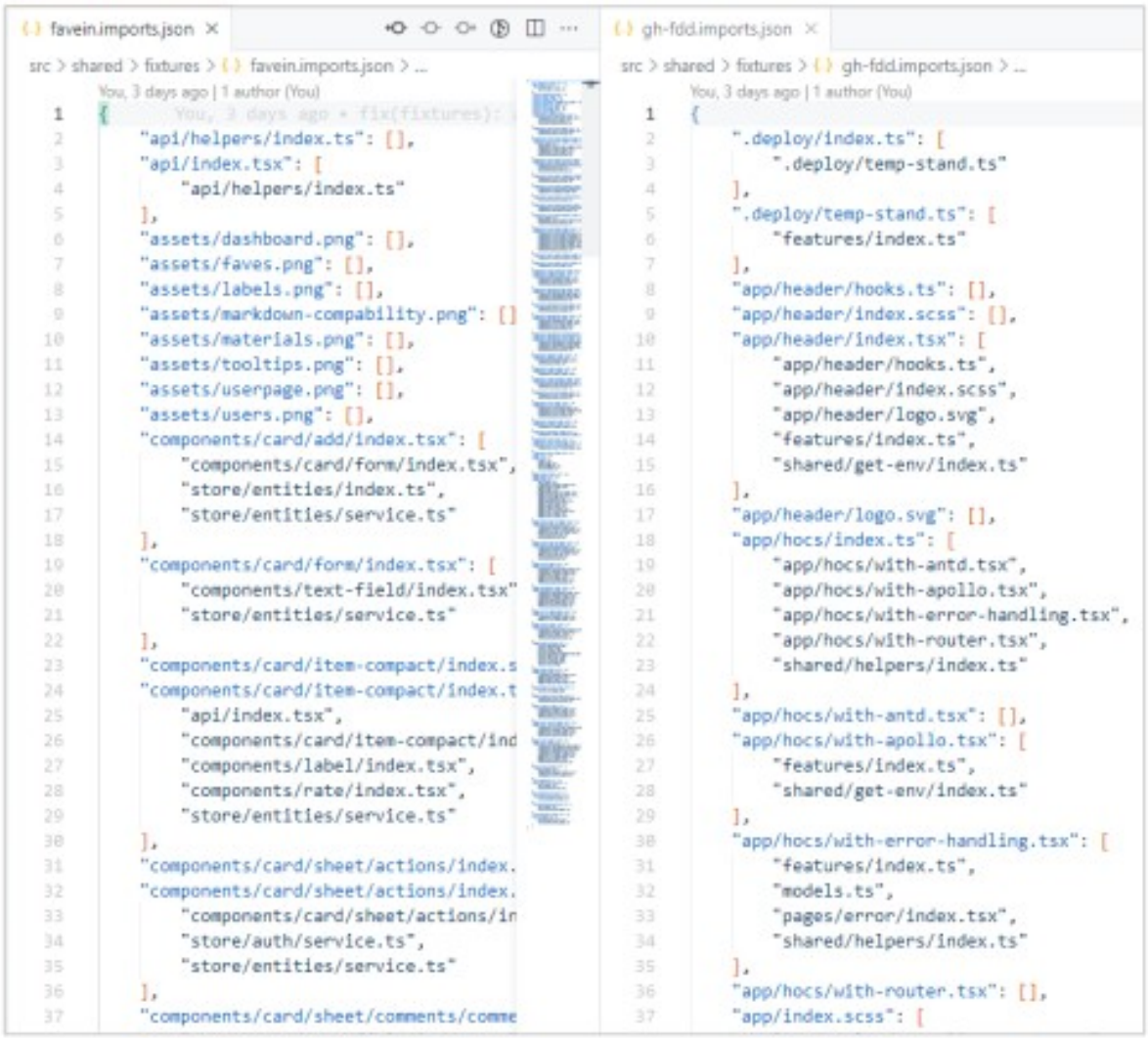

Рисунок 3.15 – Приклади тестових даних із різних проєктів

3.6.2 Тестування рішення та пошук помилок

Спочатку рішення тестувалося на одному проєкті, з різними

гіперпараметрами (рис. 3.16), для рівномірного розподілу кластерів. У процесі тестування допрацьовувалися аспекти візуалізації та алгоритму загалом. Також був використаний npm-пакет ts-node-dev, для запуску watch-режиму за зміненими файлами для більш гнучкого та швидкого налагодження.

Під час перших проб вже було видно підтвердження гіпотези, але при цьому були помічені деякі помилки та неточності. У тому числі алгоритм випробували і на інших проєктах, де результат був приблизно схожий. Ці неточності дозволили ітеративно покращити алгоритм, але тестування не припинялося, щоб знайти нові аспекти поліпшення рішення.

Рисунок 3.16 – Тестування інструменту для різних гіперпараметрів алгоритму

Деякі проблеми зі знайдених під час тестування:

– багато модулів хоч і згруповані за кластеризацією, але за структурою знаходяться зовсім у різних місцях (потрібно враховувати початковий розподіл у файловій структурі);

– деякі модулі часто використовуються разом, чого не видно на графіку (треба навчити алгоритм визначати це);

– багато модулів мають одні й самі значення Instability & Abstractness (необхідно додати розкид за допомогою обліку додаткових знань про модулі);

– деякі модулі мають прикордонні значення Instability & Abstractness (потрібно перевірити та дізнатися, чому і як часто вираховуються ці граничні значення, у тому числі додати фільтрацію файлів для обробки, див. рис. 3.17);

– на дашборді багато різної інформації, що може збивати з пантелику при використанні (бхідно пропрацювати UX/UI сторінки та залишити тільки найпотрібніше).

Рисунок 3.17 – Знайдений дефект прорахунку абстрактності модуля стилів

3.7 Використання

Розроблений інструмент дозволяє проаналізувати якість розподілу модулів з погляду архітектурних метрик з подальшою візуалізацією результатів. Для цього потрібно вміти встановлювати інструмент у JavaScript проект, з можливістю

задати налаштування користувача для використання. Тому було обрано реєстр пакетів npm: https://www.npmis.com/package/archpolisher (рис. 3.18)

Get Started

1. Install toolkit as dev-dependency

```
$ npm install -D archpolisher
# or by yarn
$ yarn add -D archpolisher
```

2. Setup user config `.archpolisherrc`

```
{
    /** Analysis strategy: by modules | files */
    "strategy": "files",
    /** Analysis options (files filter, module graph rollout depth) */
    "analyzer": {
        "extensions": ["tsx", "ts", "jsx", "js"],
        "abstractnessDepth": 3
    },
    /** Clustering options (neares neighbours num and neighbours nums) */
    "clustering": {
        "neighNum": 1,
        "neighRadius": 0.05
    }
}
```

3. Run toolkit

```
$ npm run archpolisher
# or by yarn
$ yarn archpolisher
```

Рисунок 3.18 – Інструкція зі встановлення інструменту

Після грамотного встановлення розробленого програмного інструменту у готовий проєкт можливе його подальше використання.

РОЗДІЛ 4. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Долікарська допомога при ураженні електричним струмом

Удар електричним струмом є поширеною травмою і часто може закінчитися летальним випадком. Ураження електричним струмом відбувається підчас контакту тіла людини з джерелом електричної енергії під напругою. Це реакція організму людини на проходження електричного струму через тіло. Вона проявляється по різному, від легких потрясінь до небезпечних здоров'ю травм, які можуть вплинути на тканини в організмі.

Шкода завдана електричним струмом залежать від кількох факторів: наскільки висока була напруга, яка область тіла була вражена і від виду струму. Фізичні наслідки для людини можуть коливатися від опіків частин тіла до серйозних вражень внутрішніх органів [23].

Часто люди стикаються з підвищеним ризиком ураження високою напругою. Низька напруги не наносить серйозних травм для люди, а з іншої сторони висока напруги, яка має більше 500 В, може призвести до серйозного пошкодження тканин. У дітей або підлітків при враженні електричним струмом в діапазоні від 110 до 200 В можуть вникнути значні травми. Зазвичай це трапляється підчас порушення техніки безпеки при роботі з електричними приладами, які є в побуті. Це можуть бити електричні шнури, подовжувачі, несправні розетки та багато чого іншого.

Виділяють чотири основні фактори від яких залежить вплив електричного струму на організм: величина струму, що протікає через тіло; органи, через які проходить струм; час, протягом якого струм вражає тіло; частота струму.

Фізичні наслідки на пряму залежать від величини струму, яка вражає тіло людини. Струм менший за 1 мА не завдає жодної шкоди людина фізичного ефекту. При 1 мА можуть відчуватися слабкі поколювання, а при 5 мА – легкий поштовх. Однак при струмі ведичною від 6 до 25 мА людина може відчувати больовий шок і деяку втрату контролю над м'язами.

При ураженні електричним струмом від 50 до 150 мА може спричинити у людини сильний біль, м’язове скорочення і навіть призвести до зупинку дихання. У певних випадках можливий летальний результат для людини. Струм від 1000 до 4300 мА призводить до ймовірної смерть, тому що дана напруга спричиняє пошкодження нервових зав’язків, м’язових скорочення та порушення ритму серця. До 10 000 мА електричний струм спричиняє важкі опіки шкіри людини та зупинку серця. Висока ймовірність смерті.

Найпоширеніші ознаки та симптоми враження електричним струмом включають:

– втрата свідомості;
– ускладнення або зупинка дихання;
– опіки, які виникають там, де струм входить і виходить з тіла;
– зупинка серця;
– слабкий і непостійний пульс або його припинення.

Перш за все, що потрібно зробити при першій допомозі, це відключити джерело живлення. Вимкнути електропостачання, від’єднати електроприлад від джерела електричного струму або вимкнути блок запобіжників, якщо він знаходиться неподалік. Не потрібно намагатися підходити близько до жертви, якщо не переконані, що це безпечно і живлення вимкнено.

Потрібно бути обережним у вологих місцях, тому що вода є електричним провідником і рятівник може стати теж жертвою. Якщо людина не впевнена щодо вологості поверхні, потрібно відключити основне електропостачання будинку. У випадку коли це неможливо, використати підручний предмет, який не є провідником і відокремити людину від джерела струму. Це може бути дерев’яна або пластмасова річ.

Після того як постраждалого відокремили від джерела електрики, потрібно викликати швидку і надати першу медичну допомогу. Далі потрібно визначити стан жертви. Перевірити, чи людина у свідомості і дихає. У складних випадках у жертви може бути слабкий пульс або його відсутність. Можливо, що дихання зупинилося. Якщо людина втратила свідомість і перестала дихати, потрібно

почати серцево-легеневу реанімацію. Руки розташовуємо в центрі грудної, одна на іншу. Сильно і швидко натиснути 30 раз приблизно до третини діаметра грудної клітки. Після кожного натискання на грудну клітку робиться два рятувальні вдихи. Потрібно відкинути голову потерпілого назад і підняти підборіддя. Затиснути ніс і створити повне ущільнення. Далі подути потерпілому в рот і подивитися, чи підніметься грудна клітка. Потрібно продовжувати робити натискань на грудну клітку та вдих, поки не прибуде медична допомога або людина не почне сама дихати. Якщо потерпілий живий, перемістити його у зручне йому положення подальше від небезпеки. Можна запобігти шоку, поклавши людину рівно на землю, з головою трохи нижче тіла [24].

Якщо людина при свідомості, нормально дихає і на тілі є опіки, потрібно накрити їх звичайною харчовою плівкою або іншою неклейкою пов’язкою, але без мазі чи лосьйону. Якщо кровотеча у потерпілого, може знадобитися компресія та джгут.

При роботі з соціальною мережею користувач повинен знати і вміти як правильно поводитися з ПК, тому що людина перебуває у непосередньому контакті з джерелом напруги. Удар електричним струмом є потенційно смертельною травмою. Негайна медична допомога важлива, щоб запобігти серйозним травмам і смерті. Для запобігання уражень електричним струмом при роботі за ПК слід встановити додаткові захисні пристрої, що забезпечують недоступність токопровідних частин для дотику. Для зменшення небезпеки використовувати розділовий трансформатор для розв'язки з основною мережею.

4.2 Вимоги ергономіки до організації робочого місця оператора ПК

Робоче місце – це ділянка простору, яка облаштована необхідним обладнанням, відповідно до трудової діяльності, для виконання поставлених завдань.

Правильно побудоване робоче місце повинне забезпечувати:

– найкраще розміщення обладнання і предметів праці;

– не допускати дискомфорту;

– підвищувати продуктивність праці;

– зменшувати втому працівника.

Розмір робочого місця повинен бути таким, щоб людина не виконувала лишніх рухів і не відчувала дискомфорту під час виконання роботи. Також для працівника важливо мати змогу змінити робочу позу, наприклад, положення тулуба, рук або ніг. Потрібно мінімізувати або звести до нуля всі незручності положення тіла [25].

Різні дослідження заявляють, що при правильному проектуванні робочого місця продуктивність людини може зрости від 15-25%. Такі фактори як рівень освітлення, вологість повітря, температура, шум, вібрація, токсичність, мають значний вплив на умови життєдіяльності і працездатності людини.

Антропометричні вимоги визначають відповідність робочого місця до фізіологічних параметрів тіла людини як зріст і розміри тіла. Індикатором цього є правильна робоча поза, відсутність дискомфорту, оптимальні зони досягнення, раціональні рухи. Психофізіологічні та фізіологічні вимоги формулюють відповідність обладнання і робочого місця можливостям співробітника щодо розуміння, обробки даних, пошук і реалізації рішень.

Організація робочого місця передбачає наступні пункти:

– раціональне положення робочого місця у приміщенні;

– вибір робочих меблів відповідно до фізіологічних характеристик працівника;

– правильне компонування і розміщення обладнання на робочих місцях;

– урахування особливостей та характеру професійної діяльності.

До загальних принципів організації робочого місця відносять:

– робоче місце повинне містити тільки ті предмети, які беруть учать у робочому процесі, але не заважати йому;

– предмети, які часто використовуються у роботі, розміщуються ближче,

ніж ті речі, якими користуються рідше;

– предмети, які беруться лівою рукою, повинні розміщуватися зліва, а предмети, які використовуються правою рукою — справа;

– якщо при роботі з предметом працівник використовує дві руки, то він розміщується з урахуванням зручності захоплювання його двома руками;

– робоче місце не повинно бути засмічене;

– необхідна оглядовість повинна бути забезпечена при правильній організації робочого місця.

Робоча поза – це найбільш тривале положення тіла працівника протягом робочого дня. При зручній робочій позі забезпечується стійкість положення тулуба, ніг, рук, голови і витрачається мінімальний запас енергії та максимальну продуктивність [26].

Сидячи і стоячи – дві найпопулярніших пози у робочому процесі. При проектуванні робочого місця потрібно враховувати, що з фізичним навантаженням бажана поза стоячи, а при малих зусиллях – сидячи. При роботі стоячи, людина стомлюється більше ніж сидячи. У відсотковому еквіваленті це на 10% більше енергії. При додатковому навантаженні підвищується артеріальний і венозний тиск крові, розширення вен, пошкоджуються ступені та викривляється хребет. У свою чергу при сидячій роботі нижня частина тіла розслаблена, а основне навантаження спрямоване на м'язи шиї, спини, таза, стегон. При неправильній сидячій позі розвивається застій крові у ногах, а якщо пальці виконують багато роботи можливе запалення суглобів.

Організація робочого місця при використанні ПК повинна відповідати усім ергономічним вимогам. Ключові ергономічні вимоги до проектування робочого місця оператора ПК зображені на рисунку 4.1.

При роботі з персональним комп’ютером потрібно:

– зменшувати кількість статичних напружень;

– розподіляти кількість і час статичних напружень;

– змінювати робочі пози під професійної діяльності.

Саме вибір правильної робочої пози визначається від впливу багатьох факторів. Одні з найважливіших це – кількість зусиль яка прикладається, велична робочої зони, відношення висоти робочої поверхні і ростом працівника.

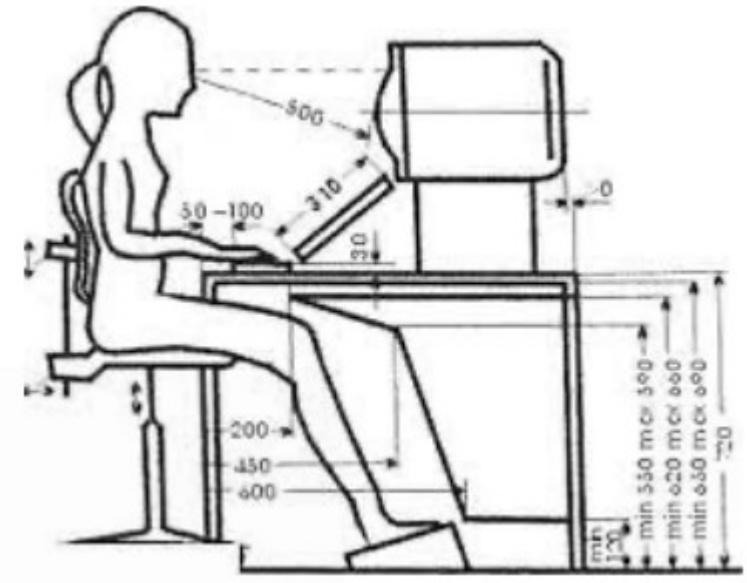

Рисунок 4.1 – Робочий стіл і розміщення користувача ПК

При використанні розробки користувач повинен дотримуватися вище зазначених правил, які будуть сприяти комфортній і продуктивній роботі. Важливо регулярно робити короткі перерви. Часта зміна заняття – кращий спосіб уникнути можливих неприємностей.

ВИСНОВОК

В результаті виконання кваліфікаційної роботи було розроблено допоміжний інструмент для аналізу компонентної архітектури на базі механізмів машинного навчання, з базовим зворотним зв'язком для користувача та можливістю налаштувати інструмент під конкретний проект та команду.

У ході роботи гіпотеза про роль зв'язків модулів поліпшення архітектури підтвердилася. Розроблене рішення дозволяє виявляти розбіжності у структурі схожих по абстрактності та стійкості модулів, що дозволить більш ефективно проводити рефакторинг та контролювати якість архітектури проекту.

Виникали труднощі з узагальненням підходів до контролю якості архітектури для різних проектів, не втрачаючи при цьому ефективності рефакторингу. А також із підрахунком метрик абстрактності у поточних реаліях.

У процесі виконання роботи було вирішено такі задачі:

– досліджено проблеми проектування архітектури реальних проєктів;

– розглянуто існуючі рішення (метрики, інструменти) для проблем архітектури;

– сформовано користувальницькі сценарії та вимоги до інструменту, на підставі отриманих знань про предметну область;

– спроектовано загальну архітектуру рішення;

– сформовано та декомпоновано план подальшої роботи;

– продумано алгоритми виявлення архітектурних проблем у проєктах і розписано труднощі у реалізації;

– вивчено підходи машинного навчання та вибрано оптимальний для інструменту;

– зібрано базовий прототип рішення для підтвердження гіпотез;

– перевірено ефективність інструменту та виявлено точки для подальшого ітеративного покращення інструменту.

Інструмент планується розвивати й надалі в частині додавання більше

архітектурних метрик та пошуку інших архітектурних проблем проєкту, а також більше налаштувань користувача інструменту (шаблони для рефакторингу, стратегії аналізу, строгість кластеризації. Планується реалізувати IDE -плагін асистент для поліпшення досвіду користувача і прискорення зворотного зв'язку, впровадити в реальні проєкти та зафіксувати переваги/недоліки інструменту для емпіричного дослідження та подальшого покращення.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Last words on UI architecture before an AI takes overe. URL: https://oleg008.medium.com/last-words-on-ui-architecture-before-an-ai-take s-over-468c78f18f0d (дата звертання: 03.04.2024).

2. Еволюція Enterprise-архітектур. Від MVC до Clean Architecture. URL: https://speakerdeck.com/dotnetru/fiedor-shchudlo-evoliutsiia-enterprise-arkh itiektur-ot-mvc-do-clean-architecture (дата звертання: 03.04.2024).

3. Книга про принципи SOLID та об’єктно-орієнтованому дизайні програм. URL: https://ota-solid.vercel.app/ (дата звертання: 12.04.2024).

4. Clean Architecture on Frontend. URL: https://dev.to/bespoyasov/ clean-architecture-on-frontend-4311 (дата звертання: 13.04.2024).

5. Just In Time And Software Development. URL: https://agiledesign.org/2015/ 01/18/jit-just-in-time-and-software-development/ (дата звертання: 14.04.2024).

6. Роберт Мартін. Чиста архітектура. Мистецтво розробки програмного забезпечення. Харків: Фабула, 2019. 368 с.

7. Atomic Design Methodology book by Brad Frost. URL: https://atomicdesign.bradfrost.com/chapter-2/ (дата звертання: 23.04.2024).

8. About architecture | Feature-Sliced Design. URL: https://feature-sliced.design/ docs/concepts/architecture (дата звертання: 23.03.2024).

9. Лайвкодінг: Рефакторимо складний. URL: https://youtu.be/Y1y4eVjj6R0 (дата звертання: 24.04.2024).

10. Історія розвитку реального проєкту з рефакторінгом в GitHub. URL: https://github.com/ani-team/github-client/commits/workshop/feature-slicednext (дата звертання: 04.05.2024).

11. Madge – Create graphs from your CommonJS, AMD or ES6 module dependencies. URL: https://github.com/pahen/madge (дата звертання: 14.05.2024).

12. Stepsize – Track and prioritise tech debt. URL: https://www.stepsize.com (дата звертання: 16.05.2024).

13. SSD 15/16: Coupling & Cohesion and Other Metrics. URL: https://yegor256.github.io/ssd16/15-metrics.pdf (дата звертання: 18.05.2024).

14. SSD 16/16: Future of Software Design. URL: https://yegor256.github.io/ssd16/16-future.pdf (дата звертання: 18.05.2024).

15. ESLint – Pluggable JavaScript linter. URL: https://eslint.org (дата звертання: 22.05.2024).

16. GitHub Copilot – Your AI pair programmer. URL: https://copilot.github.com (дата звертання: 22.05.2024).

17. Graphviz – Graph Visualization Software. URL: https://graphviz.org/ (дата звертання: 24.05.2024).

18. Мартін Роберт Чистий код: створення, аналіз, рефакторинг. Харків.: Фабула, 2019. 416 с (дата звертання: 26.05.2024).

19. Пора занятися рефакторінгом архітектури. URL: https://youtu.be/ dJZ7os3IvCQ (дата звертання: 28.05.2024).

20. Басюк Т.М. та ін. Машинне навчання: Навчальний посібник Львів: Видавництво «Новий Світ - 2000», 2021. 315 с.

21. Density Based Clustering. URL: https://www.npmjs.com/package/density-clustering (дата звертання: 29.05.2024).

22. Scatter Chart | Chart.js. URL: https://www.chartjs.org/docs/latest/charts/ scatter.html (дата звертання: 30.05.2024).

23. Удар струмом: перша допомога. URL: http://tomrda.gov.ua/news /578646863743857435/ (дата звертання: 04.06.2024).

24. Перша допомога при ураженні електричним струмом. URL: https://bozhedarivskaselrada.gov.ua/ news/1576497483/ (дата звертання: 04.06.2024).

25. Ергономічні вимоги до організації робочих місць. URL: https://pidru4niki.com/14821111/bzhd /ergonomichni_vimogi_ organizatsiyi (дата звертання: 05.06.2024).

26. Охорона праці в офісі. Вимоги до робочого місця офісного працівника. URL: https://gc.ua/uk/oxorona-praci-v-ofisi-vimogi-do-robochogomiscya-ofisnogo-pracivnika/ (дата звертання: 05.06.2024).

ДОДАТОК А

Реалізація класа Project

```
export class Project {
    imports: ImportsGraph;
    structure: Structure;
    files: TFile[];
    modules: Module[];
    modulesGraph: ModulesGraph;
    modulesWeights: ModulesWeights;

    constructor(imports: ImportsGraph, options: Options = DEFAULT_OPTIONS) {
        const inOptions = { ...DEFAULT_OPTIONS, ...options, }
        this.imports = this.cleanImports(imports, inOptions.exts);
        this.files = Object.keys(this.imports);
        this.structure = this.getStructure();
        this.modules = getModules(this.structure);
        this.modulesGraph = this.getModulesGraph();
        this.modulesWeights =
this.getModulesWeights(inOptions.abstractnessDepth);
    }

    private cleanImports(imports: ImportsGraph, exts: string[]): ImportsGraph
{...}

    private getStructure(): Structure {...}

    private getModulesGraph(): ModulesGraph {...}

    private getModulesWeights(depth = 5): ModulesWeights {...}

    public asModule(file: TFile): Module {...}
};
```

ДОДАТОК Б

Розрахунок архітектурних метрик Instability, Abstractness

```
export function calcInstability(module: Module, project: TProject): number {
    // Calc deps by external modules
    const outDeps = project.imports[module].length;
    if (outDeps === undefined) return -1;
    // Calc deps from external modules
    const inDeps = Object.entries(project.imports).filter(([gModule, gDeps]) =>
{
        const gResult = gDeps.includes(module);
        return gResult;
    }).length;

    return outDeps / (inDeps + outDeps);
}

export function calcAbstractness(module: Module, project: TProject): number {
    const maxWeight = _.max(Object.values(project.modulesWeights)) as number;
    const moduleWeight = project.modulesWeights[module];
    if (moduleWeight === undefined) return -1;
    return 1 - (moduleWeight / maxWeight);
}
```

ДОДАТОК В

Реалізація кластеризації модулів

```
import mlClustering from "density-clustering";
import * as analyzer from "analyzer";


export function prepareDataset(project: TProject) {
    const dataset = project.modules.map((unit) => [
        analyzer.metrics.calcInstability(unit, project),
        analyzer.metrics.calcAbstractness(unit, project),
    ]);
    return dataset;
}


export function cluster(dataset: Dataset, options: ClusterOptions):
ClustersResult {
    const dbscan = new mlClustering.DBSCAN();
    const clusters = dbscan.run(dataset.data, options.neighRadius,
options.neighNum);
    const noise = dbscan.noise;
    return { clusters, noise };
}
```