НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

«На правах рукопису»
УДК _____

До захисту допущено:
Завідувач кафедри
_____ Олександр РОЛІК
«___»_____2022 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Інформаційні управляючі системи та технології»

зі спеціальності 126 «Інформаційні системи та технології»

на тему: «Система виявлення архітектурних помилок програмного забезпечення в середовищі TypeScript»

Виконав:
студент VI курсу, групи ІС-01мн
Волошин Микола Віталійович _____

Керівник:
доц., к.т.н., доц каф. ІСТ
Коган Алла Вікторівна _____

Консультант:
доцент, к.т.н., доцент
Жданова Олена Григорівна _____

Рецензент:
Доцент, к.т.н., доцент
Роковий Олександр Петрович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2022 року

- *Огляд літератури та аналіз предметної області;*
- *Огляд наукових робіт та аналіз існуючих метрик вихідного коду;*
- *Розроблення нових метрик, що корелюють з якісними характеристиками вихідного коду;*
- *Розроблення програмної реалізації системи виявлення архітектурних помилок в середовищі TypeScript;*
- *Експериментальне дослідження розробленої системи.*

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформаційних систем та технологій

Рівень вищої освіти – другий (магістерський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-наукова програма «Інформаційні управляючі системи та технології»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Олександр РОЛІК

«_06_»_____06_____2022 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Волошину Миколі Віталійовичу

1. Тема дисертації «Система виявлення архітектурних помилок програмного забезпечення в середовищі TypeScript», науковий керівник дисертації Коган Алла Вікторівна, доц., к.т.н., доц каф. ІСТ, затверджені наказом по університету від «_26_»____04_____ 2022 р. №_НС/88/2022_

2. Термін подання студентом дисертації “_08_” _06_ 20_22_ р.

3. Об’єкт дослідження вихідний код програмного забезпечення в середовищі TypeScript

4. Предмет дослідження математичні моделі автоматизованого аналізу якісних характеристик архітектури, її елементів та їх зв’язків між собою

5. Перелік завдань, які потрібно розробити:

- *Огляд літератури та аналіз предметної області;*
- *Огляд наукових робіт та аналіз існуючих метрик вихідного коду;*
- *Розроблення нових метрик, що корелюють з якісними характеристиками вихідного коду;*
- *Розроблення програмної реалізації системи виявлення архітектурних помилок в середовищі TypeScript;*
- *Експериментальне дослідження розробленої системи.*
-

6. Орієнтовний перелік графічного (ілюстративного) матеріалу
Плакат 1 Функціонально-логічна структура програмного забезпечення
Плакат 2 UML-діаграма варіантів використання
Плакат 3 Діаграма послідовності
Плакат 4 UML-діаграма класів
Плакат 5 UML-діаграма компонентів
Плакат 6 Випадки, що надають підставу для покращення розрахунку сполученості модуля
Плакат 7 Можливі зв'язки між сутностями модуля для розрахунку сполученості
Плакат 8 Граф деякої програми

7. Орієнтовний перелік публікацій

Волошин М.В., Software Architecture Description Lifecycle, 2022, «Интернаука»: научный журнал – № 7(230). Часть 4. – М., Изд. «Интернаука», 2022. – с. 5-10.

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9. Дата видачі завдання " 31 " січня 20 22 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1.	Аналіз літератури предметної області	22.02	
2.	Аналіз існуючих моделей якості	19.03	
3.	Порівняння існуючих метрик вихідного коду та створення нової метрики	03.04	
4.	Розробка програмного забезпечення	29.04	
5.	Проведення експериментальних досліджень розробленої системи	05.05	
6.	Підготовка документації	22.05	
7.	Подання роботи на попередній захист	26.05	

- *Огляд літератури та аналіз предметної області;*
- *Огляд наукових робіт та аналіз існуючих метрик вихідного коду;*
- *Розроблення нових метрик, що корелюють з якісними характеристиками вихідного коду;*
- *Розроблення програмної реалізації системи виявлення архітектурних помилок в середовищі TypeScript;*
- *Експериментальне дослідження розробленої системи.*

8.	Подання роботи на основний захист	06.06	

Студент Волошин Микола

Науковий керівник Коган Алла

РЕФЕРАТ

Магістерська дисертація на здобуття ступеня «магістр» за освітньо-науковою програмою підготовки «Інформаційно управляючі системи та технології» на тему «Система виявлення архітектурних помилок програмного забезпечення в середовищі TypeScript». Дисертація містить 100 сторінок, 24 рисунки, 14 таблиць, 1 додаток, 25 джерел.

Актуальність. Ринок IT технологій розширюється і зростає через його вплив на всі інші сфери, разом з ним зростають і складність ПЗ та вимоги до його якості. Також експоненційно зростає вартість архітектурної помилки, що зумовлює актуальність валідації архітектурних рішень реалізованих в програмному забезпеченні.

Метою дослідження є полегшення та покращення процесу розробки програмного забезпечення в розрізі його архітектури.

Об’єкт дослідження - вихідний код програмного забезпечення в середовищі TypeScript.

Предметом дослідження є математичні моделі автоматизованого аналізу характеристик архітектури, її елементів та їх зв’язків між собою.

Наукова новизна одержаних результатів полягає у покращенні обчислення сполученості модулів та пропозиції до обчислення модульності як метрики.

Практичне значення одержаних результатів полягає у реалізації програмного застосунку для автоматизованого аналізу на предмет помилок в архітектурі програмного забезпечення в середовищі TypeScript.

Публікації. Матеріали роботи були опубліковані у журналі «Интернаука» («Information technology») (2022).

АРХІТЕКТУРА ПЗ, ЯКІСНІ ХАРАКТЕРИСТИКИ ВИХІДНОГО КОДУ, МОДЕЛІ ЯКОСТІ, TYPESCRIPT.

ABSTRACT

Master's dissertation for the degree of "master" in the educational and scientific program of training "Information control systems and technologies" on "Software architecture failure detection system in the TypeScript environment." The dissertation contains 100 pages, 24 figures, 14 tables, 1 appendice, 25 sources.

Topicality. The IT market is expanding and growing due to its impact on all other areas, along with the growing complexity of software and quality requirements. The cost of architectural error is also growing exponentially, which determines the relevance of validation of architectural solutions implemented in software.

The purpose of the study is to create a system for automated analysis and verification of project architecture for compliance with specified architectural requirements.

The object of research is the source code of the software in the TypeScript environment.

The subject of research is mathematical models of automated analysis of the characteristics of architecture, its elements and their relationships.

The scientific novelty of the obtained results is to improve the calculation of the module cohesion and the proposal to calculate the modularity as a metric.

The practical significance of the obtained results is the implementation of a software application for automated analysis for errors in software architecture in the TypeScript environment.

Publications. The materials of the work were published in the journal "Internauka" ("Information technology") (2022).

SOFTWARE ARCHITECTURE, QUALITATIVE CHARACTERISTICS OF SOURCE CODE, QUALITY MODELS, TYPESCRIPT.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ ВЕЛИЧИН І ТЕРМІНІВ

ПЗ - Програмне Забезпечення

TOGAF - The Open Group Architecture Framework

ADM - Architecture Development Method

MVP - Minimal Valuable Product

UML - Unified Modeling Language

CI - Continuous Integration

CD - Continuous Delivery

АП - Архітектура Підприємства

IT - Information Technologies

SQuaRE - Systems and software Quality Requirements and Evaluation

LCOM - Lack of Cohesion Of Methods

ЗМІСТ

ВСТУП

Щороку ринок інформаційних технологій зростає через підвищення попиту на програмне забезпечення, так як наразі розвиток всіх сфер безпосередньо залежить від нього. З ростом попиту зростають також конкуренція і вимоги, що призводить до необхідності з кожним роком поставляти на ринок більш якісну продукцію набагато швидше. Зважаючи на потрійні обмеження (час, вартість та якість), однозначно зростає вартість програмного забезпечення. В залежності від масштабів проекту вартість зростає експоненційно, що пов’язано з ростом складності системи і неможливістю мозку однієї людини охопити всю систему в цілому. Одна з найбільших затрат в розробці полягає в оплаті праці висококваліфікованих кадрів, серед котрих одними з найдорожчих являються архітектори ПЗ.

Архітектор ПЗ являє собою посередника між бізнесом і розробниками. Він перетворює бізнес-вимоги до проекту в архітектурний проект, що являє собою фундамент майбутнього ПЗ. Саме від цього етапу дуже сильно залежать такі важливі характеристики ПЗ:

- Масштабованість
- Модульність
- Швидкодія
- Безпека
- Зрозумілість ПЗ - наскільки легко розібратися в програмному коді системи
- Надійність - наскільки ПЗ схильне до збоїв
- Переносимість - легкість заміни залежностей ПЗ
- Стійкість - реакція на аварійні ситуації
- Повторне використання програмного коду
- Сумісність - легкість поєднання одних елементів з іншими

- Ефективність - властивість ПЗ мінімально залежати від обладнання (ресурсів)
- Верифікованість - легкість виявлення несправностей та виправлення помилок
- Цілісність - здатність ПЗ захищати свої компоненти від несанкціонованого доступу
- Однорідність - все ПЗ створено в одному стилі
- Консистентність - узгодженість, цілісність, внутрішня несуперечливість
- Економічність - ПЗ виконано в рамках бюджету

Тут з’являються перші проблеми:

- відсутність автоматизованих засобів що спростять аналіз створеної архітектури на помилки та відповідність вимогам до ПЗ, що вимагає більших витрат часу на розробку
- високий ризик допущення помилок через високу складність програмних систем

За даними наукових досліджень вартість архітектурної помилки експоненціально зростає на кожному наступному етапі і після випущення ПЗ може коштувати більш ніж в 10 разів дорожче від вартості виправлення на етапі створення архітектури*[26-28с. Steve McConnell “Code Complete”]*. На даний момент через розвиток інтернету більшість проектів являють собою сервіси що розвиваються циклічно і до яких неможливо розробити готовий архітектурний проект, тобто архітектура ПЗ на даний момент має властивість змінюватись динамічно. Звідси випливає ще декілька проблем:

- більшість архітекторів уникають створення архітектурних проектів через необхідність їх додатково підтримувати і підтримувати консистентність між ними та програмним кодом (два джерела істини)

- високі витрати грошей та висока складність контролю архітектури на проекті через необхідність архітектору мануально перевіряти коректність та відповідність коду до стандартів
- складність аналізу архітектури на проекті через високу деталізованість представлення у вигляді программного коду
- неоднозначність стандартів у разі відсутності архітектурного проекту або вимог до написання програмного коду

Для зменшення витрат на розробку, підвищення надійності та пришвидшення розробки програмного забезпечення необхідна часткова автоматизація (вірогідніше за все повна автоматизація неможлива через високу залежність архітектурних рішень від бізнес-вимог).

Розробка системи виявлення архітектурних помилок програмного забезпечення є нетиповою задачею. Хоча вона частково включає поширену задачу парсингу коду, частина аналізу, валідації та верифікації архітектури є дуже нетиповою і наразі немає жодного повноцінного аналога. Єдиний аналог представлений на ринку - плагін для Visual Studio, що дозволяє переглянути карту коду, але його функціонал дуже вузький і допомагає тільки з деякими помилками (наприклад, знайти циклічну залежність).

Найважливіша і найскладніша проблема, що вирішується в рамках даної роботи - це розробка математичних моделей для аналізу якісних характеристик архітектури, її сутностей та зв’язків між ними.

Дана робота передбачає створення MVP даної системи, що має включати збір всіх ключових сутностей вихідного коду проекту, що аналізується, аналіз кількох важливих з точки зору архітектури якісних характеристик та кілька правил валідації, що базуються на обчислених якісних характеристиках і повідомляють про їх порушення.

Дана система має дуже великий потенціал і дозволяє з розвитком отримати такі можливості:

- Валідація архітектури вихідного коду на відповідність UML-діаграмам
- Інтеграція в CI/CD для валідації на рівні git
- Кастомізація набору правил валідації на рівні проекту
- Створення власних правил валідації базуючись на отриманих метаданих з графу вихідного коду
- Створення та додавання власних характеристик разом з функціями для їх обчислення на основі графа вихідного коду
- Динамічне представлення графу вихідного коду в браузері для візуального аналізу архітектури проекту з можливістю:
    - Фільтрації даних що відображаються (наприклад, можна буде переглянути тільки класи, функції та модулі, включно з їх зв’язками, ігноруючи всі інші сутності)
    - Перегляду всіх розрахованих характеристик
    - Групування сутностей та їх шарів на основі розрахованих характеристик

Таким чином, дана система може значно спростити вирішення таких проблем, як відповідність вихідного коду архітектурі описаної в UML документах або аналіз наявної архітектури проекту перед рефакторингом, і навіть ознайомлення з архітектурою проекту для нових розробників тощо.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалась на кафедрі інформаційних систем та технологій Національного технічного університету України «Київський політехнічний інститут» ім. Ігоря Сікорського в рамках теми «Система виявлення архітектурних помилок програмного забезпечення в середовищі TypeScript».

Метою дослідження є полегшення та покращення процесу розробки програмного забезпечення в розрізі його архітектури.

Завданням дослідження є створення системи для автоматизованого аналізу та перевірки архітектури проекту на відповідність заданим архітектурним вимогам.

Об’єкт дослідження - вихідний код програмного забезпечення в середовищі TypeScript.

Предметом дослідження є математичні моделі автоматизованого аналізу характеристик архітектури, її елементів та їх зв’язків між собою.

Методи дослідження. Для виконання поставлених завдань у роботі було використано методи: аналіз вихідного коду та існуючих математичних моделей.

Наукова новизна одержаних результатів полягає у нових математичних моделях, які описують розрахунок значущих характеристик з точки зору архітектури проекту та інтеграція результатів в програмному застосунку.

Практичне значення одержаних результатів полягає у реалізації програмного застосунку для автоматизованого аналізу на предмет помилок в архітектурі програмного забезпечення в середовищі TypeScript.

Публікації. Матеріали роботи були опубліковані у журналі «Интернаука» («Information technology») (2022).

РОЗДІЛ 1 АНАЛІЗ ЗАДАЧІ. ОСНОВНІ ПОНЯТТЯ АРХІТЕКТУРИ ПЗ

1.1. Вступ

В даному розділі введені концептуальні основи та основні поняття архітектури програмного забезпечення, що дають представлення про ключові елементи архітектури важливі для проектування програмного застосунку.

1.2. Визначення архітектури системи

За визначенням стандарту ISO/IEC/IEEE 42010:2011 **архітектура системи** (architecture) - це основні поняття або властивості системи в оточуючому середовищі, втіленому в її елементах, відношеннях та конкретних принципах її проекту та розвитку.

Важливо зазначити, що архітектура будь-якої системи являє собою те, що є істотним відносно розглядаємої системи в її оточуючому середовищі. Не існує єдиної характеристики того, що є суттєвим або основним для системи; така характеристика може належати будь-чому з наступного:

- системним компонентам або елементам;
- тому, як системні елементи влаштовані або взаємопов’язані;
- принципам організації системи або проекту;
- принципам, що управляють розвитком системи в її життєвому циклі.

Відразу слід також зазначити, що в рамках термінології архітектура та опис архітектури відрізняються. Опис архітектури являє собою робочий продукт, в той час як архітектура являє собою абстракцію, що складається з понять та властивостей.

Примітка - Та ж сама система може бути зрозуміла за допомогою декількох різних архітектур (наприклад коли вони розглядаються в різних

оточуючих середовищах). Архітектура може бути виражена за допомогою декількох описів архітектури що відрізняються між собою (наприклад, коли використовують різноманітні структури архітектури). Та ж сама архітектура може характеризувати більш ніж одну систему (наприклад, сімейство систем ділення деякої загальної архітектури).

1.3. Опис архітектури

Опис архітектури (architecture description) - це робочий продукт, що використовується для вираження архітектури.

За стандартом ISO/IEC/IEEE 42010:2011 він включає:

- визначення опису архітектури та оглядову інформацію
- визначення заінтересованих сторін системи та їх інтересів
- визначення кожної точки зору на архітектуру, що використовується в описі архітектури
- **представлення архітектури та архітектурних моделей для кожної використовуємої точки зору на архітектуру**
- **застосовані правила зв'язків в описі архітектури, зв'язки** та реєстрацію відомих невідповідностей в необхідному змісті описів архітектури
- виконані обгрунтування для рішень архітектури

З цього списку нас цікавлять точки зору на архітектуру, представлення архітектури та архітектурних моделей, що містять в собі джерело інформації для розробників ПЗ.

З іншого боку можуть бути важливими застосовані правила зв'язків в архітектурі, що можуть містити важливу інформацію про відношення точок зору на архітектуру, представлень архітектури та архітектурних моделей між собою. На це важливо звернути увагу, бо це вказує на необхідність задавати правила адаптації для можливості скласти цілісне представлення архітектури.

1.4. Точка зору на архітектуру

Точка зору на архітектуру (architecture viewpoint) - Робочий продукт, що встановлює умовності конструювання, інтерпретації та використання архітектурного представлення для структуризації певних системних інтересів.

Точка зору встановлює умовності для конструювання, інтерпретації та аналізу представлення для того, щоб звернутися до інтересів, структурно представленим цією точкою зору. Умовності точки зору можуть включати мови, нотації, види моделей, правила проектування та\або методи моделювання, методики аналізу та інші операції в архітектурних представленнях.

Щоб краще зрозуміти сутність точок зору звернемося до різних UML діаграм, що реалізують різні точки зору. Наприклад Use Case діаграма виражає точки зору користувачів, а точка зору діаграми класів виражає поняття для досліджуваної предметної області.

1.5. Структури архітектури

Структура архітектури (architecture framework) - умовності, принципи та практики для опису архітектур, встановлені в границях заданої області застосування та\або об’єднання зацікавлених сторін.

Процес архітектурізації (architecting) - Процес розуміння, визначення, вираження, документування, взаємодії, відповідної сертифікації при реалізації, супроводження та покращення архітектури в життєвому циклі системи.

Структури архітектури і мови опису архітектури являються двома механізмами, що широко використовуються в процесі архітектуризації.

Структура архітектури встановлює звичайну практику для створення, інтерпретації, аналізу та застосування опису архітектури і границях визначеної області застосування або співтовариства зацікавлених сторін.

Застосування структур архітектури включає (але не обмежується цим):

- створення описів архітектури;
- розробку інструментів моделювання архітектури і методів процесу архітектуризації;
- встановлення процесів для сприяння зв’язку, зобов’язання та міжсистемному функціонуванню через множинні проекти та\або організації.

Приклади структур архітектури:

- структура архітектури Захмана для інформаційних систем [2];
- структура архітектури Британського міністерства оборони [3];
- структура архітектури відкритих груп (TOGAF) [4];
- модель представлення Крухтена “4+1” [5];
- чотири методи представлення Сіменса [6];
- еталонна модель для відкритої розподіленої обробки (RM-ODP) [ИСО/МЭК 10746];
- узагальнена еталонна архітектура підприємства (GERA) [ISO 15704];

1.6. Мови опису архітектури

Мова опису архітектури - деяка форма вираження для застосування в описах архітектури.

Мова опису архітектури забезпечує один або декілька видів моделі в якості засобу структуризації деяких інтересів для представників зацікавлених сторін. Мова опису архітектури може бути вузько-орієнтованою, призначеною для забезпечення декількох видів моделей, спеціально організованих для представлення точки зору. Часто мова опису

архітектури підтримується автоматизованими інструментаріями для супроводження створення, використання та аналізу його моделей.

Приклади мов опису архітектури:

- Rapide [12]
- Wright [13]
- SysML [14]
- ArchiMate [15]

1.7. Представлення архітектури та архітектурних моделей

Архітектурне представлення (architecture view) - робочий продукт, що виражає архітектуру деякої системи з точки зору певних системних інтересів.

Архітектурне представлення складається з однієї або більше архітектурних моделей. Архітектурна модель використовує умовності моделювання, що відповідають інтересам, на вирішення яких вони будуть направлені. Ці умовності визначені видом моделі, що управляє цією моделлю. В границях опису архітектури модель може бути частиною більше ніж одного архітектурного представлення.

Приклад архітектурної моделі - модель підсистеми побудови графу або модель авторизації користувачів. Тобто модель описує деяку частину кінцевого продукту, процесу тощо.

Для побудови архітектурних моделей частіше за все використовується уніфікована мова моделювання (UML).

Офіційні типи моделей UML 2 [16]:

- Діяльності - процедурна і паралельна поведінка;
- Класів - класи, властивості та відношення;
- Взаємодії - взаємодія між об’єктами з акцентом на зв’язках;
- Компонентів - структура і взаємозв'язок між компонентами;
- Зіставних структур - декомпозиція класу під час виконання;

- Розгортання - розгортання артефактів у вузли;
- Огляд взаємодії - комбінація діаграми послідовності і діаграми діяльності;
- Об’єктів - варіант конфігурації екземплярів;
- Пакетів - ієрархічна структура часу компіляції;
- Послідовності - взаємодія між об’єктами з акцентом на послідовності;
- Кінцевих автоматів - як події змінюють об’єкт протягом його життя;
- Часова - взаємодія між об’єктами з акцентом на синхронізації;
- Прецедентів - як користувачі взаємодіють з системою.

1.8. Життєвий цикл архітектури ПЗ

Наразі неможливо розглядати архітектуру ПЗ незалежно від архітектури підприємства (АП). Зважаючи на це розглянемо точку зору на архітектуру підприємства архітектурного фреймворку TOGAF (The Open Group Architecture Framework). Цей фреймворк був обраний для розгляду в даному контексті через 2 його особливості - наразі він вважається найпоширенішим та включає в себе метод розробки архітектури(ADM - Architecture Development Method). Саме цей метод ми і розглянемо після чого з’ясуємо життєвий цикл взаємодії архітекторів з розробниками.

1.8.1. TOGAF Architecture Development Method (ADM)

Всього TOGAF ADM виділяє 9 фаз (рис 1.1):

- Підготовча фаза
- Фаза розробки концепції АП (A)
- Фаза детальної розробки архітектури предметної області бізнесу (B)
- Фаза розробки архітектури програмних додатків і даних (C)
- Фаза розробки технологічної архітектури (D)
- Фаза перевірки можливостей розробленої архітектури (E)
- Фаза планування введення архітектури в експлуатацію (F)

- Фаза розробки рішення на основі архітектури (G)
- Фаза підготовки до майбутніх змін архітектури (H)

Як видно з рис. 1.1, Підготовча фаза виконується тільки один раз, в той час як всі інші виконуються циклічно і активно взаємодіють з джерелом вимог. Розглянемо кожну з фаз більш детально.

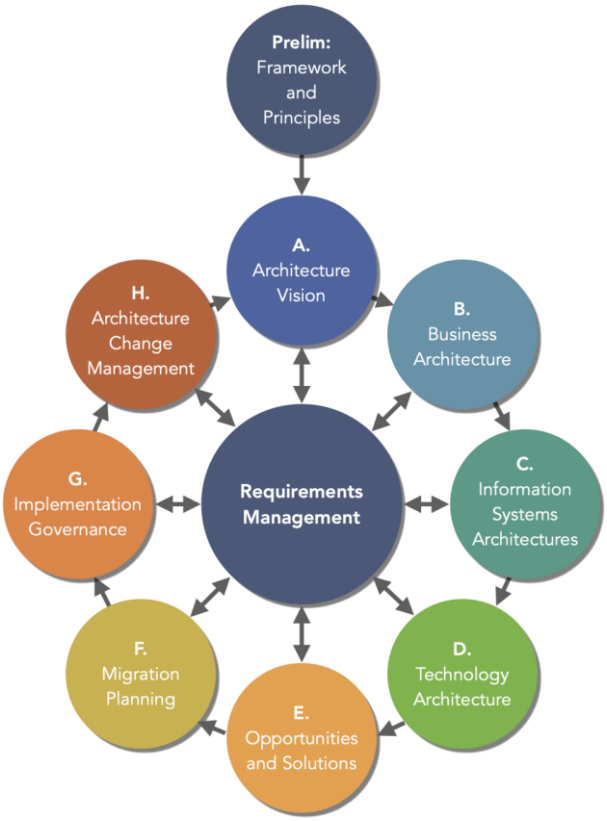


Рисунок 1.1 - Architecture Development Method

Завдання **підготовчої фази (Preliminary Phase)** - виявлення зацікавлених в процесі реалізації осіб і обговорення з ними завдань АП. На цій фазі виробляються керівні принципи архітектури (Architecture Guiding Principles), які ґрунтуються на бізнес-принципах організації і описують процеси і критерії для спостереження за процесом реалізації АП.

Фаза А призначена для вираження бачення АП. Концепція архітектури (Architecture Vision) використовує рушійні сили бізнесу, щоб визначити мету дій по створенню АП і створити опис першого разу для базового і цільового середовища. Якщо завдання бізнесу не ясні, то частина

завдання цієї фази - допомогти бізнесу ідентифікувати свої головні завдання і відповідні процеси, які повинна підтримувати АП. На цій фазі створюється спеціальний документ під назвою «Завдання на розробку архітектури» (Statement of Architectural Work), який окреслює область дії і умови АП.

Фаза B призначена для детальної розробки архітектури предметної області бізнесу. І базова і цільова архітектури, окреслені в документі «Концепція архітектури», деталізуються, щоб отримати корисні вхідні дані для технічного аналізу. На цій фазі використовуються різні методики та нотації бізнес-моделювання.

Фаза C пов'язана зі створенням архітектури предметних областей «Додаток» та «Дані (Інформація)». На цій фазі проводиться опис поточної та розробка цільової архітектури інформаційних систем. Для цього також використовуються спеціальні методики, підходи та нотації, пов'язані з даними та додатками.

Фаза D фокусується на описі та розробці технологічної архітектури (мережі, обчислювальні пристрої, вузли, технологічні інтерфейси тощо).

Мета **фази E** полягає в тому, щоб вивчити можливості, які пропонує цільова архітектура, і накреслити потенційне рішення. Робота у цій фазі концентрується навколо застосовності та практичності альтернатив реалізації. На цій фазі створюється план реалізації та впровадження, а також визначаються основні проекти.

У **фазі F** розставляються пріоритети проектів реалізації, виконуються деталізоване планування та аналіз прорахунків процесу міграції. До цього завдання входить оцінка залежностей між проектами та мінімізація їх підсумкового впливу на функції підприємства. У цій фазі оновлюється перелік проектів, деталізується «План реалізації та впровадження».

Після затвердження специфікації проекту фокус переміщається на формулювання більш конкретних умов та рекомендацій для кожного з проектів реалізації. Протягом **фази G** встановлюється зв'язок між керуючою

архітектурою (TOGAF) і розробною організацією (яка може бути налаштована за допомогою RUP та Project Management Body of Knowledge (PMBOK), наприклад, або будь-яких методологій управління проектом), а вибрані проекти реалізуються під керуванням формальної архітектури. На виході цієї фази ми маємо "Архітектурні контракти", що затверджуються організацією-розробником. Кінцевим виходом фази G є рішення, сумісні з архітектурою.

У **фазі H** акцент переноситься на управління зміною архітектури, яка досягається поставкою реалізованих рішень. У цій фазі може бути створена «Вимога до архітектурного завдання», що встановлює цілі для подальших циклів реалізації АП.

Метод масштабується і може використовуватися як для розробки архітектури компанії в цілому, так і для конкретного IT-рішення. Можливе використання разом з іншими методами, більш спеціалізованими на конкретних завданнях, а також з галузевими методами та стандартами.

Розглянувши ADM можемо виділити хоактер взаємодії між архітекторами та розробниками та життєвий цикл архітектури в рамках реалізації проектів на її основі [4].

1.8.2. Життєвий цикл взаємодії архітекторів з розробниками

Як ми з’ясували в попередньому підрозділі, всього є три точки взаємодії між архітекторами та розробниками:

- Підготовча фаза - вивчення наявних проектів з ціллю підготовки до розробки архітектури
- Фаза G - імплементація архітектури в готових рішеннях
- Фаза H - планування майбутніх змін в описі архітектури

Зазвичай перш ніж компанія вийде і закріпиться на ринку, в неї недостатньо ресурсів для створення архітектури, тож частіше за все, архітектура створюється вже на основі готового рішення з ціллю вирішити

проблеми компанії пов’язані з його якістю або швидкістю його розробки. В цей період архітектор може активно взаємодіяти з розробниками намагаючись зібрати цілісну картину проектів і їх взаємодії між собою. В цьому разі взаємодія носить суто консультативний характер.

Коли архітектурне рішення готове - в наявності є опис архітектури, що передається розробникам і імплементується ними. На цьому етапі розробники активно консультуються з архітектором і архітектор супроводжує весь процес імплементації.

За результатами розробки версії продукту архітектори та розробники акумулюють увесь отриманий досвід і дискутують, плануючи і обговорюючи майбутні зміни в архітектурному проекті.

1.9. Висновки

Якщо ми не можемо строго визначити що є істотним а що ні відносно будь-якої розглядаємої системи в деякому оточуючому середовищі, ми можемо заключити що неможливо повністю автоматизувати розробку архітектури системи без участі архітектора, в той самий час ми можемо одразу виділити деякі можливості для автоматизації перевірки архітектури ПЗ:

- відповідність базовим архітектурним обмеженням, шаблонам, вимогам та рекомендаціям
- відповідність архітектурним обмеженням, шаблонам, вимогам та рекомендаціям заданим оператором
- відповідність структури проекту правилам та обмеженням заданими оператором
- відповідність системних компонент опису архітектури (порівняння на входження архітектури отриманої з коду до архітектури в документах що описують опис архітектури)

- форматований вивід опису архітектури ПЗ для покращення аналізу архітектури оператором
- і т.д.

Звідси випливає що ми можемо допомогти контролювати зміни і розвиток проекту, допомагати його аналізувати та попередити про деякі можливі доступні для аналізу помилки.

Також можливе суттєве зменшення взаємодії між архітекторами та розробниками на попередній фазі підготовки архітектури та імплементації опису архітектури за рахунок спрощення аналізу проекту та автоматизації перевірки на відповідність вихідного коду моделям представленим в описі архітектури.

Таким чином опис архітектури набуває статусу джерела істини без необхідності підтримувати консистентність з кодом мануально і стає невід’ємною частиною реалізації проекту.

РОЗДІЛ 2 АНАЛІЗ ЗАДАЧІ. ЯКІСТЬ СИСТЕМИ

2.1. Вступ

Даний розділ розглядає основні поняття, визначення та характеристики якості інформаційних систем. Дає представлення про потреби, які задовольняє програмний застосунок і доводить його користь. Визначає характеристики якості, що можуть бути визначені у проекті з використанням застосунку. Даний розділ спирається на серію стандартів SQuaRE (Systems and software Quality Requirements and Evaluation).

2.2. Визначення якості

Якість системи - це степінь в якій система задовольняє заявлені та наявні потреби різноманітних зацікавлених сторін, що дозволяє таким чином оцінити переваги. Ці потреби представлені у вигляді моделей якості, що виражають якість продукту через розбиття на класи характеристик, які можуть поділятися на під-характеристики (рис 2.1).

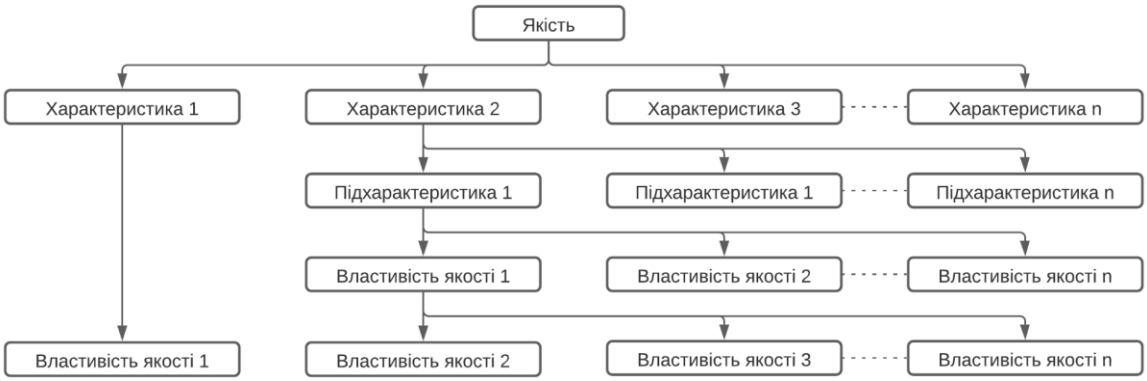


Рисунок 2.1 - Структура моделей якості

Властивості якості системи що вимірюються, називаються властивостями якості, що пов’язані з відповідними показниками якості.

На даний момент в моделі SQuaRE існують три моделі якості:

- модель якості при використанні

- модель якості продукту
- модель якості даних

Сумісне використання цих моделей якості надає підставу вважати, що взяті до уваги всі характеристики якості.

2.3. Модель якості при використанні

Модель якості при використанні визначає п'ять характеристик, пов'язаних з результатами взаємодії із системою: ефективність, продуктивність, задоволеність, свободу від ризику та покриття контексту (див. рис. 2.2).

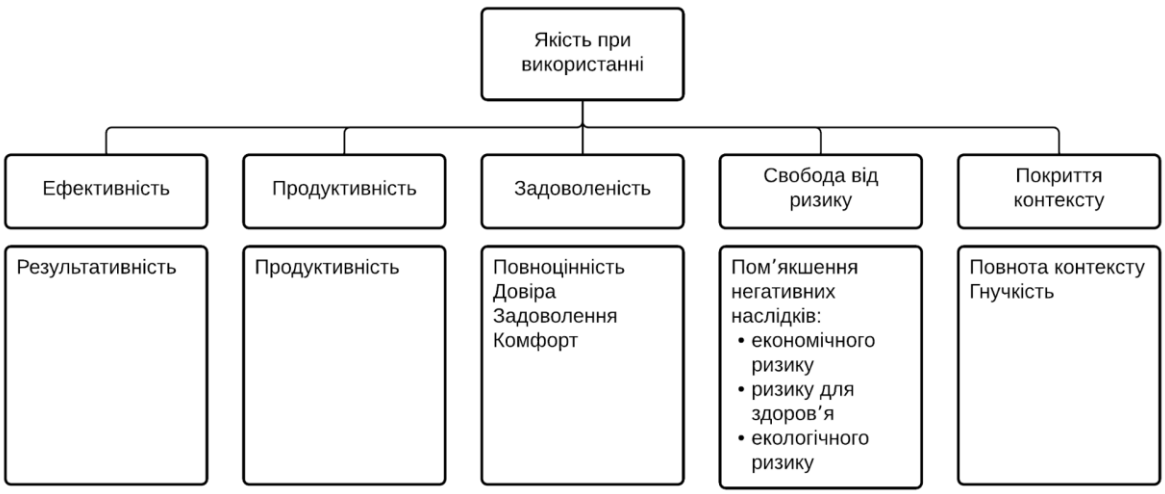


Рисунок 2.2 - Модель якості при використанні

Кожна характеристика застосовна до різних видів діяльності зацікавлених осіб, наприклад, для взаємодії оператора чи підтримки розробника.

Якість під час використання системи характеризує вплив продукції (система чи програмний продукт) на зацікавлену сторону. Воно визначається якостями програмного забезпечення, апаратних засобів, операційного середовища, а також характеристиками користувачів, завдань

та соціального середовища. Всі ці фактори роблять свій внесок у якість системи при використанні.

Так як ми не можемо отримати дані, які описує ця модель якості з вихідного коду, перейдемо до розгляду наступної моделі якості.

2.4. Модель якості продукту

Модель якості продукту зводить властивості якості системи/програмного продукту до восьми характеристик, якими є: функціональна придатність, рівень продуктивності, сумісність, зручність користування, надійність, захищеність, супроводжуваність та переносимість (мобільність). Кожна характеристика, в свою чергу, складається з низки відповідних підхарактеристик (рис. 2.3).

Рисунок 2.3 - Модель якості продукту

Модель якості продукту можна застосовувати як для програмного продукту, так і для комп'ютерної системи, до складу якої входить програмне забезпечення, оскільки більшість характеристик застосовується і до програмного забезпечення, і до систем.

2.4.1. Характеристики та підхарактеристики моделі якості продукту

До важливих для нас характеристик і підхарактеристик моделі якості продукту входять (див. рис. 2.3):

1) **Надійність (reliability)** – це ступінь виконання системою, продуктом або компонентом певних функцій за зазначених умов протягом встановленого періоду часу.

    a) У програмному забезпеченні зношування не відбувається. Проблеми з надійністю виникають через недоліки у вимогах, при розробці та реалізації або змін умов використання.

    b) Характеристики функціональної надійності програмного забезпечення включають готовність або притаманні їй, або зовнішні впливові фактори, такі як надійність і доступність (включаючи відмовостійкість і відновлюваність), безпеку (включаючи забезпечення конфіденційності і цілісність), придатність для обслуговування і довговічність.

2) **Готовність (availability)** - це рівень працездатності і доступності системи, продукту чи компонента. Загалом, готовність можна оцінити як частку загального часу, протягом якого система, продукт чи компонент перебувають у робочому стані. Готовність, таким чином, визначається поєднанням завершеності, що визначає частоту відмов, відмовостійкості та відновлюваності, яка, у свою чергу, визначає тривалість часу бездіяльності після кожної відмови.

3) **Супроводжуваність, модифікованість (maintainability)** - це результативність та ефективність, з якими продукт чи система можуть бути модифіковані передбачуваними спеціалістами з обслуговування.

    a) Модифікація може включати виправлення, поліпшення або адаптацію програмного забезпечення до змін в умовах використання, вимог і функціональних специфікаціях. Модифікації можуть бути виконані як спеціалізованим

технічним персоналом, так і робочим або операційним персоналом і кінцевими користувачами.

b) Супроводжуваність включає установку різного роду оновлень.

c) Супроводжуваність можна інтерпретувати або як продуктову або системну властивість, що спрощує процес обслуговування, або як якість при використанні, перевірена практично спеціалістами з обслуговування з метою підтримки продукту або системи.

4) **Модульність (modularity)** - це ступінь представлення системи або комп'ютерної програми у вигляді окремих блоків таким чином, щоб зміна одного компонента мала мінімальний вплив на інші компоненти.

5) **Можливість багаторазового використання (reusability)** - це ступінь, у якій актив може бути використаний у кількох системах або у створенні інших активів.

6) **Аналізованість (analysability)** - це ступінь простоти оцінки впливу змін однієї або більше частин на продукт або систему або простоти діагностики продукту для виявлення недоліків та причин відмов, або простоти ідентифікації частин, що підлягають зміні. Конкретна реалізація продукту або системи може включати механізми аналізу власних дефектів і формування звітів про відмови та інші події.

7) **Модифікованість (modifiability)** - це ступінь простоти ефективної та раціональної зміни продукту або системи без додавання дефектів та зниження якості продукту.

a) Реалізація модифікації включає кодування, розробку, документування і перевірку змін.

b) Модульність та аналізованість можуть впливати на модифікованість.

c) Модифікованість - це поєднання змінності та стійкості.

2.5. Характеристики якості, що можуть бути розраховані додатком

Коли ми визначили характеристики моделі якості при використанні та моделі якості продукту, можемо визначити характеристики, що можуть розраховуватися та визначатися використовуючи опис вихідного коду. Дуже важливо зазначити, що розрахунки відносяться до контексту архітектури та розробки і не відносяться до контексту використання та функціонування системи в будь-якому середовищі.

Переглянувши характеристики моделі якості при використанні однозначно стає зрозуміло, що вони не відносяться до статичних характеристик і всі пов’язані з зовнішньою оцінкою програмних продуктів зважаючи на контекст.

З моделі якості продукту ми виокремили такі характеристики, що можуть бути розраховані:

- Модульність
- Можливість багаторазового використання
- Модифікованість

Надалі цю інформацію буде використано для розробки методу розрахунку цих характеристик, що дозволить проводити аналіз і контроль їх якості на рівні архітектури та вихідного коду.

2.6. Вплив додатку в термінології розглянутих моделей якості

Додаток дозволяє покращити такі характеристики якості продукту:

- Надійність
- Готовність

- Супроводжуваність
- Модульність
- Можливість багаторазового використання
- Аналізованість
- Модифікованість

Тепер визначимо як утиліта впливає на програмний продукт:

1) Дуже часто проблеми виникають через не верифіковані зміни (особливо в тих випадках коли змінюються ключові сутності). Пов'язуючи їх з архітектурними документами, що їх описують, ми можемо гарантувати, що без змін архітектора внесених в документацію - ключові сутності не будуть змінені.

2) Дана утиліта контролює складність використовуючи деякі статичні параметри, що її характеризують, може зменшити вірогідність помилок і тим самим покращити надійність.

3) Дозволяючи швидко та якісно проаналізувати архітектуру та структуру системи, утиліта спрощує розуміння системи і дозволяє отримати її цілісне представлення.

4) Дозволяючи збирати і проводити валідацію статичних параметрів системи, утиліта дозволяє покращити стандартизацію методів та підходів, що застосовуються для її розробки.

Тепер розглянемо залежність визначених характеристик від різних впливів утиліти в Таблиці 2.1 (варто зазначити, що цей вплив поширюється на всі рівні програмного продукту). Для цього використаємо матричну таблицю, де визначимо який вплив покращує які якісні характеристики. Характеристики будуть обозначатися ім’ям, а вплив - порядковим номером зі списку вище.

Таблиця 2.1 - Вплив можливостей використання утиліти на якісні характеристики продукту

	1	2	3	4
Надійність	+	+	+	+
Готовність	+	+	-	+
Супроводжуваність	+	+	+	+
Модульність	-	+	+	+
Можливість багаторазового використання	-	+	+	+
Аналізованість	+	-	+	+
Модифікованість	-	+	+	+

Розглянемо аргументацію цієї таблиці:

- Залежність вихідного коду від опису архітектури не покращує модульність, можливість багаторазового використання та модифікованість через те, що ця функція лише фіксує роботу архітектора в коді і не впливає безпосередньо на ці характеристики, але значно підвищує аналізованість та надійність програмного продукту.

- Додатковий шар валідації на помилки (2 кейс) не впливає на аналізованость безпосереднім чином, а лише запобігає деяким архітектурним помилкам. В той час за рахунок автоматизованих методів визначення архітектурних помилок дозволяє покращити всі інші характеристики.

- Можливість ефективного візуального аналізу структури та архітектури системи дозволяє прямо покращити аналізованісь та через неї покращити всі інші якісні характеристики (архітектор або

розробник може з легкістю визначити помилки), але не впливає на готовність.

- Можливість покращення стандартизації впливає на всі визначені якісні характеристики програмного продукту через те, що надає змогу покращити структуру та архітектуру продукту і зробити його більш зрозумілим для розробників та архітекторів.

2.7. Висновки

Тепер, визначивши моделі якості програмних продуктів і якісні характеристики що в них входять, ми змогли довести користь розроблюваної утиліти і маємо весь необхідний базис для розробки методу розрахунку якісних характеристик, що дозволить об’єктивно вимірювати і оцінювати деякі аспекти якості продукту, і як наслідок проводити їх валідацію автоматично.

РОЗДІЛ 3 РОЗРОБКА МЕТОДУ РОЗРАХУНКУ ЯКІСНИХ ХАРАКТЕРИСТИК

3.1. Вступ

В другому розділі ми визначили характеристики якості які можуть бути розраховані, але не визначили яким чином. В цьому розділі ми визначимо методи розрахунку цих характеристик та границі достовірності методів відносно характеристик.

Границі достовірності - це характеристика, що визначає наскільки метод визначає рівень якості характеристики і які аспекти характеристики покриває а які ні.

Аспекти характеристики - це деякі параметри, що впливають на характеристику.

3.2. Класифікація статичних параметрів

Для розрахунків нам необхідно використовувати тільки статичні параметри. Всі ці параметри мусять бути невід’ємно пов'язані зі структурою та вихідним кодом проекту.

3.2.1. Контекст параметрів

Щоб розглядати параметри треба визначити сутності яким вони належать. Такими сутностями є всі сутності що являються частиною проекту. Так як ми в першу чергу розглядаємо вихідний код, то ми будемо брати до уваги лише ті сутності, які можуть його містити. В такому разі ми можемо назвати їх структурними рівнями.

Всі параметри будуть розглядатися на 4-х базових структурних рівнях:

- **Проект** - це кореневий модуль, що означає те що він є контекстним
- **Модуль** - це основна структурна одиниця проекту, що може містити інші модулі. Являє собою абстрактну структурну одиницю, що покращує інкапсуляцію і організацію проекту.
- **Директорія** - це структурна одиниця подібна до модуля, але на відміну від нього завжди має вертикальну структуру відносно проекту і являє собою конкретну структуру через те що є елементом файлової системи, в той час як модуль є абстрактним.
- **Файл** - це крайній елемент файлової системи, що може містити код.
- **Компонент** - це деякий елемент коду, що містить в собі деякий код. Тобто рядок коду де визивається функція не є компонентом, але сама функція ним являється. Також інші об’єкти і константи являють собою компоненти і можуть аналізуватися.

3.2.2. Визначення статичних параметрів

Коли ми визначили основні структурні одиниці, що ми бажаємо аналізувати, наступним кроком визначити деякі статичні параметри, що ми можемо отримати.

До таких параметрів відносяться:

- Кількість зовнішніх залежностей (зовнішні бібліотеки, тощо.)
- Характер зовнішніх залежностей
- Кількість рядків коду
- Кількість внутрішніх залежностей
- Характер внутрішніх залежностей
- Кількість вкладених об’єктів
- Тип об’єкту (наприклад функція, клас, тощо.)
- Кількість параметрів

- Стабільність (як часто змінюється код сутності)
- Кількість об’єктів, що використовують даний об’єкт в контексті репозиторію
- Глибина залежності (залежить від глибини графу залежності)
- Цикломатична складність (Cyclomatic complexity)

Не важко помітити, що параметри діляться на 2 типи - якісні та кількісні. Володіючи цією інформацією можна допустити, що якісні характеристики слід представити як коефіцієнти і пов'язати з кількісними.

3.3. Визначення аспектів якісних характеристик

Коли ми визначили дані і їх статичні параметри з якими ми можемо працювати, необхідно визначити аспекти, які забезпечують якісні характеристики. Дуже важливо умовитися, що для кожної якісної характеристики необхідно забезпечити повноту покриття аспектами, які впливають на них.

Повнота покриття якісної характеристики - це характеристика, яка визначає можливість точно визначити ступінь відповідності якості нормі, що визначена деякими конкретними умовами, які виражені в аспектах, що впливають на цю характеристику.

Всього в нас є, як ми визначили в минулому розділі, 4 якісні характеристики, які ми безпосередньо можемо обчислити використовуючи доступні статичні параметри:

- Модульність
- Можливість багаторазового використання
- Аналізованість
- Модифікованість

3.3.1. Метрики

Для початку визначимо, які метрики вже наявні для обчислення кожного з цих пунктів.

3.3.1.1. Фактори, що характеризують модульність

З визначення видно, що ключем являються залежності компоненти від інших компонент. Отже як ми розуміємо, що компонента має гарну або ж погану модульність? Для цього ми використовуємо аналіз зв’язків з іншими компонентами і їх ролей.

Модульними є компоненти, що мають низьку зв'язність і високу сполученність. Тож зв’язність і сполученість будуть основними метриками для обчислення модульності.

3.3.1.2. Фактори, що характеризують можливість багаторазового використання

З дослідження [22, с. 224] ми визначили, що на можливість багаторазового використання впливають такі фактори:

- Зв’язність
- Сполученість
- Складність
- Переносимість
- Наслідування
- Частота використання
- Зрозумілість
- Складність інтерфейсу
- Рівень можливості кастомізації
- Адаптивність
- Модульність
- Придатність до налагодження

- Композиційність
- Зручність використання
- Функціональність
- Надійність
- Задокументованість
- Розмір
- Функціональні залежності
- Мінливість
- Легкість використання
- Можливість налаштування
- Сумістність
- Завершеність
- Та деякі інші.

3.3.1.3. Фактори, що впливають на модифікованість

На модифікованість мають вплив такі фактори:

- Мінливість
- Розмір
- Задокументованість
- Функціональні залежності
- Композиційність
- Модульність
- Зв’язність
- Сполучність
- Адаптивність
- Наслідування
- Частота використання
- Зрозумілість
- Складність інтерфейсу

- Складність
- Та інші.

3.3.1.4. Метрики, що обрані для реалізації в рамках даної роботи

Так як зв’язність та сполученість мають вплив на всі якісні характеристики - їх було обрано для використання та модифікації в даній роботі. Так як вони характеризують модульність, в рамках дисертації буде виведена нова метрика, що описує модульність.

3.3.2. Зв’язність компонент

Зв'язність - це міра у якій компонент залежить від кожного іншого компоненту (використовує якусь інформацію про нього).

В функціональному програмуванні виділяються різні види зв’язності:

- **Зв'язність за змістом (сильна)**
  - З'являється, коли один модуль модифікує або залежить від внутрішнього змісту іншого модуля (наприклад, використовує його змінні). Тому зміна способу, яким другий модуль обробляє дані, вимагатиме зміни залежного модуля.
- **Зв'язність із спільності даних**
  - Коли два модулі мають загальні глобальні дані (глобальні змінні). Зміна загального ресурсу передбачає зміну всіх модулів, що його використовують.
- **Зовнішня зв'язність**
  - З'являється, коли два модулі поділяють нав'язаний ззовні формат даних, протокол комунікації або інтерфейс пристрою. Зазвичай це пов'язані з взаємодією із зовнішніми інструментами чи апаратурою.

- **Зв'язність контролю**
    - З'являється, коли один модуль контролює хід роботи іншого, передаючи йому інформацію про те, що робити.

- **Залежність-штамп (залежність від структурованих даних)**
    - Коли модулі мають загальну складну структуру даних, і використовують тільки її частини, можливо різні (наприклад, функції передається запис, хоча вона вимагає лише його частину).

- **Зв'язність даних**
    - Зв'язок даних виникає коли модулі діляться загальними даними через, наприклад, параметри. Кожний об’єкт даних є елементарним і єдиним яке ділиться (наприклад передача числа функції квадратний корінь, що обчислюється).

- **Зв'язок за повідомленням (слабка)**
    - Слабкий тип зв'язності. Досягається за допомогою децентралізації стану (об'єкти). Взаємодія компонентів здійснюється через параметри та обмін повідомленнями.

- **Зв'язність відсутня**
    - Модулі взагалі не взаємодіють між собою.

Також окремо виділяються деякі типи зв'язності в ООП:

- **Зв'язність підкласу**
    - Описує зв'язок між предком та нащадком. Нащадок прив'язаний до пращура, а предок немає.

- **Тимчасова зв'язність**

- Коли дві дії упаковані в один модуль лише тому, що вони можуть відбуватися одночасно.

3.3.2.1. Нестабільність

Для оцінки зв'язності ми використаємо визначення нестабільності.

Нестабільність (instability) - це властивість компоненти, що враховує кількість компонент, які використовують дану компоненту (AC) та кількість компонент, що використовуються даною компонентою (EC).

Дана характеристика оцінюється за формулою:

$$I = \frac{EC}{AC + EC} \quad (3.1)$$

Так як ця величина знаходиться в границях [0;1], ми можемо її з легкістю використати для оцінки. Чим менша кількість залежностей компоненти відносно всіх залежностей, що пов'язані з нею - тим більш стабільним є компонент. Тобто чим ближче ми до 0 тим краще і тим стабільнішим може вважатися компонент.

3.3.2.2. Окремий випадок нестабільності

Звернімо увагу на випадок, коли деяка сутність ніким не використовується і нікого не використовує, таким чином отримуємо наступний вираз:

$$I = \frac{0}{0 + 0} = \frac{0}{0} \quad (3.2)$$

, що неприпустимо в математиці.

Таким чином ми маємо визначити значення нестабільності для цього випадку.

Якщо сутність ні від кого не залежить і ніхто не залежить від неї, це означає, що вона ні з ким не пов’язана і немає необхідності її змінювати, тож з точки зору даної метрики, нестабільність сутності дорівнює нулю.

3.3.3. Спорученість компонент

Спорученість - це характеристика, що виражає згрупованість сутностей системи за деяким принципом. При сильній спорученості компоненти організовані в чіткі групи, при низькій - чіткі групи відсутні [19].

3.3.3.1. Спорученість класів

Щоб розрахувати спорученість для класів, зазвичай використовують метрику LCOM (Lack of Cohesion Of Methods, що в перекладі “Відсутність згуртованості методів”). Т.я. ця метрика з’явилася ще в минулому столітті і багато разів еволюціонувала, існує досить багато її варіантів реалізації [20]. Найкраще нам підходить варіант з нормалізацією [21], що дає нам діапазон метрики від 0 до 1:

$$LCOM = 1 - (sum(MF)/(M * F)) \quad (3.3)$$

, де:

- MF - кількість методів, що звертаються до певного поля класса;
- sum(MF) - сума MF всіх полів класу;
- M - всі методи класу (включно з конструкторами, статичними функціями, методами і т.д.);
- F - кількість полів класу.

3.3.3.2. Сполученість класів для класів без полів

Коли у класу відсутні поля (тобто клас зовсім не має даних) - виходить, що він є набором методів (або просто пустим та непотрібним) і формула набуває вигляду:

$$LCOM = 1 - (0/(M * 0)) = 1 - (0/0) \quad (3.4)$$

, тобто отримуємо ділення на 0.

У такому випадку клас порушує ідеологію ООП і має визначатися як клас у якого немає даних, а значить його LCOM має дорівнювати 1.

3.3.3.3. Сполученість класів для класів без методів

Коли у класу є поля але немає методів, формула набуває вигляду:

$$LCOM = 1 - (0/(0 * F)) = 1 - (0/0) \quad (3.5)$$

, тобто знов отримуємо ділення на 0, і при цьому клас вважається сполученим та цілісним так як головне - це дані. LCOM в такому випадку має повертати 0.

3.3.3.4. Сполученість модулів

Для розрахунку сполученості модулів Briand, Daly, and Wust [25] розробили метрику, що має схожу ідею з тим, як розраховується сполученість класів. В основі ідеї лежить підрахунок кількості всіх не напрямлених зв’язків між сутностями, що належать модулю та ділення її на максимально можливу кількість зв’язків між ними.

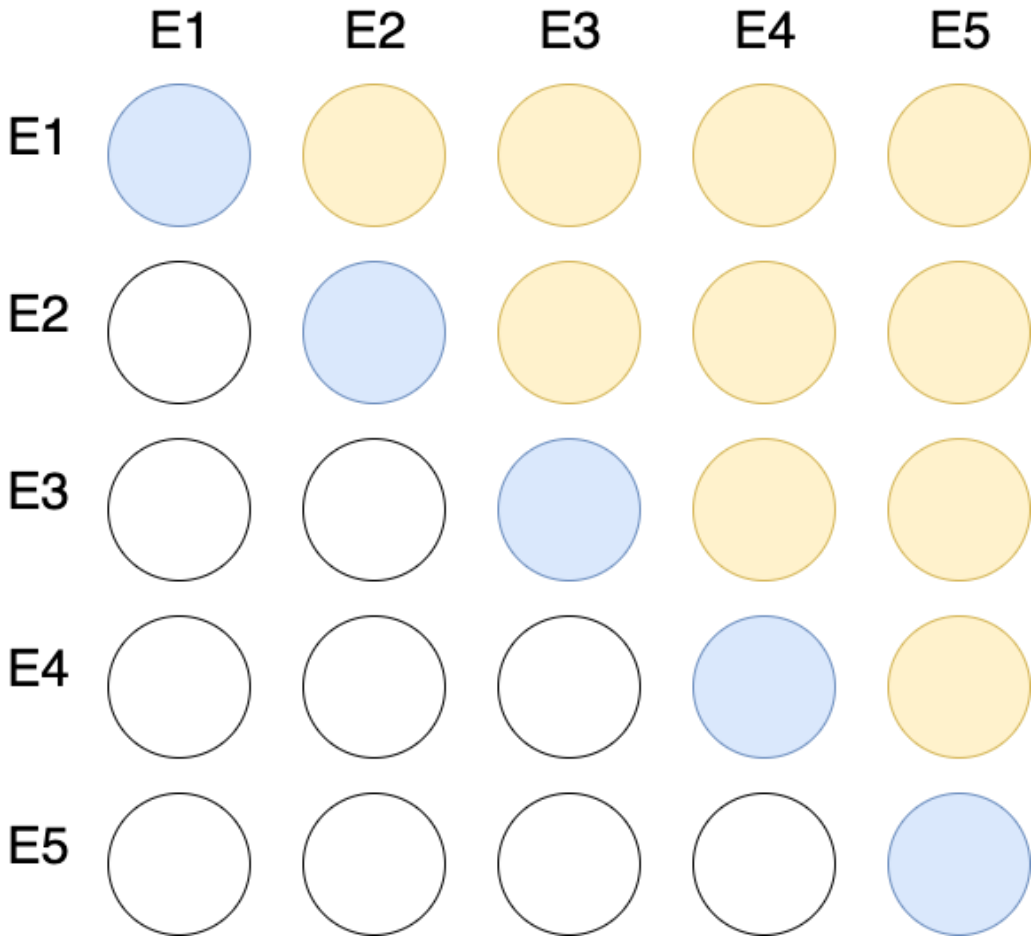


Рисунок 3.1 - Матриця відносин між сутностями модуля

На рисунку 3.1 зображено, як всі сутності можуть бути пов’язані між собою. Так як для даної метрики ніякого значення не має чи використовує сутність модуля сама себе і нас цікавлять тільки зв’язки між різними сутностями, ми не повинні ці зв’язки враховувати, тож ми повністю прибираємо діагональ матриці.

Після цього в нас залишається матриця напрямлених зв’язків між сутностями, але нам абсолютно не важливо яка з сутностей яку використовує і нам важливо знати тільки сам факт зв’язку між двома різними сутностями. Таким чином діленням на два ми прибираємо напрямленість і залишаємо кількість можливих не напрямлених зв’язків. На рисунку обрані зв’язки зображені білим кольором.

Отже, базова формула сполученості виглядає так:

$$MC = CE \,/\, ((AE^2 - AE)/2),\ AE \notin \varnothing \qquad (3.6)$$

, де:

- MC (module cohesion) - спорученість модуля;
- CE (connected entities) - кількість пов’язаних між собою сутностей модуля;
- AE (all entities) - кількість сутностей, що об’явлені в модулі.

3.3.3.5. Модифікована спорученість модулів

Представлена вище спорученість модулів досить непогано передає ідею, але є одна проблема. Давайте розглянемо два окремі випадки. Припустимо, що у нас є модуль, який в себе включає 6 сутностей. 3 сутності пов’язані між собою всіма можливими зв’язками та інші три сутності пов’язані між собою всіма можливими зв’язками. Ці трійки ніяк між собою не пов’язані (рис. 3.2).

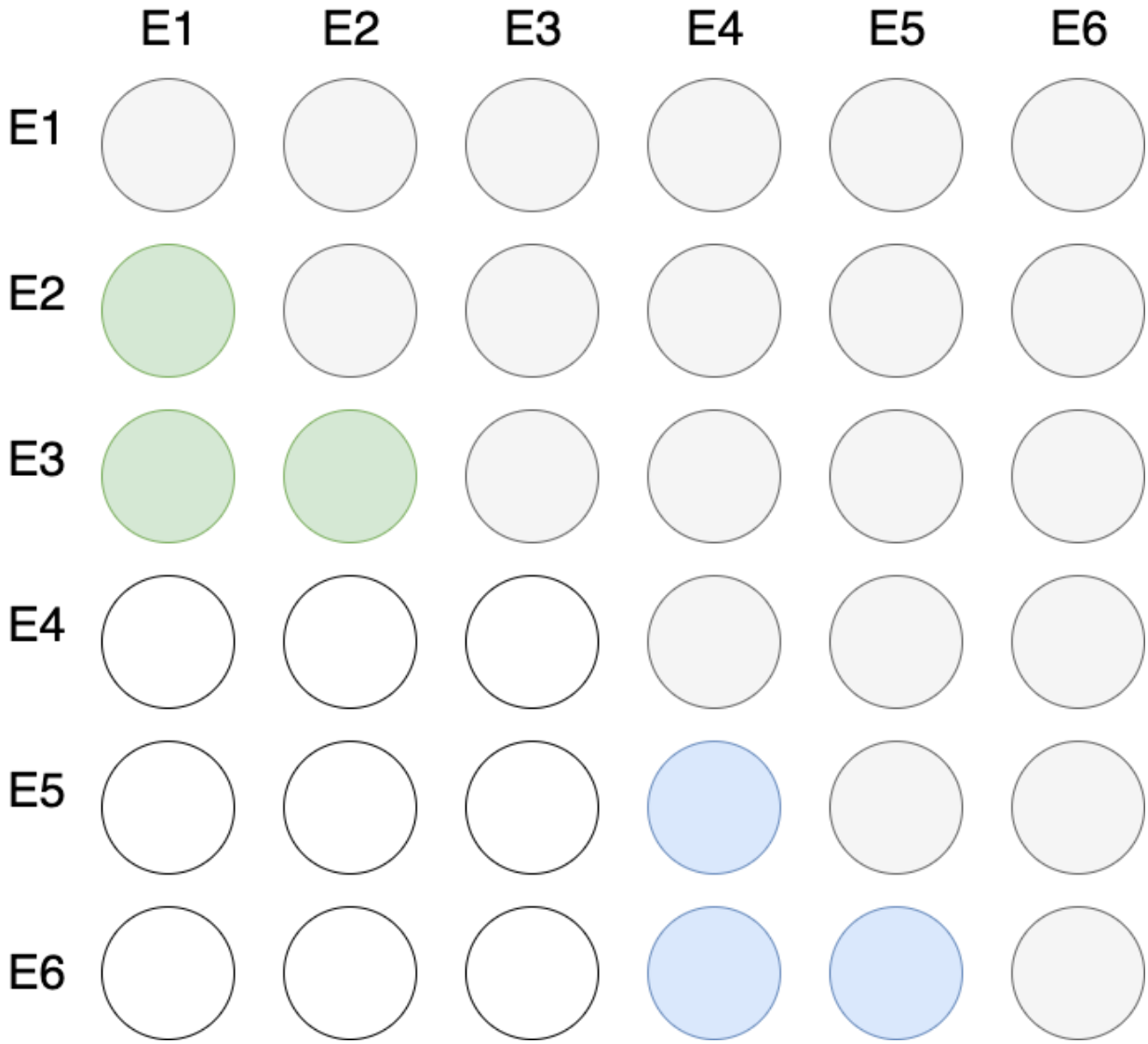


Рисунок 3.2 - Випадок з двома тіснопов'язаними між собою групами сутностей модуля

На рисунку 3.2 сірим зафарбовані неможливі зв'язки, зеленим зв'язки між першою групою сутностей та синім зв'язки між другою групою сутностей.

Тепер розглянемо другий випадок, коли ми маємо ту саму кількість зв'язків між сутностями, але на цей раз вони розподілені так, щоб всі сутності були пов'язані між собою і утворювали зв'язаний граф.

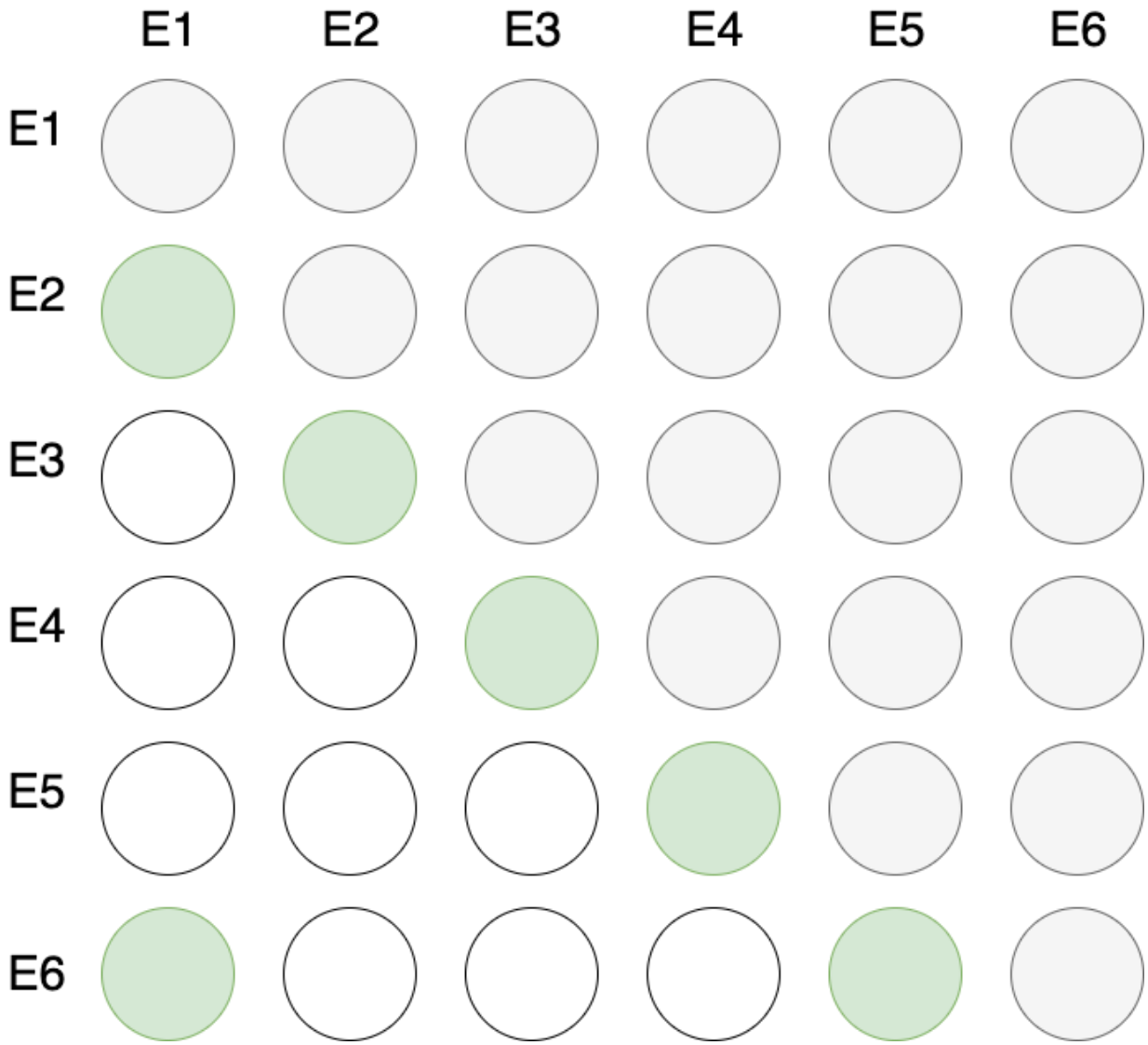


Рисунок 3.3 - Випадок з циклічним графом сутностей модуля

Випадок на рисунку 3.3 демонструє ту саму кількість залежностей між сутностями модуля, але цього разу граф зв’язаний, що значить, що кожна сутність опосередковано зв’язана з іншими.

Зверніть увагу, що якщо ми будемо розраховувати сполученість модуля за запропонованою формулою, вона буде однакова для обох випадків, хоча абсолютно очевидно, що другий випадок мусить мати кращу сполученість за перший, так як перший, очевидно, має бути розділено на два незалежні модуля.

Таким чином, щоб вирішити цю проблему непов’язаних підгруп, пропонується ввести додаткове обмеження. Якщо в графі трапляються непов’язані між собою підгрупи елементів - вони вважаються інородними, і мають знизити показник сполученості модуля. У випадку, коли граф складається з декількох окремих графів, що не пов’язані між собою - за основний підграф (сутності якого вважаємо пов’язаними між собою) беремо підграф з найбільшою кількістю вершин і ігноруємо всі зв’язки в інших підграфах.

Відобразимо цю модель математичною формулою:

$$MC = RCE \,/\, ((AE^2 - AE)/2), AE \notin \otimes \quad (3.7)$$

де,

- MC (module cohesion) - сполученість модуля;
- RCE (Relations of Connected Entities) - загальна кількість відносин між пов’язаними вершинами найбільшого зв’язаного підграфу;
- AE (All Entities) - всі сутності модуля;
- $((AE^2 - AE)/2)$- загальна можлива кількість не напрямлених відносин між усіма сутностями модуля (рис. 3.1).

Тепер, метрика більш точно виражає сполученість модуля. При розрахунку сполученості модуля використовуючи дану метрику, перший випадок представлений на рисунку 3.2 буде мати вдвічі меншу сполученість за випадок представлений на рисунку 3.3. Також варто звернути увагу, що випадок 3.3 буде мати однакове значення сполученості розраховане за оригінальною формулою метрики та розраховане за покращеною формулою метрики.

3.3.3.6. Сполученість модуля без дітей

Якщо модуль не має жодної сутності, формула набуває таких значень:

$$MC = 0/((0^2 - 0)/2) = 0/(0/2) = 0/0 \quad (3.8)$$

В такому випадку його сполученість має дорівнювати 0, так як модуль без жодного члена не має сенсу.

3.3.3.7. Сполученість модуля з однією дитиною

Якщо модуль має лише одну дитину, формула перетворюється у наступний вираз:

$$MC = 0/((1^2 - 1)/2) = 0/(0/2) = 0/0 \quad (3.9)$$

, що неприпустимо з точки зору математики.

Таким чином, маємо вирішити якого значення має набувати сполученість модуля у цьому випадку, і, зважаючи на те що модуль не пустий і його дитина ніяк не конфліктує з собою пропонується повертати 1.

3.3.4. Метрика модульності

Модульність для класів буде еквівалентна:

- Для класів:

$$M = (1 - LCOM) * (1 - I) \quad (3.10)$$

- Для модулів та файлів:

$$M = MC * (1 - I) \quad (3.11)$$

- Для інших сутностей:

$$M = (1 - I) \quad (3.12)$$

, де:

- M - модульність;
- MC - сполученість модулів;
- 1 - LCOM - згуртованість методів;
- 1 - I - стабільність.

M знаходиться в діапазоні [0;1]. Таким чином, чим менше значення - тим менша модульність обраної сутності.

3.5. Висновки

В даному розділі ми дослідили можливі фактори, що впливають на якісні характеристики програмного забезпечення, обрали ключові з них та визначили нову метрику, що повністю або частково характеризує кожну з характеристик.

Також в ході дослідження була покращена метрика сполученості модуля, що передбачає врахування зв’язності графу модуля.

РОЗДІЛ 4 ОПИС ПРОГРАМНОГО ТА ТЕХНІЧНОГО ЗАБЕЗПЕЧЕННЯ

Дана робота передбачає аналіз програмного забезпечення в середі TypeScript. Дуже важливо, що це не вимагає, щоб дане програмне забезпечення було написано на даній мові, але мовою програмування було обрано TypeScript через строгу типізацію і універсальність. До того ж дана система зможе проаналізувати сама себе у випадку написання на TypeScript, що є великим плюсом.

4.1. Вимоги до програмного продукту

Програмний продукт має:

- Запускатися з терміналу
- Збирати такі дані:
    - Сутності:
        - Класи
            - Чи є клас абстрактним
        - Файли
            - Шлях від кореня проекту
        - Простори імен
        - Функції
        - Перерахування
        - Інтерфейси
        - Типи
        - Змінні
    - Загальні дані кожної сутності:
        - Назва сутності
        - Тип сутності
        - Чи є сутність внутрішньою відносно проекту

    - Дані сутностей з коду:
        - Кількість стрічок коду
        - Шлях до файлу в якому сутність знаходиться відносно кореня проекту
    - Зв’язки між сутностями:
        - Включення (в якій сутності дана сутність об’явлена)
        - Розширення (наслідування)
        - Реалізація типу (implements)
        - Використовується
- Мусить будувати граф програми
- Мусить аналізувати граф програми за обраними в попередньому розділі метриками
- Має зберігати в файл звіт, що включатиме:
    - Список сутностей, в якому кожна сутність має:
        - Назва сутності
        - Шлях до сутності з кореня проекту, що включає:
            - Шлях до файлу
            - Назву файлу
            - Всі сутності, що включають в себе об’явлену сутність в ієрархічному порядку
        - Загальний показник якості, що вираховується як середнє арифметичне всіх наявних метрик сутності
        - Кожну метрику сутності
            - Назва метрики
            - Показник у відсотках
    - Глобальні метрики проекту
        - Назва метрики
        - Показник у відсотках

Архітектура має відповідати таким вимогам:

- Можливість підтримки всіх можливих мов програмування і всіх можливих мов опису архітектури програмного забезпечення
- Підтримувати всі основні платформи для розробки:
    - Windows
    - Linux
    - Mac OS
- Спосіб опису метрик, методів їх збору і правил валідації архітектури має бути універсальним для програмістів незалежно від специфіки їх середи розробки і мови програмування
- Додаток мусить надати можливість легко розширяти метрики для валідації
    - Розробникам системи
    - Користувачам (інженерам програмного забезпечення, які використовують дану систему)
- Додаток має надавати можливість легко розширяти інформацію, що може аналізуватися (наприклад, нові сутності, типи зв’язків, рівні деталізації тощо.)
- Додаток має надавати можливість відображати дані у декількох форматах (залежно від потреби користувача)
    - Графічний формат
        - Включає представлення графа програми
        - Наявні фільтри
            - По сутності
            - По типу зв’язків
            - По метрикам
        - Доступне групування сутностей за:
            - Метриками
            - Просторами імен

    - Ієрархією директорій в проекті
  - Можливість побачити глобальні метрики
  - Можливість отримати всю детальну інформацію по сутності, що цікавить
- Файловий формат (описаний у вимогах до програмного продукту)

4.2. Архітектура програмного забезпечення

Щоб підготувати архітектуру за визначеними вимогами, треба спершу розбити дану систему на модулі.

Рисунок 4.1 - Процес роботи системи представлений її модулями

На рисунку 4.1 представлені модулі системи, що описують весь процес її роботи.

Спочатку парсер збирає всі необхідні дані з вихідного коду програми. Основна його задача - зібрати всю інформацію про сутності та зв’язки між ними. В залежності від налаштувань, він може аналізувати тільки внутрішні файли, або також і залежності.

Далі всі зібрані дані поступають до будівельника графу. Він з розрізненої інформації про сутності та їх зв’язки будує графове представлення, з яким в свою чергу вже працюють всі інші модулі програми.

Наступним кроком графове представлення потрапляє до аналізатора, який збирає всі метрики і доповнює ними наявні дані.

Після того як всі метрики зібрані, граф разом з усіма зібраними даними потрапляють до валідатора, який в свою чергу являє собою набір тестів. Він являється опціональним і може вимикатися якщо треба тільки проаналізувати програму, або переходити в попереджувальний режим (замість помилок, залишає інформацію про попередження).

Останній крок - це представлення результатів. Це може бути як веб-сторінка для аналізу, так і вивід в файл або термінал.

Тепер ознайомимося більш детально з проблемами та вимогами до кожного модуля окремо.

4.2.1. Парсер

Згідно до вимог до архітектури системи, вона має підтримувати різні мови програмування та мови опису архітектури. Для парсера це означає, що саме йому доведеться розуміти різні мови і повертати інформацію по протоколу, для того, щоб не доводилося дублювати код всіх інших модулів для кожної окремої мови.

Таким чином, для парсера необхідно протокол (див. рис. 4.2). Після його введення виходить, що парсер - це модуль, що відповідає за збір даних зі специфічної мови програмування і передачу їх до будівельника графу за спільним протоколом.

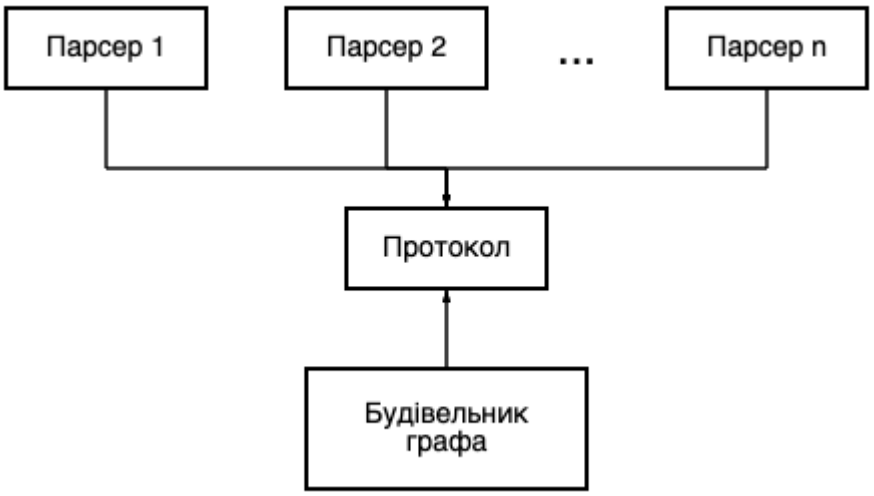


Рисунок 4.2 - Введення протоколу

Але є одна проблема. Більшість бібліотек, що допомагають парсити деяку мову програмування, написані на цій же мові. Звідси робимо висновок, що парсери можуть бути написані на різних мовах програмування. Це означає, що потрібно знайти можливість якось передавати дані між парсером написаним на деякій мові і системою написаній на TypeScript.

Рішення просте - підпрограми. Основна програма може викликати парсери як підпрограми, де протокол грає виключну роль гаранту формату даних і їх складу.

Тепер, після введення підпрограм, стає проблема гарантування протоколу і виклику підпрограм. Для цього, як видно на рисунку 4.3, необхідно ввести додаткову сутність - адаптер.

Адаптер буде відповідати за визначення які саме парсери треба викликати, їх безпосередній виклик, перевірку даних що надійшли і їх парсинг (бо вони надійдуть в якомусь загальному форматі: json, xml, тощо., а їх треба буде перетворити в типізовані сутності, для подальшої роботи). Після цих процедур, дані переходять далі, в будівельник графу.

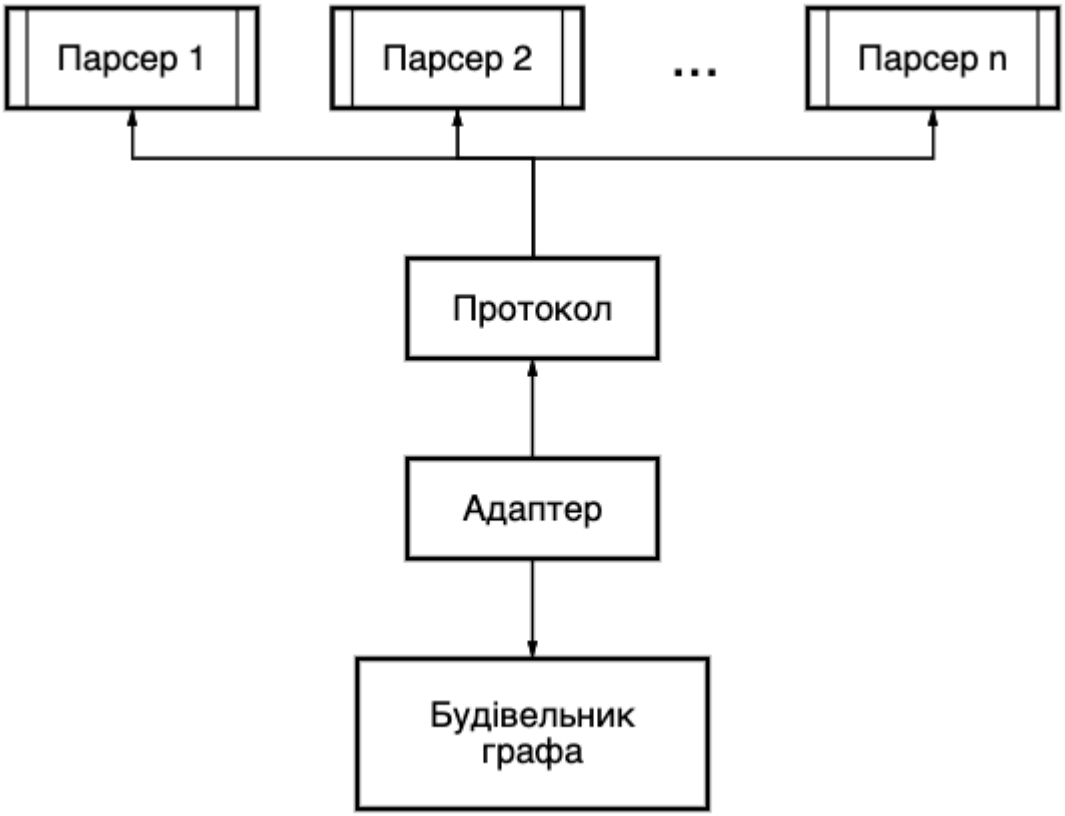


Рисунок 4.3 - Введення адаптера і перетворення парсерів на підпрограми

Також важливо розуміти, що одночасно парсер може повернути 1..n наборів даних, відповідно до налаштувань системи. Наприклад, якщо ми захочемо перевірити, що архітектура описана в UML є підграфом вихідного коду проекту, нам необхідно буде отримати спарсити набір даних UML документу і набір даних вихідного коду.

Таким чином кожен з модулів має працювати з декількома наборами даних одночасно.

4.2.2. Контроллер

Контроллер - це сутність диригент. Він відповідає за те коли і які модулі будуть викликатися в залежності від зовнішньої конфігурації. Безпосередньо, крім контроллера, жоден модуль не має знати про будь-який

інший. Тільки контроллер викликає функції з модулів, отримує результат і передає його далі (рисунок 4.4).

Рисунок 4.4 - Контроллер модулів системи

4.2.3. Будівельник графа

Будівельник графа приймає на вхід набір наборів даних і повертати набір графів. Це найпростіший етап, основна задача якого пов’язати між собою всі вузли оптимальним чином і підготувати їх до подальшої обробки.

4.2.4. Аналізатор

Аналізатор виконує дуже важливу роль. Він збирає інформацію з графів та отримує необхідні метадані - метрики. Далі вони можуть використовуватися валідатором, або ж просто для мануального аналізу.

Найскладніше - це те, що необхідно надати універсальну можливість писати та додавати свої власні метрики. При цьому треба враховувати кілька дуже важливих проблем:

- Метрики можуть залежати від інших метрик (циклічність строго заборонена)
- Інженери програмного забезпечення можуть володіти абсолютно різними мовами програмування
- Метрики можуть вираховуватися по рідному для різних сутностей

Звісно, можна використати для цього SQL, але це не оптимально і не очевидно з точки зору використання, особливо зважаючи на проблему з взаємозалежністю метрик. Тож треба або створити спеціальну нотацію для

цього, що теж не кращий варіант, через необхідність задання алгоритму обчислень метрики. Тож залишається запропонувати декілька мов програмування, що можуть використовуватися для цілі завдання нових метрик і параметрів.

Обов’язково має бути перевірка на унікальне ім’я метрики, але де його взяти? Нехай ім’ям буде ім’я функції, що обчислює цю метрику. Таким чином користувач має створити файл на запропонованій мові, який буде включати тільки функції, які обчислюють метрику.

Наступне питання в тому, як це має працювати з точки зору системи? Відповідь полягає знов у підпрограмах. Алгоритм полягає у розподілі обчислень на 2 основні етапи:

1. Обчислення заготовлених метрик
2. Обчислення користувацьких метрик

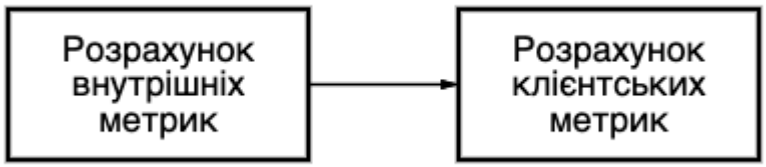


Рисунок 4.5 - Порядок розрахунку метрик

Перший етап має пройти всередині головної програми системи, і після того як обчислення завершені - основна програма викликає підпрограму (її контроллер), якій передає дані (графи та їх вже підраховані метрики) і викликаючи функції заготовлені користувачем доповнює їх. Коли всі користувацькі метрики обраховані - підпрограма повертає тільки нові метрики головній програмі. Коли головна програма спарсила ці дані - цей етап вважається завершеним.

На рисунку 4.5. можна побачити схему роботи аналізатора. Дуже важливо звернути увагу, що адаптерів може бути багато і який адаптер (або адаптери) виконується - залежить від конфігурації (яке розширення у файла з функціями).

Рисунок 4.6 - Схема роботи аналізатора

4.2.5. Валідатор

Валідатор на відміну від аналізатора не має визначеного порядку через відсутність взаємозалежності серед функцій, що перевіряють відповідність заданим правилам, тож його роботу можна розпаралелити і виконувати визначену кількість функцій одночасно. Також, як визначено на рисунку 4.7, для ще більшої ефективності, внутрішні та клієнтські правила виконуються паралельно. Це має бути ефективнішим через те, що підпрограма має виконуватися на окремому ядрі процессора.

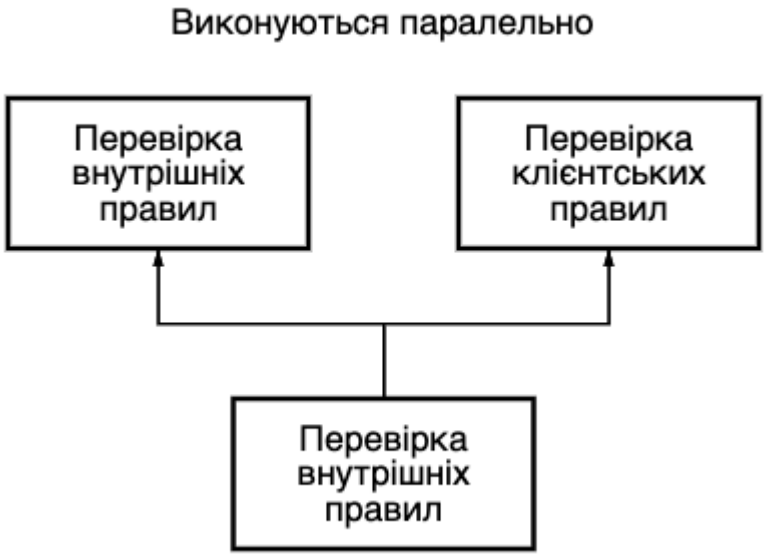


Рисунок 4.7 - Порядок виконання перевірок

Валідатори не просто виконуються, а мають повернути звіт, зміст якого складається з:

- Назва перевіряємої сутності
- Назва правила
- Статус правила (Є чи немає помилки)
- Опціонально текст помилки
- Рівень значимості помилки

Сам валідатор може просто викинути помилку, всю іншу інформацію можна отримати ззовні.

Клієнтські валідатори так само як і метрики додаються через підпрограму та вкладений контроллер (аналогічна архітектура).

Важливо відмітити, що валідатори самі обробляють результати своєї роботи (окремий вкладений модуль). З модуля валідатора вже повертаються готові для відображення дані.

4.2.6. Представлення результату

Модуль представлення результатів отримує вже готові дані для відображення. Для нього важливо зазначити декілька ключових факторів:

- Він може налічувати декілька різних варіантів представлення результату:
    - вивід у файл(и);
    - вивід у термінал;
    - представлення у вигляді веб сторінки;
- Різні варіанти представлення результату можуть використовуватися разом. Наприклад вивід у файл та вивід у термінал;
- Програма має закінчитися успішно або з помилкою в залежності від того, які налаштування (чи режим тестування чи ні) та наявності помилок.

Звідси ми бачимо, що кожен окремий тип представлення даних має бути окремим підмодулем зі спільним інтерфейсом для контроллера представлень.

Також має бути окремий підмодуль для завершення програми. Саме він вирішує як вона має бути завершена, з помилкою (та з якою) чи успішно.

Дану архітектуру можна розглянути на рисунку 4.8.

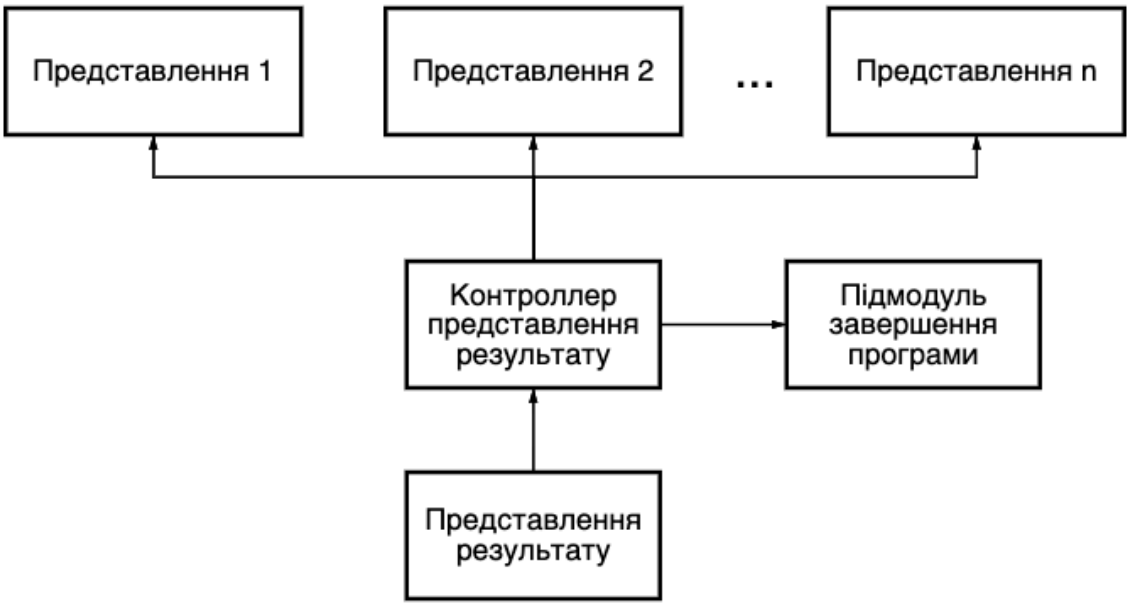


Рисунок 4.8 - Архітектура модуля представлення результату

4.3. Розподілення по пакетам

Для легкого масштабування і зручного використання, деякі частини цієї системи мають бути виділені в окремі пакети.

Перелік виокремленого функціоналу:

- Кожний парсер має бути в окремому пакеті
- Протокол між парсером та його адаптером
- Кожний аналізатор (підпрограма) має бути виокремлений в окремий пакет
- Протокол між аналізатором та основною програмою
- Кожний валідатор (підпрограма) має мути виокремлений в окремий пакет
- Протокол між валідатором та основною програмою

Важливо відмітити, що представлення результату не вважається кастомізуємою частиною і його представлення не мають бути відокремлені від основного репозиторія.

4.4. MVP

Важливо відмітити, що через обмежений час, реалізувати дану архітектуру самостійно не можливо, тож в рамках MVP для даної дисертації вона буде значно спрощена.

Підпрограми не будуть реалізовані і вся логіка буде знаходитися в рамках основного процесу.

Розширюваність програми буде значно зменшена через заміну протоколів розподілом на модулі TypeScript. Таким чином, гнучке додавання додаткових парсерів, метрик та валідаторів буде вимагати зміни основного коду програми.

Конфігурації будуть скорочені. Наприклад, не буде можливості отримати інформацію про зовнішні залежності.

4.4. Вибір технологій для реалізації програмного забезпечення

Задача цього підрозділу обґрунтувати вибір технологій для реалізації даної системи.

4.4.1. Парсинг в середовищі TypeScript

Для збору даних недоцільно писати свій власний парсер і необхідно знайти вже готову бібліотеку, що автоматизує збір даних і інкапсулює логіку парсингу токенів і їх взаємозв’язків.

Як ми пам’ятаємо, парсерів може бути багато, але для даної роботи, нас цікавить саме парсер для TypeScript. В даному сегменті, багато бібліотек не знайшлося для виконання даної функції, тож було дуже легко і однозначно обрати бібліотеку ts-morph. Вона надає всю необхідну інформацію і доступ у форматі дерева до всіх елементів програми, тож повністю покриває всі необхідні потреби.

Хоча ця бібліотека і надає представлення графом всієї необхідної інформації, для того, щоб закласти основу протоколу і можливість розширити програму, в MVP було створено тип, який грає роль внутрішнього протоколу. Саме до цього типу необхідно привести зібрані дані, щоб передати на обробку подальшим модулям.

4.4.2. Протоколи

Основними претендентами на мови опису протоколів є Protocol Buffer та Apache Avro. Ці дві мови дозволяють декларувати протоколи та використовувати їх для різних мов програмування.

В цілому, ці два варіанти мало чим відрізняються одне від одного, тож обирати слід більш знайомий. У них є свої переваги та недоліки, але вони дуже незначним образом впливають на результат.

Для реалізації MVP було обрано спростити протоколи до внутрішньої типізації в TypeScript, через те, що це значно економить час.

4.4.3. Допоміжні утиліти

Для забезпечення високої якості та спрощення коду було використано додаткову бібліотеку утиліт lodash. Вона налічує велику кількість додаткових функцій, які можуть використовуватися для покращення та спрощення програмного коду.

4.5. Опис розробленої системи

Розроблена система обмежена збором метаданих програмного забезпечення, обчисленням метрик на їх основі та виводом результатів в консоль.

4.5.1. Сутності програмного коду які підтримуються

Розроблена версія програмного забезпечення підтримує такий список сутностей:

- Файл;
- Простір імен (являє собою модуль);
- Клас;
- Функція;
- Перерахування;
- Інтерфейс;
- Змінна;
- Тип (являє собою псевдонім деякого типу в середовищі TypeScript);
- Метод (відноситься до класів);
- Конструктор;
- Поле (відноситься до класів);
- Геттер;
- Сеттер;
- Функція об’явлена в змінній;
- Статичний блок в класі;
- Функція об’явлена як поле класу;

В реалізованій системі ці сутності описані перерахуванням (див. рисунок 4.9).

```
export enum EntityType {
  File = 1,
  Namespace = 2,
  Class = 3,
  Function = 4,
  Enum = 5,
  Interface = 6,
  Variable = 7,
  Type = 8,
  Method = 9,
  Constructor = 10,
  Field = 11,
  Getter = 12,
  Setter = 13,
  FunctionVariable = 14,
  ClassStaticBlock = 15,
  FunctionProperty = 16,
  Unknown = 17
}
```

Рисунок 4.9 - Підтримувані розробленою системою сутності

Вони є ключовими і достатніми для опису будь-якої програми на TypeScript.

4.5.2. Реалізація графа програми

Для ефективності роботи з графом під час його обробки та обчислень метрик він був реалізований наступним чином. На рисунку 4.10 зображено інтерфейс, що описує граф програми яка аналізується. З нього видно, що граф складається з вершин, які для зручності одразу згруповані по типам сутностей а також доступні повним списком, і зв’язків (відношень) між вершинами (як видно на рисунку 4.11, базовий інтерфейс GraphRelationsOwner декларує список відносин). Граф містить повний список відношень між сутностями програми.

Як видно з рисунка 4.11, кожна вершина також має список відносин. Це повний список відносин в яких дана вершина фігурує, що реалізований

по посиланню. Це означає, що кожні відносини і кожна вершина зустрічаються в оперативній пам’яті тільки раз, але при цьому можна легко отримати з будь-якої вершини доступ до всіх пов’язаних вершин, а з них до вершин пов’язаних з ними і так далі.

```
/**
 * Every node has relations to other related nodes.
 * Also the graph entity contains all of the relation into a single field to make it easier
 * search operations.
 */
export interface Graph extends GraphRelationsOwner {
  files: FileGraphNode[];
  namespaces: NamespaceGraphNode[];
  classes: ClassGraphNode[];
  functions: FunctionGraphNode[];
  enums: EnumGraphNode[];
  interfaces: InterfaceGraphNode[];
  variables: VariableGraphNode[];
  types: TypeGraphNode[];
  constructors: ConstructorGraphNode[];
  methods: MethodGraphNode[];
  fields: FieldGraphNode[];
  functionProperties: FunctionPropertyGraphNode[];
  functionVariables: FunctionVariableGraphNode[];
  getters: GetterGraphNode[];
  setters: SetterGraphNode[];
  classStaticBlocks: ClassStaticBlockGraphNode[];
  allNodes: SupportedGraphNodes[];
}
```

Рисунок 4.10 - Інтерфейс графа аналізуємої програми

Для роботи з графом використовуються спеціально підготовлені функції помічники, які виконують вузькі специфічні функції, такі як, наприклад, знайти всіх дітей вглиб сутності (тобто всі сутності, що задекларовані всередині даної), що, безумовно, необхідно для визначення чи використовує одна сутність іншу (якщо будь-хто з дітей на будь-якій глибині використовує іншу сутність - вважається, що дана сутність теж її використовує).

Також такі функції помічники відіграють дуже важливу роль, інкапсулюючи реалізацію деякого алгоритму для роботи з графом в одному місці, що дозволяє покращити модульність даної системи.

```typescript
export interface GraphRelation {
  id: string;
  node1: GraphNode<EntityType>;
  node2: GraphNode<EntityType>;
  type: RelationType;
}

export interface GraphRelationsOwner {
  relations: GraphRelation[];
}

export interface GraphNode<T extends EntityType> extends GraphRelationsOwner {
  id: string;
  name: string;
  entityType: T;
  isInternal: boolean;
}

export interface GraphCodeNode<T extends Exclude<EntityType, EntityType.File>> extends GraphNode<T> {
  parentFile: FileGraphNode;
}
                                    interface GraphNode<T extends EntityType>
export interface FileGraphNode extends GraphNode<EntityType.File> {
  path: string;
}
export interface NamespaceGraphNode extends GraphCodeNode<EntityType.Namespace> {}
export interface ClassGraphNode extends GraphCodeNode<EntityType.Class> {
  isAbstract: boolean;
}
export interface FunctionGraphNode extends GraphCodeNode<EntityType.Function> {}
export interface EnumGraphNode extends GraphCodeNode<EntityType.Enum> {}
export interface InterfaceGraphNode extends GraphCodeNode<EntityType.Interface> {}
export interface VariableGraphNode extends GraphCodeNode<EntityType.Variable> {}
export interface TypeGraphNode extends GraphCodeNode<EntityType.Type> {}
export interface ConstructorGraphNode extends GraphCodeNode<EntityType.Constructor> {}
export interface MethodGraphNode extends GraphCodeNode<EntityType.Method> {}
export interface FieldGraphNode extends GraphCodeNode<EntityType.Field> {}
export interface FunctionPropertyGraphNode extends GraphCodeNode<EntityType.FunctionProperty> {}
export interface FunctionVariableGraphNode extends GraphCodeNode<EntityType.FunctionVariable> {}
export interface GetterGraphNode extends GraphCodeNode<EntityType.Getter> {}
export interface SetterGraphNode extends GraphCodeNode<EntityType.Setter> {}
export interface ClassStaticBlockGraphNode extends GraphCodeNode<EntityType.ClassStaticBlock> {}
```

Рисунок 4.11 - Інтерфейси вершин графу та зв'язків між ними

4.5.3. Реалізація метрик

В розробленій системі, метрики реалізовані таким чином, щоб легко додавати нові метрики та було можливо створювати залежності між метриками.

Щоб додати такі можливості, були виділені базові типи метрик:

- Метрика сутності;
- Метрика групи сутностей;
- Глобальна метрика.

Відповідно, кожна з метрик розраховується по різному (див. рисунок 4.12).

Для розрахунку метрики сутності, метрика приймає параметрами сутність та вже розраховані значення залежних метрик сутності і повертає число (важливо зазначити, що кожна метрика мусить повернути число).

Метрика групи сутностей приймає параметрами групу сутностей визначеного метрикою типу та вже підраховані метрики для цих сутностей і повертає число.

Глобальна метрика в свою чергу приймає граф, що виражає всю програму (описує всі сутності та їх зв’язки між собою), та всі попередньо зібрані метрики. Це необхідно для обчислення деякої загальної для всієї програми метрики.

Важливо зазначити, що в даній версії програми не була реалізована функція визначення залежностей між метриками та автоматичного визначення порядку розрахунків. Це зумовлено обмеженнями часу і тим, що це не критичний функціонал для демонстрації роботи даної програми. Тож, порядок розрахунку метрик в програмі в даній версії був визначений вручну і вимагає редагування при необхідності зміни порядку визначення метрик. Також важливо звернути увагу, що цей механізм легко може бути додано без значної зміни вихідного коду системи т.я. всі необхідні для нього дані вказані в кожній з метрик.

```typescript
export interface Metric {
  id: string;
  name: string;
  description: string;
  type: MetricType;
  dependencies?: Metric[];
}

export type EntityMetricHandlerParams<T extends EntityType = any> = {
  node: GraphNode<T>;
  dependencies?: MetricDependencies;
};
export interface EntityMetric<T extends EntityType = any> extends Metric {
  type: MetricType.Entity;
  supportedEntities: EntityType[];
  handler: (data: EntityMetricHandlerParams<T>) => number;
}

export type EntityGroupMetricHandlerParams<T extends EntityType = any> = {
  nodes: GraphNode<T>[];
  collectedMetrics: AnalyticsData;
};
export interface EntityGroupMetric<T extends EntityType = any> extends Metric {
  type: MetricType.EntitiesGroup;
  supportedEntities: EntityType[];
  handler: (data: EntityGroupMetricHandlerParams<T>) => number;
}

export type GlobalMetricHandlerParams = {
  graph: Graph;
  collectedMetrics: AnalyticsData;
};
export interface GlobalMetric extends Metric {
  type: MetricType.Global;
  handler: (data: GlobalMetricHandlerParams) => number;
}
```

Рисунок 4.12 - Реалізовані інтерфейси метрик

4.5.2.1. Нестабільність

Дана метрика була реалізована для всіх визначених елементів, т.я. кожен з них може мати залежності та може використовуватися іншими сутностями.

На рисунку 4.13 можна побачити функцію обчислення даної метрики.

4.5.2.2. LCOM

Дана метрика була реалізована тільки для класів, т.я. призначена тільки для них. На рисунку 4.14 зображений її вихідний код.

```
handler: ({node}: EntityMetricHandlerParams): number => {
  // For now the instability is only internal, as we are omitting external project dependencies
  const relations = [node, ...getChildrenDeep(node)]
    .reduce<InstabilityRelations>(((rels, n) => {
      for (let rel of n.relations) {
        if (
          [
            RelationType.Extends,
            RelationType.Implements
          ].some(relType => relType === rel.type)
        ) {
          if (rel.node1.id === n.id) {
            rels.outcoming.push(rel);
          } else {
            rels.incoming.push(rel);
          }
        }

        if (rel.type === RelationType.UsedBy) {
          if (rel.node2.id === n.id) {
            rels.outcoming.push(rel);
          } else {
            rels.incoming.push(rel);
          }
        }
      }
      return rels;
    }), {incoming: [], outcoming: []});

  if (!relations.incoming.length && !relations.outcoming.length) {
    return 0;
  }

  return relations.outcoming.length / (relations.incoming.length + relations.outcoming.length);
}
```

Рисунок 4.13 - Функція обчислення метрики “Нестабільність”

```
export const LCOM: EntityMetric<EntityType.Class> = {
  id: "LCOM",
  name: "LCOM",
  description: "Lack of Cohesion Of Methods",
  type: MetricType.Entity,
  supportedEntities: [EntityType.Class],
  handler: ({node}: EntityMetricHandlerParams): number => {
    const collableChildren = getCollableChildren(node);
    const fields = getDataChildren(node);

    if (!fields.length) {
      return 1;
    }

    if (!collableChildren.length) {
      return 0;
    }

    const mfSum = fields
      .map(field =>
        collableChildren.filter(collable => isUsedBy(field, collable)).length
      )
      .reduce((sum, mf) => sum + mf, 0);

    return 1 - mfSum / (collableChildren.length * fields.length);
  }
}
```

Рисунок 4.14 - Вихідний код метрики LCOM

4.5.2.3. Сполученість Модуля

Дана метрика була реалізована для обчислення сполученості модулів. Як модулі ми можемо представити файли та простори імен. Саме тому на рисунку 4.15 можна побачити, що дана метрика передбачає обчислення їх обох.

```
export const ModuleCohesion: EntityMetric<EntityType.Namespace | EntityType.File> = {
  id: "Module Cohesion",
  name: "Module Cohesion",
  description: "Cohesion of a module",
  type: MetricType.Entity,
  supportedEntities: [EntityType.Namespace, EntityType.File],
  handler: ({node}): number => {
    const children = getChildren(node);

    if (!children.length) {
      return 0;
    }

    // Actually not sure
    if (children.length === 1) {
      return 1;
    }

    const subgraphs = getSubgraphs(children);

    const largestSubgraph = subgraphs.reduce((largest, subgraph) => {
      return largest.length < subgraph.length ? subgraph : largest;
    }, []);

    const subgraphRelationsCount = getSubgraphRelationsCount(largestSubgraph);

    // With `- children.length` we are removing medium diagonal of the matrix
    // And with division for 2 we are removing mirror dependencies as our dependencies are without direction
    return subgraphRelationsCount / ((children.length * children.length - children.length) / 2);
  }
};
```

Рисунок 4.15 - Вихідний код метрики “Сполученість Модуля”

4.5.2.4. Модульність

Дана метрика для обчислення значення використовує всі попередньо визначені метрики і може бути обчислена для будь-якої сутності (див. рисунок 4.16).

Важливо зазначити, що вона обчислюється по різному в залежності від типу сутності, для якої вона обчислюється. Слід виділити класи, файли та простори імен серед інших.

Класи є сутністю, що представляють шаблон об’єкту, в якому будуть знаходитися дані та пов’язані з ними методи. Таким чином, вони є основним елементом ООП і значно відрізняються від всіх інших сутностей, тож модульність розраховується для них по іншому.

Модулі, якими являються файли та простори імен, є основними контейнерами для інших типів сутностей і їх обчислення теж відрізняється.

```
handler: ({node, dependencies}: EntityMetricHandlerParams) => {
  const instability = dependencies?.[Instability.id];

  if (!isNumber(instability)) {
    throw new Error("Instability should be already calculated");
  }

  const stability = 1 - instability;

  if (node.entityType === EntityType.Class) {
    const lcom = dependencies?.[LCOM.id];

    if (!isNumber(lcom)) {
      throw new Error("LCOM should be already calculated!");
    }

    const cohesion = 1 - lcom;

    return stability * cohesion;
  }

  if (
    [
      EntityType.Namespace,
      EntityType.File
    ].some((type: EntityType) => type === node.entityType)
  ) {
    const cohesion = dependencies?.[ModuleCohesion.id];

    if (!isNumber(cohesion)) {
      throw new Error("Cohesion should be already calculated");
    }

    return stability * cohesion;
  }

  return stability;
}
```

Рисунок 4.16 - Реалізація функції обчислення метрики “Модульність”

4.5.4. Вивід результатів

Вивід результатів реалізовано через інтерфейс командного рядка (рис. 4.17).

```
MacBook-Pro-Nick-2:Arcan nickvoloshyn$ ts-node index.ts

##################
# Entity Metrics #
##################

Class "BaseClass" (children of "test.ts")
------------------------------------------------------------------
- Instability: 44.44%
- LCOM: 100.00%
- Modularity: 0.00%

Class "OutsideClass" (children of "BaseClass")
------------------------------------------------------------------
- Instability: 56.25%
- LCOM: 100.00%
- Modularity: 0.00%

Class "OtherClass" (children of "InsideNamespace")
------------------------------------------------------------------
- Instability: 0.00%
- LCOM: 0.00%
- Modularity: 100.00%

Class "TestClass" (children of "SomeTestNamespace")
------------------------------------------------------------------
- Instability: 42.86%
- LCOM: 0.00%
- Modularity: 57.14%

Class "SomeClass" (children of "SomeTestNamespace")
------------------------------------------------------------------
- Instability: 100.00%
- LCOM: 0.00%
- Modularity: 0.00%

Constructor "Constructor" (children of "BaseClass")
------------------------------------------------------------------
- Instability: 66.67%
- Modularity: 33.33%

Constructor "Constructor" (children of "OutsideClass")
------------------------------------------------------------------
- Instability: 66.67%
- Modularity: 33.33%

Constructor "Constructor" (children of "TestClass")
------------------------------------------------------------------
- Instability: 60.00%
- Modularity: 40.00%

Enum "SomeTestEnum" (children of "test.ts")
------------------------------------------------------------------
- Instability: 0.00%
- Modularity: 100.00%

Enum "SomeEnum" (children of "some")
------------------------------------------------------------------
- Instability: 0.00%
- Modularity: 100.00%

Enum "EntityType" (children of "entity.ts")
```

Рисунок 4.17 - Вивід результатів в командний рядок

Як видно з рисунка 4.17, вивід відбувається в такому форматі:

1. Назва групи метрик (глобальні, групові або метрики сутностей)
2. Список Кожного елементу групи метрик
    a. Для глобальних метрик це буде просто список глобальних метрик та їх значень
    b. Для метрик груп сутностей - це групи сутностей (класи, інтерфейси тощо.)
    c. Для метрик елементів - це елементи

Якщо якась група метрик не розраховується - інформація про неї не виводиться. Наприклад, так як ми розраховуємо тільки метрики для сутностей, метрики для груп сутностей та глобальні метрики не будуть виведені.

Якщо елемент групи метрик не є метрикою - кожен елемент буде відображати список розрахованих метрик для нього (див. рисунок 4.17).

Всі метрики, що розраховують значення в діапазоні [0;1] у виводі результатів переводяться у відсотки для покращення сприйняття даної інформації.

4.6. Висновки

В даному розділі було розглянуто реалізацію системи виявлення архітектурних помилок, її архітектурні особливості та вибір технологій для реалізації. Також розглянуто її інтерфейс та реалізацію метрик, що реалізовують розроблені алгоритми в минулому розділі.

РОЗДІЛ 5 РОЗРОБКА СТАРТАП-ПРОЕКТУ

5.1. Опис ідеї проекту

Опис ідеї проекту та способи застосування описані в таблиці 5.1.

Таблиця 5.1 - Опис ідеї стартап-проекту

Зміст ідеї	**Напрямки застосування**	**Вигоди для користувача**
Розробка системи виявлення архітектурних помилок програмного забезпечення в середовищі TypeScript	1. Аналіз існуючого проекту в розрізі архітектури	Можливість набагато краще за менший час зрозуміти які в проекті є сутності та як вони взаємодіють між собою. Це корисно як для новачка, що тільки прийшов на проект, так і для досвідченого розробника на проекті, щоб зекономити час і спланувати необхідні зміни.
	2. Контроль якості архітектури проекту	Можливість автоматизувати перевірку якості архітектури як кожної з компонент так і системи в цілому. Це дозволяє попередити помилки окремих розробників і покращити якість рішень.

Продовження Таблиці 5.1

Зміст ідеї	**Напрямки застосування**	**Вигоди для користувача**
	3. Перевірка на відповідність архітектури проекту до архітектури описаної мовою опису архітектури	Це дозволить не просто орієнтуватися на архітектуру описану мовою опису архітектури проекту, а перевіряти проект на відповідність до описаної архітектором архітектури на входження. Гарантує, що архітектура додатку реалізована як задумано.

Система виявлення архітектурних помилок програмного забезпечення дозволяє аналізувати архітектуру, виявляти помилки та контролювати якість реалізації системи, що перевіряється.

Довгострокові перспективи такі:

- Можливість групувати представлення сутностей у графі за типом та спираючись на метрики;
- Можливість аналізувати вихідний код включно з його зовнішніми залежностями (бібліотеками);
- Можливість зробити опис архітектури створений архітектором єдиним джерелом істини;
- Можливість фільтрувати дані в графічному представленні по сутностям та метрикам.

У таблиці 5.2 можна побачити характеристики ідеї відносно конкурентів.

Таблиця 5.2 – Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	Проекти конкурентів			W слабка сторона	N нейтральна сторона	S сильна сторона
		Мій проект	Visual Studio	NDepend			
1.	Візуальний аналіз графу програми	Дозволяє аналізувати граф програми, використовуючи фільтри та групування	Дозволяє аналізувати граф програми без можливості прибрати зайві елементи або згрупувати необхідним шляхом	Дозволяє аналізувати граф програми у декілька способів використовуючи графове та матричне представлення даних. Передбачає групування кластерами (недостатньо гнучке) та не передбачає фільтрацію			+
2.	Кросплатформеність	Підтримує всі основні платформи для розробки: Windows Linux MacOS	Доступний на Windows у повній версії та урізаний на MacOS. Для Linux недоступний	Являє собою плагін для Visual Studio			+
3.	Підтримка мов програмування	Підтримує будь-яку мову програмування і легко розширюється	Підтримує C# та C++	Підтримує C# та C++			+
4.	Підтримка мов опису архітектури	+	-	-			+
5.	Валідації архітектури	Надає вже готові метрики, дозволяє їх розширяти і надає можливість писати тести на якість архітектури	Валідація неможлива	Валідація присутня. Також присутня можливість писати власні правила валідації		+	

Продовження таблиці 5.2

№ п/п	Техніко-економічні характеристики ідеї	Проекти конкурентів			W слабка сторона	N нейтральна сторона	S сильна сторона
		Мій проект	Visual Studio	NDepend			
6.	Можливість валідації в CI/CD	+	-	+			+
7.	Адаптивність	Висока	Середня	Середня			+
8.	Фінансування	Власні кошти	Комерційне	Комерційне	+		

Відносно конкурентів система майже в усьому виграє через універсальність, високий потенціал та високу адаптивність. Основний недолік - це не налагоджене і недостатнє на даному етапі фінансування. Отже, якщо вирішити цю проблему - дана система може зайняти ключову позицію на ринку.

Основними недоліками конкурентів є занадто вузька направленість. Тільки користувачі Visual Studio та інженери, що використовують C# чи C++ можуть використовувати наявні плагіни. Отже більшість ринку наразі взагалі не має подібних інструментів.

5.2. Технологічний аудит ідеї проекту

Щоб визначити технологічну здійсненність ідеї проекту, необхідно проаналізувати такі складові з таблиці 5.3:

- технологія виготовлення товару відповідно до ідеї проекту
- існування всіх потрібних технологій, або необхідність їх дороблення
- доступність технологій авторам проекту

Таблиця 5.3 - Технологічна здійсненність ідеї проекту системи виявлення архітектурних помилок програмного забезпечення

№ п/п	Ідея проекту	Технології і реалізації	Наявність технологій	Доступність технологій
1.	Створення можливості аналізувати та тестувати архітектуру проекту; Додати можливість гарантувати відповідність архітектури проекту до архітектури описаної архітектором	Технологія 1 (парсер вихідного коду проекту TypeScript)	Наявні	Доступні
2.		Технологія 2 (відображення графа у браузері)	Наявні	Доступні
3.		Технологія 3 (реалізація протоколів між підпрограмами)	Наявні	Доступні
Обрана технологія реалізації ідеї стартап-проекту розробки системи виявлення архітектурних помилок програмного забезпечення: є можливою.				

5.3. Аналіз ринкових можливостей запуску стартап-проекту

Таблиця 5.4 описує ринкові можливості продукту, що використовуються при впровадженні проекту на ринок, дозволяють розрахувати можливі ринкові ризики та спланувати можливі напрямки розвитку проекту враховуючи ситуацію на ринковому середовищі, пропозиції конкурентів та потреби потенційних клієнтів.

За даними проведеного дослідження, даний проект є дуже привабливим з точки зору входження на ринок. Сам по собі ринок є зростаючим, при цьому дуже велика його частина є вільною. В той же час наявність великих гравців говорить про необхідність даного класу продуктів.

Таблиця 5.4 - Попередня характеристика потенційного ринку стартап-проекту

№ п/п	Показники стану ринку	Характеристика
1.	Кількість головних гравців, од	2
2.	Загальний обсяг продажів, грн./ум.од.	1200 грн/ліцензія за даними [24]
3.	Динаміка ринку	Зростає
4.	Наявність обмежень для входу	Немає
5.	Специфічні умови до стандартизації та специфікації	Немає
6.	Середня норма рентабельності в галузі	24%

Під час проведення даного дослідження було визначено та сформульовано основних користувачів та цільову аудиторію. Вся отримана інформація у розрізі їх характеристик та особливостей наводиться у таблиці 5.5.

Таблиця 5.5 - Характеристики потенційних користувачів стартап проекту

№ п/п	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці різних потенційних груп клієнтів	Вимоги споживачів до товару
1.	Покращення та контроль якості вихідного коду проекту	Бізнес, що займається розробкою програмного забезпечення	Бажання бути кращими на ринку, потреба зменшення кількості помилок в програмному забезпеченні,	Надійність, легкість у використанні, підтримка у впровадженні та використанні системи

Продовження таблиці 5.5

№ п/п	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці різних потенційних груп клієнтів	Вимоги споживачів до товару
1.	Покращення та контроль якості вихідного коду проекту	Бізнес, що займається розробкою програмного забезпечення	необхідність зменшити терміни на розробку та витрати пов’язані з якістю архітектури	Надійність, легкість у використанні, підтримка у впровадженні та використанні системи
2.	Аналіз та вивчення архітектури проекту	Архітектори програмного забезпечення, студенти, розробники	Архітекторам та розробникам потрібно якомога швидше зібрати інформацію про проект, щоб приймати рішення; студенти бажають покращити своє розуміння та отримати зворотній зв’язок своїй роботі	Функціональність, зручність в аналізі

Основною цільовою аудиторією є розробники, що працюють на комерційних проектах, так як саме вони вирішують чи треба даний продукт бізнесу чи ні.

Архітектори знаходяться на другому місці, через те, що вони не мають безпосередньої потреби у даному продукті. Вони можуть його використати, щоб швидко зібрати необхідну інформацію для прийняття архітектурних рішень і для валідації своїх власних рішень (завжди приємно отримати зворотній зв’язок заснований на математичних розрахунках).

Самі по собі бізнеси не являються цільовою аудиторією т.я. менеджери не мають відношення до технічних питань, але вони можуть вимагати покращення характеристик продукту від розробників, що може наштовхнути їх на думку про купівлю ліцензії на даний продукт.

Основні фактори загроз розташовані у таблиці 5.6 у порядку сили впливу від сильнішого до слабшого. Вони спроможні зашкодити продукту зайняти своє місце на ринку.

Таблиця 5.6 - Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1.	Недостатній рекламний бюджет	Даний напрямок хоч і не являється новим, але для мов де він вже реалізований, там сильна конкуренція, в іншому випадку - даний напрямок ще не відомий, і потрібен значний бюджет, щоб поширити інформацію про продукт	Розвивати продукт і продати його компанії, яка має необхідний бюджет на його поширення; партнерство з компанією, яка зацікавлена у поширенні продукту та його розвитку на долевій основі; пошук інвестора; краудфандинг.

Продовження таблиці 5.6

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
2.	Нестабільність політичної ситуації в світі	У випадку світової війни або інших світових проблем можливе скорочення бюджету на розробку продуктів, а даний інструмент являється додатковим	Знижки і акції на використання продукції
3.	Велика кількість помилок в продукті	Велика кількість помилок можуть сформувати погану репутацію компанії та продукту і завадити продукту зайняти місце на ринку	Перегляд процесу розробки та введення додаткових мір задля підвищення надійності продукту
4.	Конкуренти-копії на ринку	Коли з’являються конкуренти копії - цінність та унікальність пропозиції значно падає	Покращення позиціонування компанії на ринку, краща взаємодія з користувачами і надання кращої підтримки ніж конкуренти. Швидка реакція на необхідність розширення функціоналу системи
5.	Відсутність достатньої кількості продажів	Щоб розвивати та рекламувати систему - необхідні продажі, які будуть наповнювати бюджет компанії. Якщо ж по якійсь причині з часом їх	Змінити рекламну та маркетингову стратегію, робити акції та знижки, використовувати для реклами зв’язки з іншими брендами

Продовження таблиці 5.6

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
5.	Відсутність достатньої кількості продажів	кількість не буде зростати - компанія може обанкротитися	
6.	Проблеми у цільової аудиторії з розумінням функціональності системи	Якщо реклама працює досить добре, багато хто знає про цей продукт, але ніхто особливо не розуміє навіщо він потрібен саме йому, в чому його переваги	Покращення документації продукту та FAQ секції, створити можливість отримати підтримку людини, яка знає переваги та недоліки даного продукту

Отож найбільшим ризиком для цього стартап-проекту може стати відсутність достатньої кількості бюджету на рекламу, що зумовлено вільними ринками на всіх мовах крім C# та C++.

Разом з високими ризиками також існують не гірші можливості. Таблиця 5.7 приводить їх аналіз, а найпотужнішою можливістю є можливість розширення ринку поза мови програмування C# та C++.

Таблиця 5.7 - Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1.	Відсутність подібної продукції для всіх мож опису архітектури та мов програмування окрім C++ та C#	Можливість значно розширити ринок і зайняти перше місце серед подібних продуктів	Націленість на незайняті сектори ринку (пріоритети)

Продовження таблиці 5.7

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
2.	Зростання вимог до програмного забезпечення	Можливість запропонувати інструмент, що розширить кількість аспектів покритих автоматичними тестами	Акцентування на поліпшенні якості продукції за рахунок додаткових автотестів для архітектури проекту та сутностей
3.	Зростання вимог до покращення та здешевлення процесу розробки програмної продукції	Можливість покращити позиціонування компанії за рахунок цих факторів впливу	Акцентувати увагу на поліпшенні за здешевленні процесу розробки програмної продукції за рахунок використання даного стартап-проекту у життєвому циклі своєї продукції

Аналіз пропозиції на ринку приведено в таблиці 5.8.

Таблиця 5.8 - Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства
Чистий тип конкуренції	На ринку наразі існує сильна перевага у компанії ZEN PROGRAM HLD, продукт якої називається Ndepend на ринку мов C# та C++	Не варто починати освоєння ринку з цих мов програмування (C#, C++). Треба обрати декілька популярних мов без подібного інструментарію
Локальність за рівнем конкуренції боротьби	Локальність - без обмежень	Необхідно почати локалізацію продукту з англійської мови і її поступово розширяти

Продовження таблиці 5.8

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства
Міжгалузева за галузевою ознакою	Конкуренція відбувається на рівні архітектури програмного забезпечення, тобто не прямими конкурентами є архітектори ПЗ та розробники	Ніякий. Часткова автоматизація і тести - це основна ідея даного програмного забезпечення
Нецінова за характером конкурентних переваг	По ціні даний продукт до виходу стабільної версії буде безкоштовним, або навіть open source	Не просити гроші до стабілізації функціоналу
Марочна за інтенсивністю	Visual Studio та Ndepend є досить відомоми продуктами і дуже важко буде конкурувати на рівні марки та бренду	Необхідно створити свій бренд та просувати його на рівні з продуктом

Таблиця 5.9 деталізовано описує аналіз умов конкуренції за М. Портером.

Таблиця 5.9 - Аналіз конкуренції в галузі за М. Портером

кладові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Користувачі	Товарозамінники
	Ndepend та Visual Studio	Лінтери та фреймворки для написання	Відсутні	Архітектори та інженери програмного забезпечення	Мануальний аналіз та перевірка

Продовження таблиці 5.9

кладові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Користувачі	Товарозамінники
		тестів			архітекторами та інженерами програмного забезпечення
Висновки:	Інтенсивність буде дуже висока на кластерах ринку, які вже зайняті. Тож просто можна зайняти позицію на ще не зайнятих сегментах опісля легко розширившись.	Дані види програм не є конкурентами в даний момент часу, але в них є потенціал до розширення в обраному напрямку.	Дана система не залежить від інших систем та не потребує забезпечення будь-якими ресурсами для коректної роботи.	Треба автоматизувати все що є можливим. Саме це надає можливості рости та розвиватися, бо ми делегуємо частину роботи комп'ютеру, який, до того ж, може виконувати її набагато точніше, коли розмова йде про архітектурні питання.	Наразі подібні функції або ігноруються і не виконуються на проектах, або виконуються вручну. Це є серйозним недоліком і при появі подібної системи - конкуренція дуже слабка, якщо, звісно, в даній системі є необхідність.

Проаналізувавши конкуренцію, стає очевидно, що дана продукція потрібна даному ринку. Єдина серйозна конкуренція складається для мов

C++ та C# в рамках Visual Studio, але поза ними, простір абсолютно вільний.

Розглянемо тепер фактори конкурентоспроможності для розробки в таблиці 5.10.

Таблиця 5.10 - Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтквання
1.	Конкуренція поза мовами C# та C++	Все дуже просто, ринок не зайнятий і аналоги у вигляді мануальної роботи архітекторів та розробників ПЗ значно програють для реалізації поставлених цілей.
2.	Простота використання	Дана системи має дуже простий консольний інтерфейс і не потребує значних знань для початку роботи з нею.
3.	Рівень кастомізації	Дана система дозволяє дуже легко додати підтримку нової мови програмування або опису архітектури, нові метрики та нові тести.
4.	Ціна	Якщо ціна й буде, то буде нижчою за ціну конкурентів, до того ж система буде частково безкоштовною і лише деякий функціонал коштувати гроші.
5.	Рівень підтримки користувачів	Це дозволить швидко вирости і створити не просто продукт, а бренд.
6.	Маркетинговий та рекламний бюджет	Маркетинговий та рекламний бюджет безпосередньо відображає швидкість поширення інформації про продукт та рівень ознайомлення потенційних клієнтів з продуктом.

В таблиці 5.11 можна побачити результат аналізу на сильні та слабкі сторони по представленим факторам конкурентоспроможності данного стартап-проекту.

Таблиця 5.11 - Порівняльний аналіз сильних та слабких сторін "Системи виявлення архітектурних помилок"

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з "Системою виявлення архітектурних помилок"						
			-3	-2	-1	0	+1	+2	+3
2	Простота використання	17						+	
3	Рівень кастомізації	20							+
4	Ціна	20							+
5	Рівень підтримки користувачів	17						+	
6	Маркетинговий та рекламний бюджет	12			+				
1	Конкуренція поза мовами C# та C++	20							+

Основна слабка сторона проекту - це маркетинговий та рекламний бюджет, який грає дуже важливу роль для становлення проекту.

На основі таблиці 5.11 складено SWOT-аналіз, який представлено в таблиці 5.12.

Таблиця 5.12 - SWOT-аналіз проекту

Сильні сторони: - Простота використання системи - Рівень кастомізації - Ціна - Конкуренція поза мовами C# та C++	Слабкі сторони: - Маркетинговий та рекламний бюджет - Рівень підтримки користувачів
Можливості: - Зайняти ключову позицію на ринку через зайняття вільних ринкових секторів - Можливість покращити якість продуктів за рахунок обширної автоматизації в розрізі перевірки якості архітектури - Можливість здешевити та покращити якість процесу розробки ПЗ	Загрози: - Недостатній рекламний бюджет - Нестабільність політичної ситуації в світі - Конкуренти-копії на ринку - Відсутність достатньої кількості продажів - Проблеми з розумінням функціональності системи цільовою аудиторією

За результатами SWOT-аналізу було підготовлено декілька стратегій ринкового впровадження стартап-проекту. Ці стратегії представлені в таблиці 5.13.

Таблиця 5.13 - Альтернативи ринкового впровадження стартап проекту

№ п/п	Альтернатива	Ймовірність отримання ресурсів	Строки реалізації
1.	Реалізація підтримки найпопулярнішої мови програмування і діаграм класів UML з валідацією	Висока	9 місяців

	програми по UML підграфу з маркетинговим планом		

Продовження таблиці 5.13

№ п/п	Альтернатива	Ймовірність отримання ресурсів	Строки реалізації
2.	Реалізація підтримки тільки найпопулярнішої мови програмування з розширеними можливостями візуального аналізу з маркетинговим планом	Середня	8 місяців
3.	Реалізація підтримки тільки найпопулярнішої мови програмування з маркетинговим планом	Низька	6 місяців

Зважаючи на результати аналізу в таблиці 5.13, доцільно обрати стратегію з реалізацією підтримки найпопулярнішої мови програмування та діаграми класів UML для можливості порівняння їх на входження графу діаграми класів у граф вихідного коду проекту. Вихід на ринок без такої унікальної можливості не надає значних конкурентних переваг в порівнянні з продуктами, які вже існують на ринку. З іншого боку, це потребує більшого часу на реалізацію.

5.4. Висновки

Описана ідея стартап-проекту “Система виявлення архітектурних помилок”, як системи для аналізу, виміру якісних характеристик вихідного коду проекту та автоматизованому тестуванню архітектури проекту на відповідність до встановлених якісних вимог. Також було проведено

технологічний аудит ідеї проекту та проаналізовано ринкові можливості запуску стартап-проекту.

Згідно з проведеним аналізом, дана система є реалізуємою, конкуренція наявна, але більшість ринку абсолютно вільна, що і дасть змогу легко зайти з даним продуктом.

Показник рентабельності є дуже високим і витрачені кошти мають дуже швидко окупитися.

ВИСНОВКИ

В даній роботі було проведено дослідження архітектури та якісних характеристик програмного забезпечення, створено нову метрику на основі вже існуючих, розроблено систему виявлення архітектурних помилок програмного забезпечення в середовищі TypeScript та розроблено стартап-проект для виходу на ринок. Розроблене програмне забезпечення реалізує створену метрику і використовує її для аналізу вихідного коду.

Було виконано таку завдання:

- досліджена предметна область і визначено процес архітектуризації, з чого він складається та якими поняттями оперує;
- досліджені якісні характеристики програмного забезпечення та визначено характеристики, що можна вирахувати з вихідного програмного коду;
- виконано аналітичний огляд існуючих факторів, які впливають на визначені вимірювані якісні характеристики;
- виконано огляд існуючих метрик, що описують вплив приведених факторів;
- визначено існуючі фактори, які впливають на всі вимірювані характеристики і на їх основі підібрані метрики, які характеризують кожну з характеристик;
- з підібраних метрик було створено нову метрику, що базується на них і описує модульність вихідного програмного коду;
- розроблено MVP, що реалізує створену метрику і надає звіт по якості сутностей в коді та частково реалізує описану архітектуру;
- розроблено стартап-проект для даної системи.

Дана система може бути застосована для покращення продуктивності та ефективності розробки програмних систем, безпосередньо впливаючи на якість вихідного коду програмного забезпечення. Використовувати її

можуть як компанії, так і незалежні розробники. Безпосередніми користувачами являються розробники та архітектори ПЗ.

ЛІТЕРАТУРА

1. ISO/IEC/ IEEE 42010, Systems and software engineering — Architecture description
2. Zachman, J.A., "A Framework for Information Systems Architecture", IBM Systems Journal, 26(3), 1987
3. Ministry of Defense Architecture Framework (MODAF), http://www.modaf.org.uk/
4. The Open Group, TOGAF 9.2 Specification, April 2018, https://www.opengroup.org/togaf
5. Kruchten, P.B., "The '4+1' View Model of Architecture", IEEE Software, 12(6), 45-50, 1995
6. Hofmeister, C., R.Nord, and D.Soni, Applied Software Architecture, Reading, MA: Addison-Wesley, 1999
7. ISO/IEC 10746-1, Information technology - Open Distributed Processing - Reference model: Overview
8. ISO/IEC 10746-2, Information technology - Open distributed processing - Reference model: Foundations
9. ISO/IEC 10746-3, Information technology - Open distributed processing - Reference model: Architecture
10. ISO 15704, Industrial automation systems - Requirements for enterprise-reference architectures and methodologies
11. ISO/IEC/IEEE 42010:2011, Systems and software engineering — Architecture description
12. Luckham, D.C., J.J.Kenney, L.M.Augustin, J.Vera, D.Bryan and W.Mann, "Specification and analysis of system architecture using RAPIDE", IEEE Transactions on Software Engineering, 21(4), 336-355, April 1995
13. Wright ЯОА website, http://www.cs.cmu.edu/~able/wright/

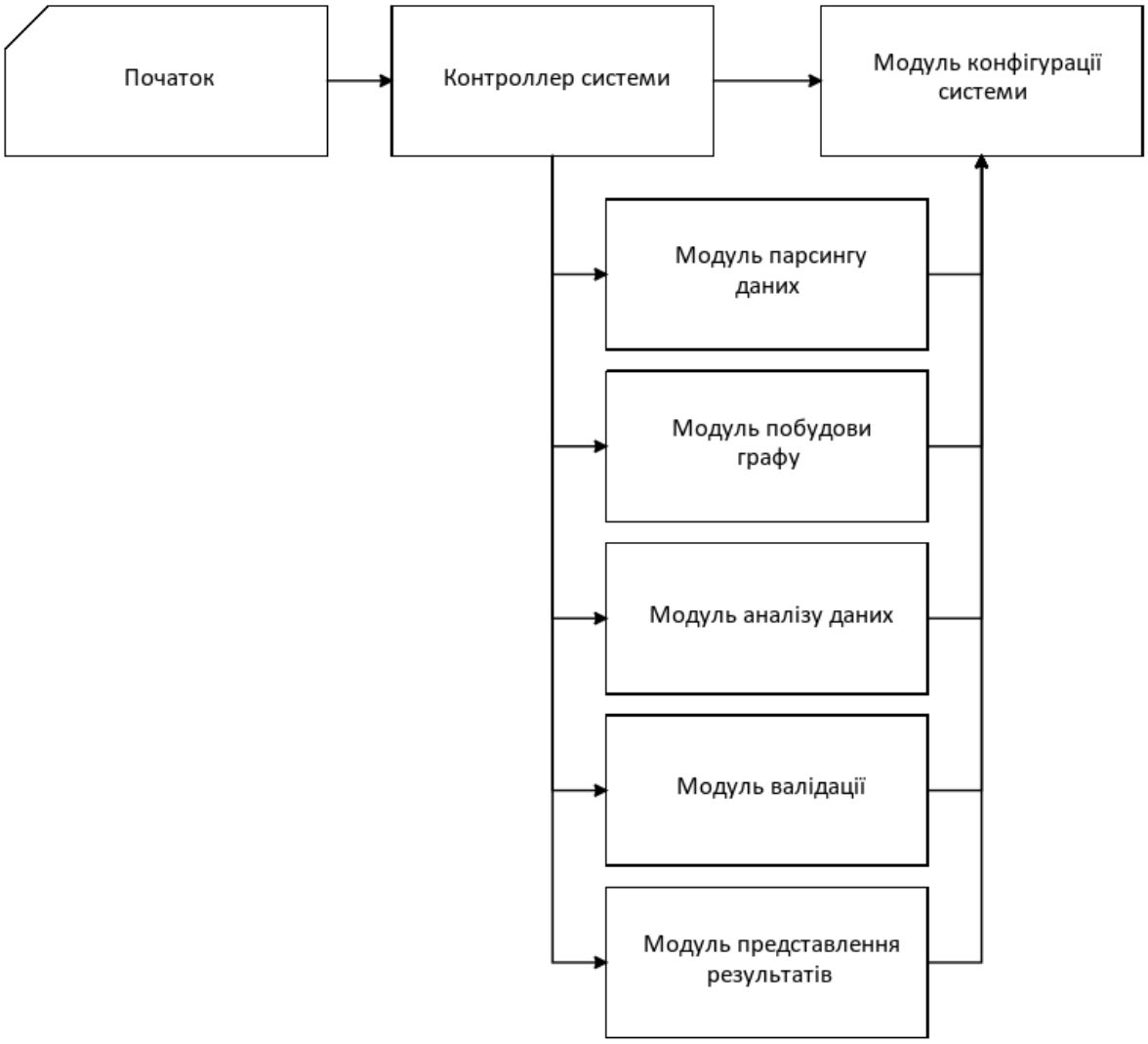
14. OMG formal/2008-11-01, Systems Modeling Language, version 1.1, November 2008
15. The Open Group, ArchiMate 1.0 Specification, February 2009, http://www.archimate.org/
16. Martin Fowler, “UML Distilled, A Brief Guide to the Standard Object Modeling Language”, 38, 2004
17. ISO/IEC 25010, Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models
18. Eeles P. and P.Cripps, The Process of Software Architecting. Addison Wesley, 2010
19. Hitz M., Montazeri B. Measuring coupling and cohesion in object-oriented systems. – na, 1995. – С. 25-27.
20. Lakshminarayana A., Newman T. S. Principal component analysis of lack of cohesion in methods (lcom) metrics //Technical Report TRUAH-CS-1999-01. – 1999.
21. Briand L. C., Morasca S., Basili V. R. Property-based software engineering measurement //IEEE transactions on software Engineering. – 1996. – Т. 22. – №. 1. – С. 68-86.
22. Mijač M., Stapić Z. Reusability metrics of software components: survey //Proceedings of the 26th Central European Conference on Information and Intelligent Systems. – 2015. – С. 221-231.
23. Washizaki H., Yamamoto H., Fukazawa Y. A metrics suite for measuring reusability of software components //Proceedings. 5th International Workshop on enterprise networking and computing in healthcare industry (IEEE Cat. No. 03EX717). – IEEE, 2004. – С. 211-223.
24. Payment costs for Ndepend system, https://www.ndepend.com/purchase
25. L. C. Briand, J. W. Daly, and J. Wust. A uni- ed framework for cohesion measurement in object oriented systems. In Proceedings of the Fourth

International Symposium on Software Metrics, pages 43{53, Albuquerque, NM USA, Nov. 1997. IEEE Computer Society.

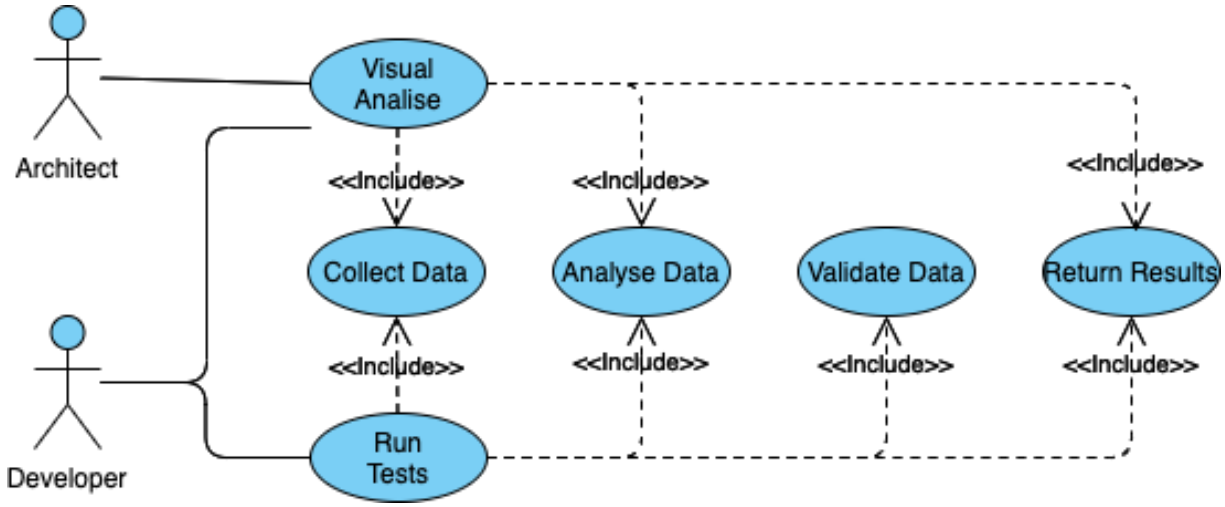
ДОДАТОК А

Графічний матеріал

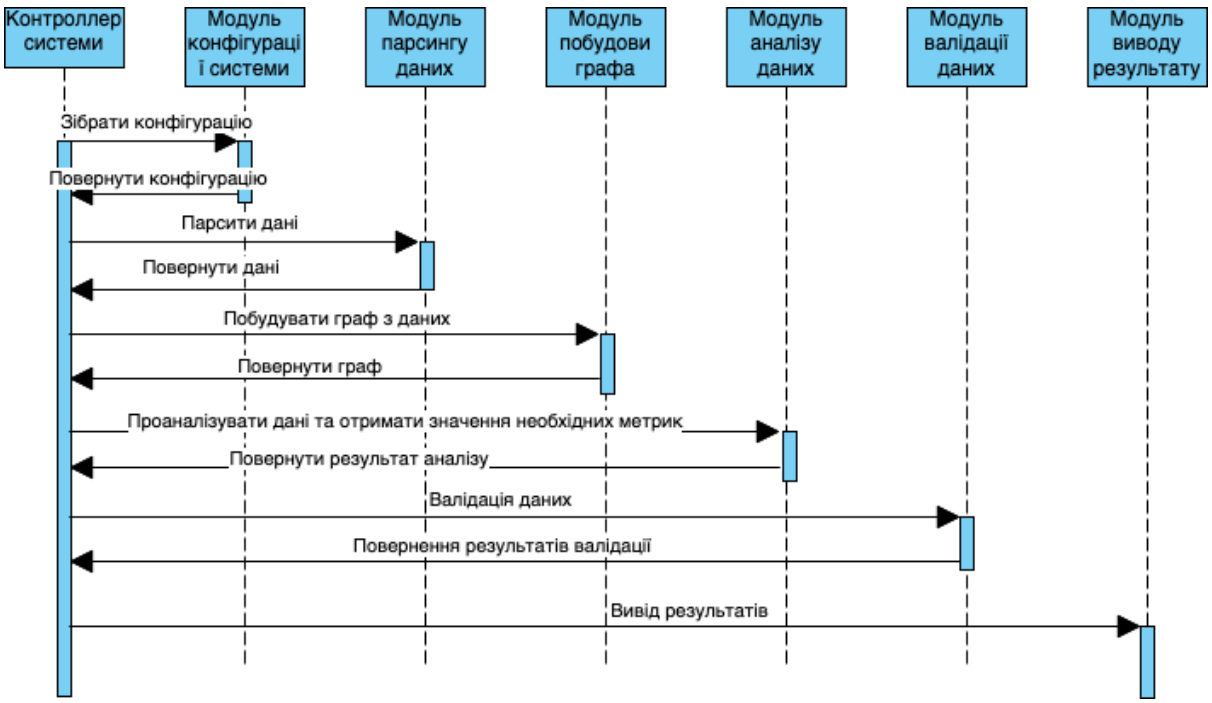
Плакат 1 Функціонально-логічна структура програмного забезпечення



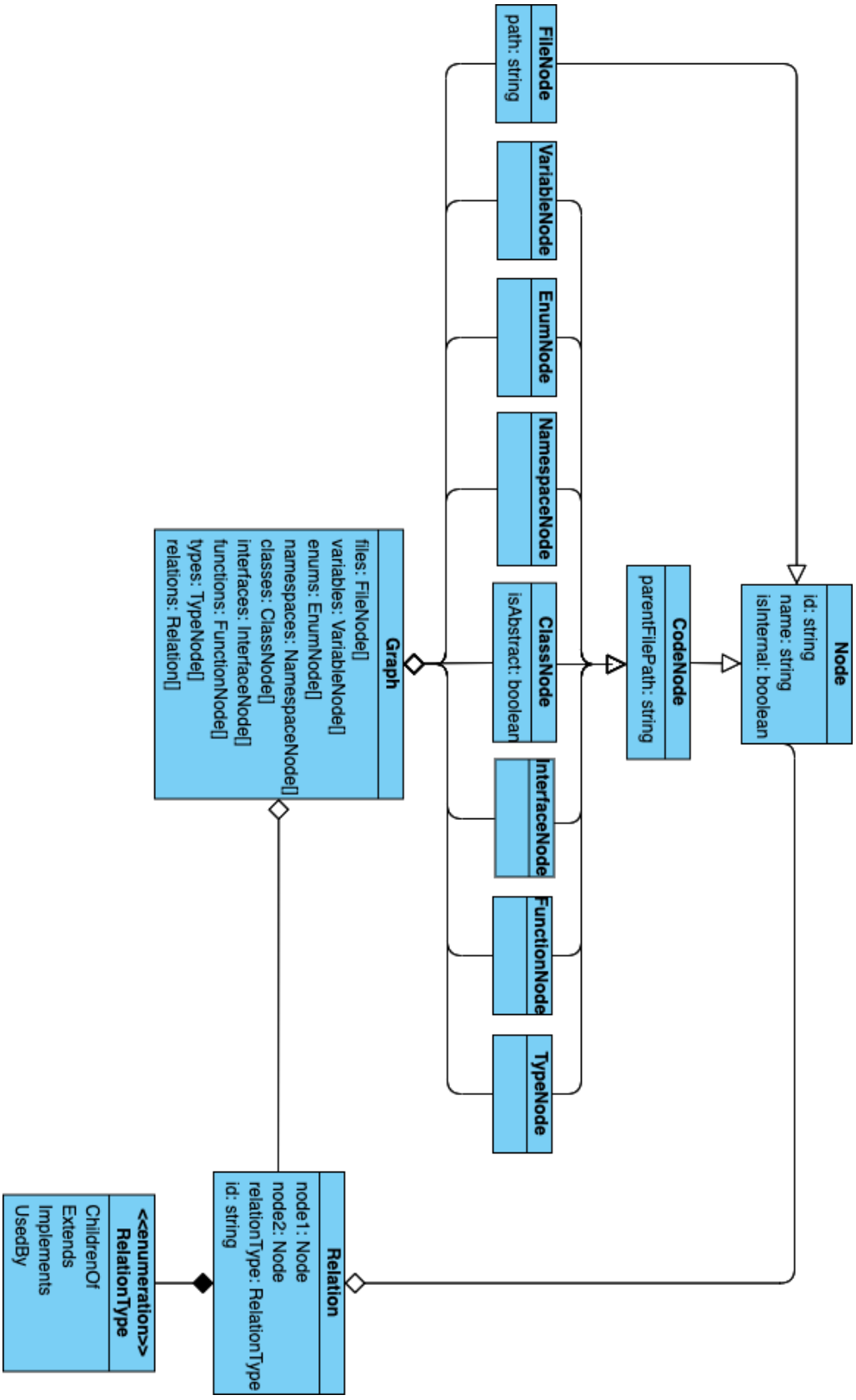
Плакат 2 UML-діаграма варіантів використання



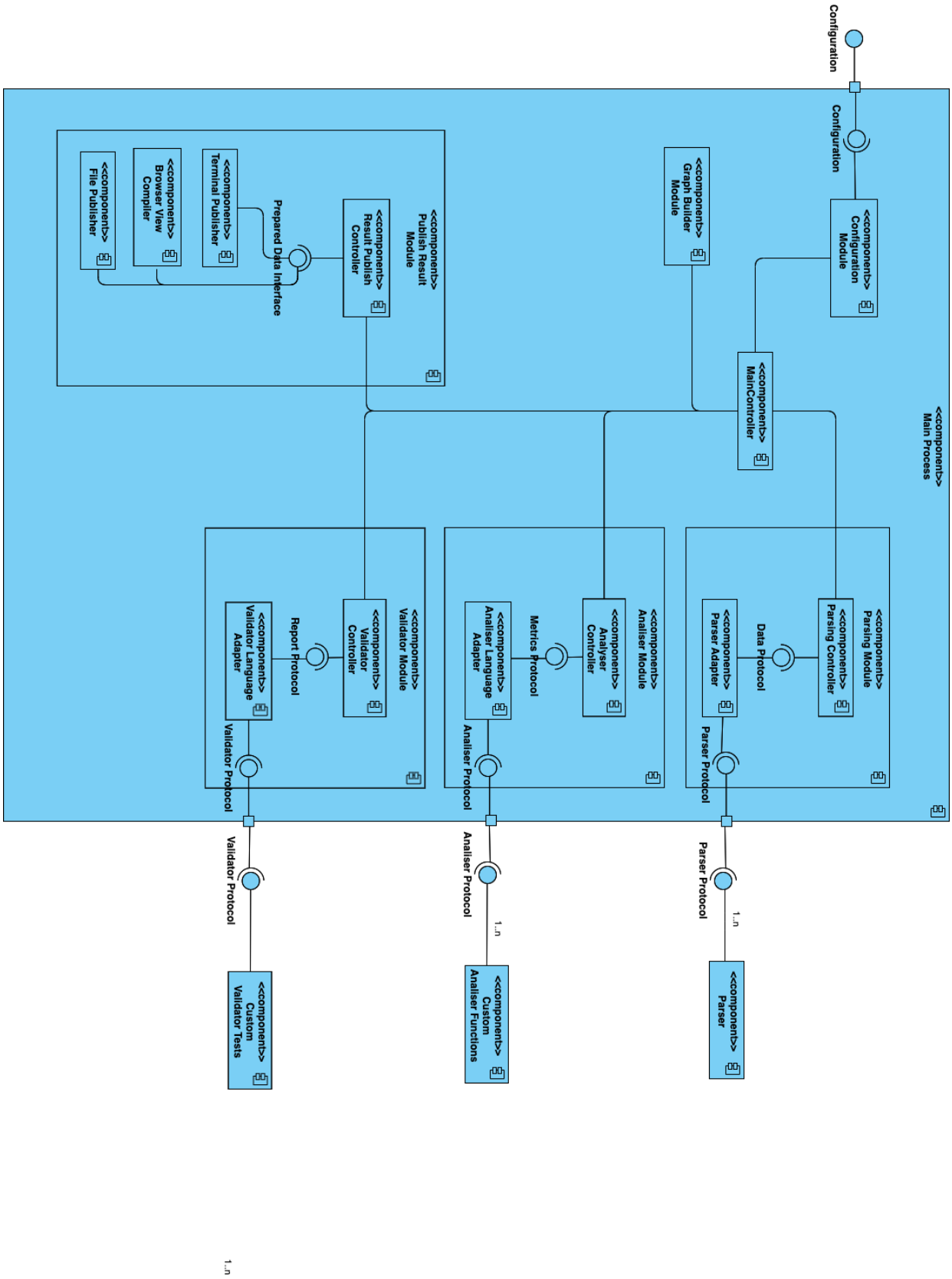
Плакат 3 Діаграма послідовності



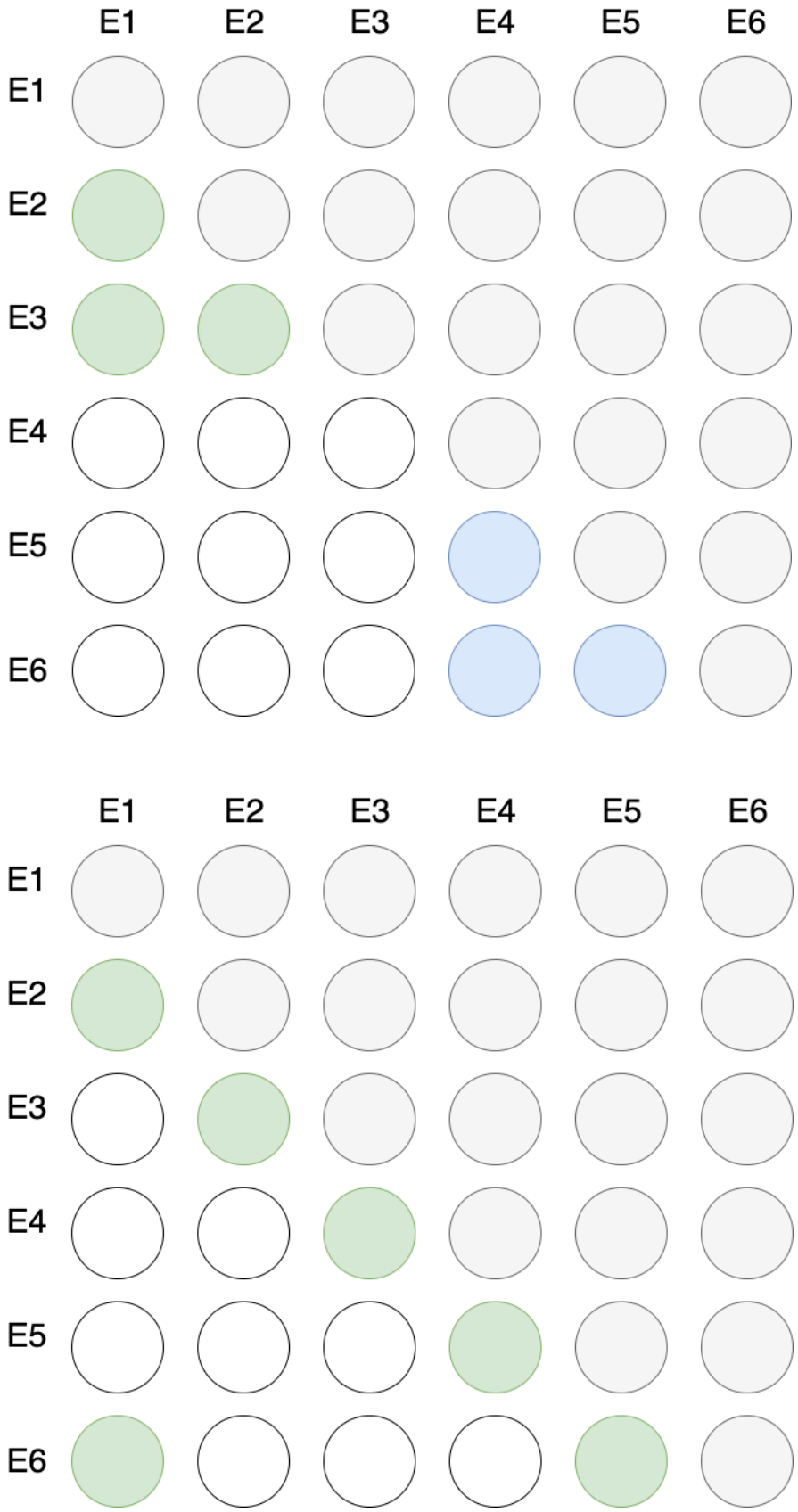
Плакат 4 UML-діаграма класів



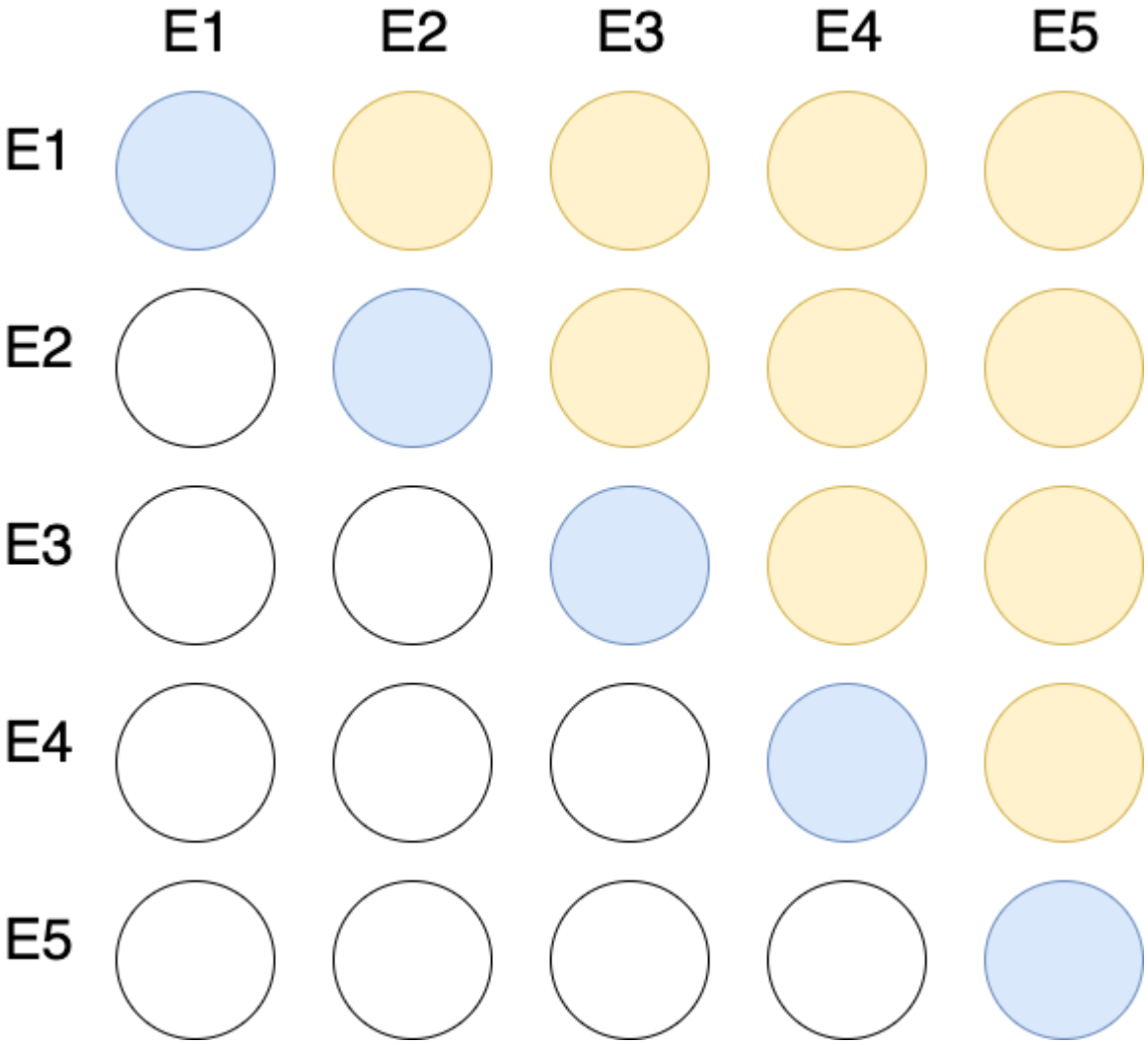
Плакат 5 UML-діаграма компонентів



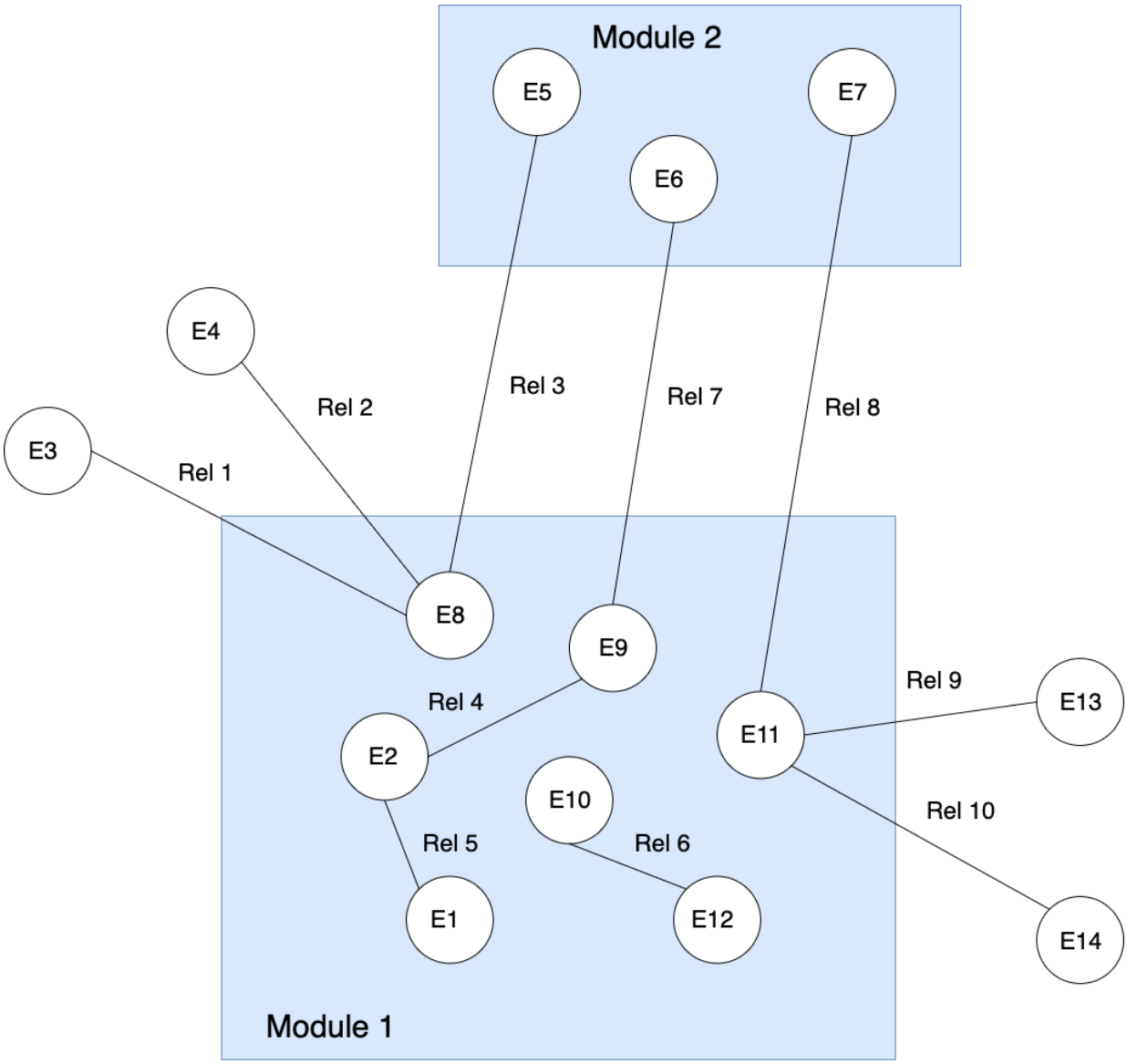
Плакат 6 Випадки, що надають підставу для покращення розрахунку сполученості модуля



Плакат 7 Можливі зв’язки між сутностями модуля для розрахунку сполученості



Плакат 8 Граф деякої програми


Module 2
E5
E7
E6
E4
E3
Rel 1
Rel 2
Rel 3
Rel 7
Rel 8
E8
E9
Rel 4
E2
E10
Rel 5
Rel 6
E1
E12
E11
Rel 9
E13
Rel 10
E14
Module 1