**МАГІСТЕРСЬКА РОБОТА**

МР. ІПм - 16.00.00.000 ПЗ

**Група ІПм-23-2**

**Мацюк Іван**

**2024**

**Івано-Франківський національний технічний університет нафти і газу**

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

**Мацюк Іван Володимирович**

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

# МАГІСТЕРСЬКА РОБОТА

## Методи інтерактивної візуалізації програмного забезпечення на основі JavaScript

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Мацюк І.В.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Вовк Роман Богданович, к.т.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

**Допущено до захисту**

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

**Івано-Франківський національний технічний університет нафти і газу**

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

**ЗАТВЕРДЖУЮ:**
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2024 р.

# ЗАВДАННЯ

## НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

**Мацюку Івану Володимировичу**

(прізвище, ім’я, по-батькові)

**1. Тема магістерської роботи “ Методи інтерактивної візуалізації програмного забезпечення на основі JavaScript ”**

керівник проекту (роботи) Вовк Роман Богданович, к.т.н., доцент

затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781/7

**2. Строк подання студентом проекту (роботи)** 15 грудня 2024 р.

**3. Вихідні дані до проекту (роботи)** Теоретичні концепції та формальні моделі побудови та функціонування інформаційних та програмних технологій візуалізації ПЗ

**4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)**

1. Дослідження та аналіз предметної області візуалізації програмного забезпечення
2. Дослідження методів, алгоритмів та інструментів візуалізації програмного забезпечення
3. Аналіз викликів функцій і використання бібліотек для побудови візуалізацій
4. Представлення методології інтерактивної візуалізації структури програмного забезпечення

**5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)**

1. Концептуальна архітектура системи (рис. 1.1)
2. Програмна архітектура бібліотеки AVSXLib (рис. 1.2)
3. Візуалізація за допомогою UML (рис. 1.3)
4. Огляд платформи візуалізації графів CodeGraph (рис. 2.1)
5. Графічний інтерфейс Hunter (рис. 2.2)

**6. Консультанти розділів проекту (роботи)**

| Розділ | Консультант | Підпис, дата |
|---|---|---|
| Перевірка на плагіат | доц., к.т.н. Вовк Р.Б. | |
| | | |
| | | |

**7. Дата видачі завдання** 04 вересня 2024 р.

**Керівник** _____
(підпис)

**Завдання прийняв до виконання**_____
(підпис)

# КАЛЕНДАРНИЙ ПЛАН

| № п/п | Назви етапів магістерської роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|---|
| 1 | Підбір і вивчення літератури по темі магістерської роботи | 15.09.2024 | виконано |
| 2 | Аналіз концепцій та алгоритмів предметної області | 29.09.2024 | виконано |
| 3 | Дослідження та аналіз предметної області візуалізації програмного забезпечення | 15.10.2024 | виконано |
| 4 | Дослідження методів, алгоритмів та інструментів візуалізації програмного забезпечення | 08.11.2024 | виконано |
| 5 | Аналіз викликів функцій і використання бібліотек для побудови візуалізацій | 20.11.2024 | виконано |
| 6 | Представлення методології інтерактивної візуалізації структури програмного забезпечення | 01.12.2024 | виконано |
| 7 | Затвердження пояснювальної записки роботи завідувачем кафедри | 15.12.2024 | виконано |

**Студент – магістр** _____
(підпис)

**Керівник роботи** _____
(підпис)

# АНОТАЦІЯ

**Магістерська робота:** 82 с., 22 рис., 1 табл., 55 джерел.

**Тема:** Методи інтерактивної візуалізації програмного забезпечення на основі JavaScript

**Об'єкт дослідження:** структура програмного забезпечення, зокрема її архітектурні елементи, функціональні залежності та бібліотеки.

**Мета роботи:** дослідження та розробка інтерактивного інструменту для візуалізації структури програмного забезпечення на основі JavaScript, що дозволить розробникам більш ефективно аналізувати архітектуру програмних систем.

**Предмет дослідження:** методи та алгоритми інтерактивної візуалізації структури програмного забезпечення на основі JavaScript.

**Результати дослідження**

В роботі запропоновано підхід до автоматизованого виявлення залежностей між файлами та бібліотеками програмного проекту за допомогою аналізу абстрактного синтаксичного дерева (AST) і застосовано комбінований підхід для візуалізації архітектури проекту, що включає кілька підграфів, які відображають різні аспекти залежностей і структури програмного забезпечення.

**Висновок**

Розроблена методологія може бути використана для автоматизованого аналізу архітектури проектів на JavaScript, що значно спрощує процес виявлення проблем, оптимізації коду та покращення організації проектів.

**ІНТЕРАКТИВНА ВІЗУАЛІЗАЦІЯ, СТРУКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, JAVASCRIPT, АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, АЛГОРИТМИ ВІЗУАЛІЗАЦІЇ, AST, БІБЛІОТЕКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ЗАЛЕЖНОСТІ**

# ABSTRACT

**Master Thesis:** 82 pp., 22 fig., 1 tab., 55 sources.

**Thesis Subject:** Methods of interactive visualization of JavaScript-based software

**Research object:** software structure, including its architectural elements, functional relationships, and libraries.

**The purpose of the work:** research and development of an interactive tool for visualizing the structure of software based on JavaScript, which allows developers to more effectively analyze the architecture of software systems.

**Subject of research:** methods and algorithms of interactive visualization of the software structure based on JavaScript.

**Research results**

Your work proposes to approach the automated dependency between files and libraries of a software project using Abstract Syntax Tree (AST) analysis and applies a combined approach to visualizing architectural designs that includes several subgraphs that represent different aspects of the dependency structure and software.

**Conclusion**

The developed methodology can be used for automated analysis of architectural projects in JavaScript, which significantly improves the process of problem identification, code optimization, and improvement of project organization.



**INTERACTIVE VISUALIZATION, SOFTWARE STRUCTURE, JAVASCRIPT, SOFTWARE ARCHITECTURE, VISUALIZATION ALGORITHMS, AST, SOFTWARE LIBRARIES, DEPENDENCIES**

**ЗМІСТ**

## ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

AST - Abstract Syntax Tree

LUNA - Library Usage in Node.js Analyzer

PWA - Progressive Web Application

XHR - XMLHttpRequest

CLI - Command Line Interface

IDE - Integrated Development Environment

JSONP - JSON with Padding

D3 - Data-Driven Documents

OSS - Open Source Software

NPM - Node Package Manager

# ВСТУП

**Актуальність теми.**

З розвитком програмних систем їх архітектура стає дедалі складнішою, що ускладнює аналіз, підтримку та оптимізацію проектів. Візуалізація структури програмного забезпечення є важливим інструментом, який полегшує розуміння залежностей між компонентами, бібліотеками та іншими елементами. Зокрема, інтерактивні візуалізації, що базуються на JavaScript, дозволяють розробникам гнучко взаємодіяти з архітектурою системи та швидко знаходити проблемні місця. Дослідження сучасних методів і алгоритмів візуалізації є актуальним для поліпшення процесу розробки та супроводу програмного забезпечення.

В сучасних умовах, коли розробка програмного забезпечення стає дедалі більш складною і масштабною, виникає нагальна потреба у засобах, що сприяють розумінню архітектури системи та її взаємозалежностей. Програмні проекти часто включають велику кількість зовнішніх бібліотек, мають складну ієрархію модулів і компонентів, що створює труднощі у підтримці та оптимізації коду. Особливо це стосується проектів, що розробляються на JavaScript, який є однією з найбільш поширених мов програмування для веб-додатків, але також є особливо вразливою до проблем масштабованості через динамічний характер мови і численні залежності від сторонніх бібліотек.

Інтерактивна візуалізація структури програмного забезпечення є одним з ефективних інструментів для вирішення цих проблем. Вона дозволяє розробникам в реальному часі бачити взаємозв'язки між компонентами, виявляти слабкі місця у структурі, аналізувати залежності між модулями та бібліотеками. Сучасні інструменти візуалізації, хоча й існують, часто не забезпечують необхідної гнучкості та деталізації для великих і складних проектів. Дослідження методів та алгоритмів, що дозволяють оптимізувати

процес візуалізації та покращити ефективність аналізу архітектури програмного забезпечення, є актуальним для індустрії програмної розробки.

Крім того, актуальність дослідження підкріплюється необхідністю інтеграції інструментів візуалізації у сучасні розробницькі процеси, зокрема у контексті DevOps і CI/CD, де автоматизовані засоби аналізу структури програмного забезпечення можуть стати ключовим елементом для підтримки якості та стабільності проектів. Розробка інструментів, що дозволяють гнучко взаємодіяти з архітектурою проекту та аналізувати бібліотеки, є важливою для оптимізації розробки, спрощення супроводу коду та покращення роботи над великими проектами.

**Мета дослідження** – дослідження та розробка інтерактивного інструменту для візуалізації структури програмного забезпечення на основі JavaScript, що дозволить розробникам більш ефективно аналізувати архітектуру програмних систем.

**Об'єкт дослідження** – структура програмного забезпечення, зокрема її архітектурні елементи, функціональні залежності та бібліотеки.

**Предмет дослідження** - методи та алгоритми інтерактивної візуалізації структури програмного забезпечення на основі JavaScript.

Відповідно до мети роботи було сформовано наступні **задачі:**

- Провести огляд існуючих інструментів для візуалізації структури програмного забезпечення, таких як CodeGraph, Hunter та Eunice.
- Дослідити алгоритми побудови візуалізацій, зокрема Breadthfirst, Cola, Cose-Bilkent та алгоритми з набору Eclipse Layout Kernel.
- Розробити структуру рішення для інтерактивної візуалізації архітектури програмного забезпечення.
- Дослідити інструмент для візуалізації, що дозволяє аналізувати структуру проекту JavaScript, виявляти залежності між файлами, бібліотеками та їх використанням.
- Перевірити ефективність інструменту шляхом тестування на реальних проектах.

**Методи дослідження**

Для досягнення мети дослідження використано методи аналізу існуючих інструментів та алгоритмів візуалізації, методи абстрактного синтаксичного аналізу (AST), алгоритми для побудови графів та візуалізації залежностей між компонентами програмного забезпечення.

**Наукова новизна отриманих результатів.**

Запропоновано підхід до автоматизованого виявлення залежностей між файлами та бібліотеками програмного проекту за допомогою аналізу абстрактного синтаксичного дерева (AST) і застосовано комбінований підхід для візуалізації архітектури проекту, що включає кілька підграфів, які відображають різні аспекти залежностей і структури програмного забезпечення.

**Практичне значення магістерської роботи**

Розроблена методологія може бути використана для автоматизованого аналізу архітектури проектів на JavaScript, що значно спрощує процес виявлення проблем, оптимізації коду та покращення організації проектів.

**Структура магістерської роботи.** Робота складається зі вступу, трьох розділів та висновків. Загальний обсяг роботи становить 811 сторінку, і містить 23 рисунки, 1 таблицю, перелік використаних джерел із 55 позицій.

# РОЗДІЛ 1. ДОСЛІДЖЕННЯ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ВІЗУАЛІЗАЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 1.1. Опис області дослідження

У процесі розробки програмного забезпечення значна частина кінцевого коду є насправді творінням сторонніх розробників, зазвичай через так звані бібліотеки з відкритим кодом. Бібліотеки стали звичайним явищем у розробці програмного забезпечення. Вони чудові, оскільки можуть абстрагувати та спрощувати код і проблеми програмування. Однак складно підтримувати огляд цих бібліотек в архітектурі програмного забезпечення, особливо коли вони використовуються з високою частотою.

В даній роботі досліджується інструмент візуалізації, який може показати абстрактний огляд того, як бібліотеки взаємодіють із програмним забезпеченням. Ми представляємо LUNA, або Library Usage in Node.js Analyzer. Це інструмент розробки програмного забезпечення, орієнтований на бібліотеки для проектів node.js. Як випливає з назви, LUNA є JavaScript-додатком для екосистеми Node.js і NPM. Ця екосистема відома своєю кількістю мікропакетів і складною мережею залежностей.

Метою LUNA є допомогти розробникам зрозуміти, як бібліотеки використовуються в їхніх проектах. Ми досліджуємо, як LUNA може відновити архітектуру програмного забезпечення та використання бібліотеки шляхом аналізу абстрактного синтаксичного дерева (AST) вихідного коду та створити візуалізацію, яка може відобразити всю цю інформацію в інтерактивному графіку. Нарешті, корисність LUNA досліджується через інтерв'ю з користувачами. Результати показують, що інструмент може відновити архітектуру програми та використання бібліотеки, а візуалізація може відобразити цю інформацію в зрозумілому вигляді. Інтерв'ю також показали, що LUNA може справді допомогти розробникам краще зрозуміти

архітектуру своєї програми та використання бібліотеки, як це робиться в цій роботі.

Термін програмний пакет має декілька значень [1]. В цій роботі він використовується для позначення бібліотеки або фреймворку, який є набором коду, написаного для повторного використання іншими програмними додатками. Отже, основна функція бібліотеки програмного забезпечення полягає в тому, щоб запропонувати рішення конкретної проблеми або набору проблем, що може бути виконано, пропонуючи набір методів, структур даних або класів. Його можна розглядати як чорний ящик. Бібліотеки можуть істотно допомогти в розробці програмного забезпечення. Як правило, їх можна імпортувати та використовувати будь-де у вихідному коді програми. Однак де це має найбільший сенс робити в архітектурі програмного забезпечення? А що програмне забезпечення робить на практиці? Як можна керувати та відстежувати всі бібліотеки, які використовуються? Насправді ми не маємо однозначної відповіді на ці проблеми, але це було б цікаво знати, щоб приймати правильні рішення щодо дизайну програмного забезпечення, пов’язаного з бібліотеками.

Таким чином, мета цієї роботи полягає в тому, щоб зрозуміти та проілюструвати, як програмні пакети використовуються в архітектурі програмного забезпечення. Це досягається шляхом дослідження того, як програмні бібліотеки використовуються в програмному забезпеченні. Для цього використовується інструмент під назвою LUNA, який докладно описано в даній роботі.

Зараз ми спочатку обговоримо актуальність проблеми та визначимо зацікавлених сторін. Після цього ми розглядаємо питання дослідження та пояснюємо, чому вони важливі для проблеми дослідження.

Як згадувалося раніше, мета цієї роботи — надати більше інформації про те, як бібліотеки інтегруються в кодову базу програмного забезпечення. Це актуально як для дослідників, так і для практиків. Дослідники можуть використовувати інструмент, розроблений для цієї дисертації, для інших

досліджень або для розширення цього дослідження. Практикуючі спеціалісти, як-от розробники та супроводжувачі програмного забезпечення, можуть використовувати результати цього дослідження для прийняття більш глибоких проектних рішень під час включення бібліотек. Проблема дослідження стосується кількох зацікавлених сторін. Давайте розглянемо, які варіанти використання пакетів програмного забезпечення можуть мати ці зацікавлені сторони та що вони можуть отримати від цього дослідження.

1. Інженери програмного забезпечення

1.1. Вони можуть оцінити вплив бібліотеки на їхній проект (з точки зору частоти використання)

1.2. Вони можуть зменшити складність свого проекту, використовуючи бібліотеки, які забезпечують спрощення або абстракцію

1.3. Вони можуть продемонструвати інформацію про використані бібліотеки іншим зацікавленим сторонам

1.4. Вони можуть дізнатися, що потрібно змінити в коді проекту при заміні бібліотек

2. Розробники бібліотеки

2.1. Вони можуть бачити, які частини їх бібліотеки використовуються в проекті

2.2. Вони можуть бачити, які частини проекту використовують їхню бібліотеку

2.3. Вони можуть осягнути дерево залежностей своєї бібліотеки

3. Експерти з безпеки

3.1. Вони можуть дізнатися, де в проекті використовується вразлива або скомпрометована бібліотека

3.2. Вони вказують, якою мірою в проекті використовується вразлива або скомпрометована бібліотека

У цьому розділі ми розглянемо дослідницькі питання, пов’язані з проблемою, описаною вище. Є одне питання, центральне для проблеми, яке полягає в наступному:

1. Як полегшити розуміння використання бібліотеки в архітектурі програмного забезпечення?

Це основне питання дослідження. Однак, оскільки це досить широке питання дослідження, пропонується кілька підзапитань, які допоможуть відповісти на основне питання дослідження:

1.1. Як відновити архітектуру програмного забезпечення ?

Важливо зрозуміти, як відновити архітектуру програмного забезпечення, тому що нам потрібно знати, як виглядає архітектура, щоб відповісти на основне дослідницьке запитання. Це також має відношення до інших питань дослідження. Перш ніж ми зможемо відновити архітектуру програмного забезпечення, ми повинні спочатку проаналізувати, що її визначає. Це робиться в даній роботі, де ми обговорюємо значення архітектури програмного забезпечення.

1.2. Як визначити бібліотеки та їх використання ?

Перш ніж ми зможемо відповісти на головне питання дослідження, нам потрібно знати, як виявляти бібліотеки та їх використання. Архітектура програмного забезпечення відіграє в цьому роль, оскільки нам потрібно знати, де шукати бібліотеки. Отже, це питання пов’язане з попереднім підпитанням

1.3. Як візуалізувати цю інформацію корисним способом ?

Нарешті, ми повинні зрозуміти, як візуалізувати зібрану інформацію, щоб полегшити розуміння використання бібліотек в архітектурі програмного забезпечення. Ця інформація випливає з питань 1.1 і 1.2. Зауважте, що в цьому дослідницькому питанні в основному йдеться про дві речі: як візуалізувати архітектуру програмного забезпечення за допомогою бібліотек і як це зробити з урахуванням корисності. Візуалізація не має сенсу, якщо вона не може бути корисною для користувача.

Відповіді на ці підзапитання разом повинні дати відповідь на основне дослідницьке питання цієї роботи. Дослідницькі питання сформульовані

таким чином, щоб на них можна було відповісти за допомогою інструменту, пропонованого в цій роботі.

## 1.2. Поняття архітектури, бібліотек та візуалізації програмного забезпечення

### *1.2.1. Архітектура програмного забезпечення*

Основоположні структури програмної системи, а також дисципліна побудови таких структур і систем називаються архітектурою програмного забезпечення. Кожна структура складається з компонентів програмного забезпечення, зв’язків між ними та атрибутів обох елементів і зв’язків [2]. Ці компоненти можуть бути модулями, класами або функціями, а зв’язки можуть бути залежностями, асоціаціями або успадкуванням. Атрибути можуть бути властивостями компонентів, такими як їхня назва, або властивостями зв’язків, такими як їхній тип. Архітектура програмного забезпечення — це модель системи, яка використовується для опису структури та поведінки системи. Архітектура також використовується для керівництва розробкою системи та для передачі інформації про структуру та поведінку системи зацікавленим сторонам [3].

Архітектура програмного забезпечення – це фундаментальна структура, яка лежить в основі будь-якого програмного продукту. Це високорівневий опис того, як система побудована, які компоненти вона містить та як ці компоненти взаємодіють між собою.

Важливість архітектури полягає в наступному:

- Спрощує розуміння: Архітектура надає загальну картину системи, що полегшує розуміння її функціональності та взаємозв'язків між різними частинами.

- Забезпечує основу для розробки: Архітектура слугує своєрідним "планом будівництва", керуючи розробкою та прийняттям рішень на всіх етапах життєвого циклу програмного продукту.

- Покращує масштабованість: Добре продумана архітектура дозволяє легко розширювати та змінювати систему в майбутньому.

- Сприяє повторному використанню коду: Архітектура заохочує модульність і повторне використання компонентів, що зменшує витрати на розробку.

- Поліпшує якість: Завдяки чітко визначеній архітектурі легше знаходити та виправляти помилки, а також забезпечувати стабільність системи.

Основні компоненти архітектури:

- Компоненти: Це окремі модулі або служби, які виконують певні функції.

- Коннектори: Забезпечують взаємодію між компонентами.

- Контейнери: Групують компоненти та коннектори.

- Інтерфейси: Визначають, як компоненти взаємодіють один з одним.

Хороша архітектура програмного забезпечення необхідна з багатьох причин: вона полегшує написання, розуміння та модифікацію коду; це може зробити код більш придатним для обслуговування та без помилок; і, мабуть, найважливіше, хороша архітектура програмного забезпечення дозволяє створювати масштабовані програми. Іншими словами, за наявності добре спроектованої архітектури програма може легко справлятися зі збільшеними навантаженнями, не виходячи з ладу та не стаючи нестабільною. І навпаки, погано спроектована архітектура, швидше за все, призведе до безладу в коді, який дуже важко (якщо взагалі неможливо) масштабувати.

Концептуальна архітектура описує систему з точки зору її основних елементів дизайну та взаємозв'язків між ними. Це дуже високорівнева структура системи, що використовує елементи дизайну та взаємозв'язки, специфічні для домену. Концептуальні архітектури є незалежними від рішень реалізації та підкреслюють протоколи взаємодії між елементами дизайну.

У більшості досліджених нами систем концептуальні архітектури були неявними та вбудованими в документацію інших структур. Хоча

концептуальна архітектура системи не задокументована явно, архітектор системи намалював неформальні діаграми, що зображують основні компоненти системи, під час опису системи нам. Ми отримали модель компонент-коннектор її концептуальної архітектури на основі цих дискусій з архітектором системи та через широкі дослідження документації та коду. Ця модель неофіційно описана на рисунку 1.1; заокруглені прямокутники представляють функціональні компоненти, а прямокутні прямокутники представляють коннектори (тобто інтерфейси та протоколи) між компонентами.

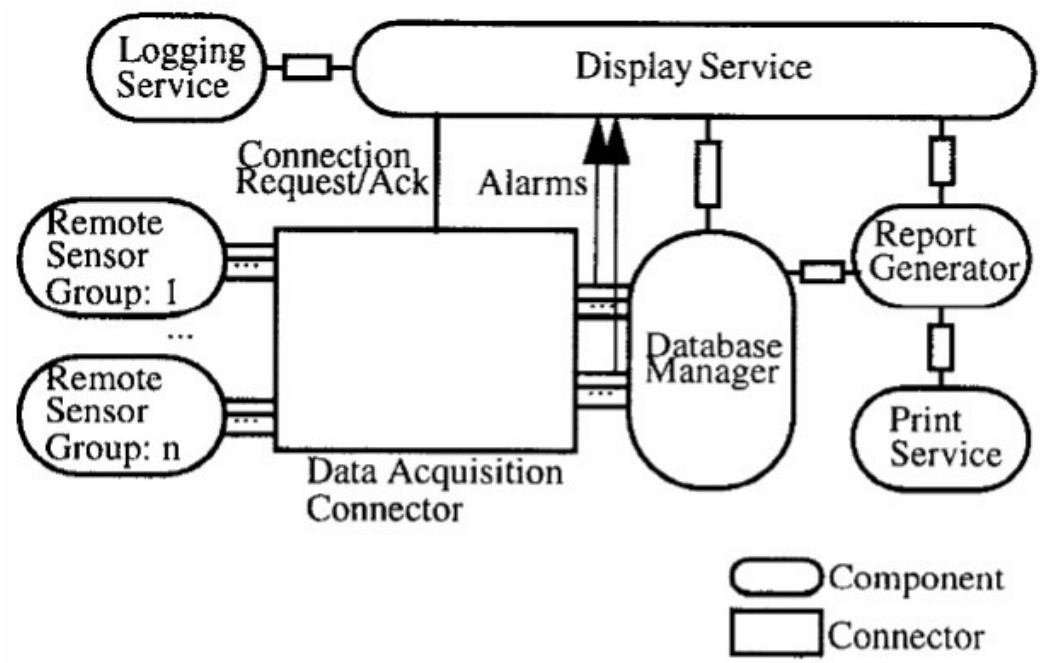


Рис. 1.1. Концептуальна архітектура системи

Якщо ми хочемо проаналізувати архітектуру програмного забезпечення, нам потрібно її відновити. Основна ідея полягає в тому, щоб проаналізувати деяку структурну інформацію програмного забезпечення, наприклад, його код або залежності, і спробувати зробити висновок з цього аналізу, як виглядає високорівнева організація програмного забезпечення. Існує багато методів відновлення архітектури програмного забезпечення. У

розділі 1.3 , будуть обговорені дослідження порівняння методів відновлення архітектури програмного забезпечення [4, 5].

### *1.2.2. Бібліотеки програмного забезпечення*

Бібліотеку програмного забезпечення можна визначити як набір підпрограм і функцій, які використовуються для виконання типових завдань. Це набір коду, який можна повторно використовувати в різних програмах. Ми називаємо надані функціональні можливості бібліотеки інтерфейсом прикладного програмування (API), за допомогою якого бібліотека взаємодіє з рештою програми. Коли ви пишете програму, ви можете використовувати ці бібліотеки без необхідності писати власний код з нуля. Це економить ваш час і зусилля, оскільки вам не потрібно винаходити колесо кожного разу, коли вам потрібна певна функція. Бібліотеки програмного забезпечення важливі, оскільки вони є будівельними блоками для програмістів. Вони дозволяють нам розбити складні проблеми на більш дрібні частини, а потім знову об’єднати їх у робочі рішення. Без бібліотек нам довелося б починати з нуля кожного разу, коли ми хотіли б створити щось нове.

Основні особливості бібліотек:

- Модульність: Бібліотеки зазвичай структуровані на окремі модулі, кожен з яких відповідає за певну функціональність. Це дозволяє легко інтегрувати їх у різні проекти та використовувати лише ті функції, які необхідні.

- Повторне використання: Головна перевага бібліотек – можливість повторного використання коду. Це економить час і ресурси розробників.

- Спеціалізація: Бібліотеки часто створюються для вирішення конкретних завдань або для роботи з певними технологіями. Наприклад, існують бібліотеки для роботи з базами даних, графікою, мережевими протоколами тощо.

- Абстрагування: Бібліотеки приховують складні деталі реалізації, надаючи розробникам простий інтерфейс для взаємодії.

- Розширення: Багато бібліотек мають модульну структуру, що дозволяє легко розширювати їх функціональність за допомогою плагінів або додаткових модулів.

- Спільнота: Бібліотеки з відкритим кодом часто мають активні спільноти розробників, які допомагають у вирішенні проблем, вносять свій вклад у розвиток і підтримують документацію.

Типи бібліотек:

- Стандартні бібліотеки: Вбудовані в мову програмування і надають базові функції для роботи з вводом-виводом, математичними операціями, обробкою рядків тощо.

- Сторонні бібліотеки: Розроблені незалежними розробниками або компаніями і доступні для використання в сторонніх проектах. Вони можуть бути як з відкритим, так і з закритим кодом.

- Фреймворки: Більш масштабні за бібліотеки, вони надають готову структуру для розробки додатків певного типу (наприклад, веб-фреймворки).

Розглянемо бібліотеку програмного забезпечення (рис. 1.2).

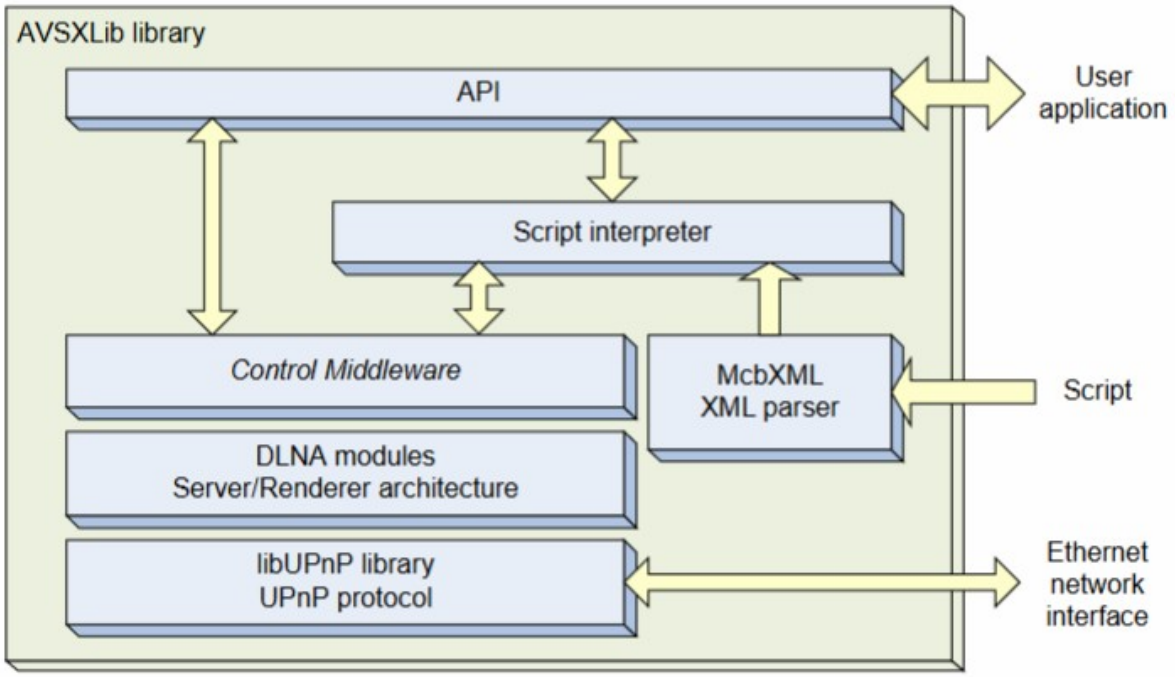


Рис. 1.2. Програмна архітектура бібліотеки AVSXLib

Бібліотека AVSXLib складається з таких базових програмних блоків: (1) бібліотека UPnP (libupnp), (2) модулі підтримки DLNA, (3) проміжне програмне забезпечення управління, (4) інтерпретатор XML-скриптів та (5) API бібліотеки (інтерфейс програмування додатків). Підтримка як Windows, так і Linux досягається за допомогою використання примітивів POSIX (Portable Operating System Interface) та бібліотеки pthread, що мінімізувало необхідність подвійного кодування. Архітектура програмного забезпечення AVSXLib наведена на рисунку 1.2.

### *1.2.3. Візуалізація*

Візуалізація, або, точніше, візуалізація даних, — це представлення даних у графічному чи графічному форматі. Це спосіб передачі інформації шляхом кодування чисел, символів і зображень у візуальні об’єкти. Мета візуалізації даних — чітко й ефективно передавати інформацію за допомогою статистичної графіки, графіків, інформаційної графіки та інших візуальних форматів. Візуалізація даних є потужним інструментом для передачі інформації. Це дозволяє нам побачити закономірності та тенденції в даних, які інакше було б важко виявити. Це також дозволяє нам робити прогнози щодо майбутнього на основі минулих даних. Візуалізація даних є важливою частиною науки про дані, оскільки вона дозволяє нам розуміти дані таким чином, що неможливо за допомогою лише чисел і тексту. Існує велика різноманітність методів візуалізації даних, таких як точкові діаграми, гістограми та кругові діаграми. У цій дипломній роботі ми зосередимося на техніках візуалізації графів [6]. Візуалізація графів — це спосіб представлення структурної інформації у вигляді діаграм абстрактних графів і мереж. Це спосіб подання інформації у спосіб, який легко зрозуміти та інтерпретувати.

UML є незамінним інструментом для розробників програмного забезпечення, що дозволяє їм створювати більш якісні та надійні системи. За

допомогою UML можна візуалізувати складні концепції, покращити комунікацію між членами команди та спростити процес розробки.

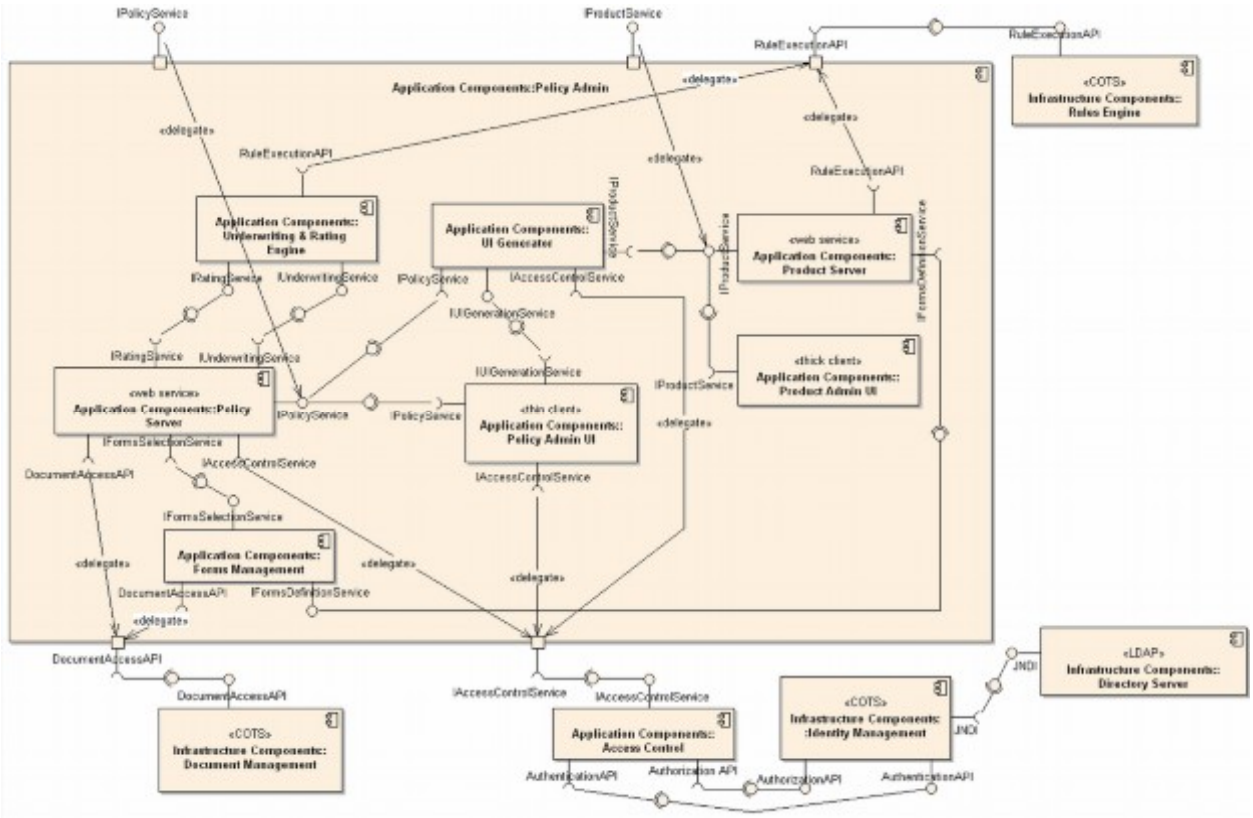


Рис. 1.3. Візуалізація за допомогою UML

## 1.3. Огляд літератури по темі дослідження

Цей розділ присвячено літературі, яка має відношення до цієї магістерської роботи, охоплюючи такі теми: відновлення архітектури програмного забезпечення, бібліотеки програмного забезпечення та візуалізація. Ця література використовується для надання довідкової інформації для методології, використаної в цій дипломній роботі. Кожна робота буде коротко представлена та описана, а також подібності та відмінності між ними та те, як вони пов’язані з проблемою дослідження.

Для початку в [7], досліджується вплив мікропакетів на екосистему програмного забезпечення з відкритим вихідним кодом (OSS), зокрема на екосистему JavaScript NPM, яка також є актуальною темою, яка

обговорюється в роботі. Як було показано пізніше, інструмент, розроблений для цієї дипломної роботи, може чітко продемонструвати цю проблему для екосистеми NPM. У цій роботі мікропакети визначаються як мінімізовані бібліотеки, і вони можуть мати довгі ланцюжки залежностей, що може стати проблемою для критичних систем.

Знову ж таки, менеджер пакетів програмного забезпечення NPM досліджується в роботі [8]. Зокрема, вони досліджують значення показників популярності, які використовуються для бібліотек програмного забезпечення, і порівнюють існуючі.

Робота [9] є прикладом магістерської роботи, яка аналізує NPM і JavaScript у масштабі. Серед іншого, вони обговорюють причини високого відсотка клонів і пропонують кілька ідей щодо того, які аналізи можна буде зробити в майбутньому на основі зібраних даних.

В [3] йдеться про практичну архітектуру програмного забезпечення. У книзі детально описується архітектура програмного забезпечення, включно з тим, чому вона важлива, а також як розробляти, створювати екземпляри, аналізувати, розвивати та керувати нею дисциплінованим і ефективним способом.

Одним із способів аналізу архітектури програмного забезпечення є виявлення загальних шаблонів проектування. Саме це було зроблено в [10] як етап створення документації для проектів JavaScript. В якості своєї методології тут використовували відбитки пальців на побудованому абстрактному синтаксичному дереві (AST) кодової бази. Дослідження [11] також фокусується на створенні документації, зокрема для веб-додатків Node.js. Вони поєднують автоматизовану генерацію діаграми класів з деякими налаштуваннями вручну, а також аналіз вихідного коду та неофіційні інтерв'ю, як метод зворотного проектування програмного забезпечення для виведення двох документів: специфікації вимог до програмного забезпечення та документу з розробки програмного забезпечення. Інструмент, розроблений для цієї роботи, також аналізує AST

кодової бази, але він не фокусується на створенні документації. Він також забезпечує мінімальну абстракцію класу.

Інша стаття, пов'язана з відновленням архітектури - це дослідження [12], де представляють автоматизований підхід на основі машинного навчання для класифікації рольового стереотипу класів у Java. Крім того, вони порівнюють свій підхід з іншим існуючим підходом класифікації на основі правил.

Існує кілька методів відновлення архітектури програмного забезпечення, В [4] пропонується дослідження, щоб порівняти їх усіх. Вони використовують засоби тестування програмного забезпечення, щоб порівняти алгоритми відновлення в кількох проектах. Подібним чином стаття [5] порівнює ефективність кількох методів відновлення архітектури програмного забезпечення за допомогою набору з 8 проектів з відкритим вихідним кодом та їхніх істин щодо архітектури.

Бібліотеки програмного забезпечення з часом оновлюються, що може призвести до критичних змін у доступному API. У [13] провели широкомасштабне емпіричне дослідження, досліджуючи зміни в роботі API та їхній вплив на клієнтські програми. Їхнє дослідження виявило, що:

1) 14,78% змін API порушують сумісність із попередніми версіями,
2) частота несправних змін збільшується з часом,
3) це впливає на 2,54% їхніх клієнтів
4) системи з більшою частотою критичних змін є більшими, популярнішими та активнішими.

Подібно до попередньої роботи, робота [14] також досліджував вплив несправних змін в API, але на відміну від роботи [13], як уже згадувалося раніше, вони зосереджуються саме на бібліотеках і враховують їх еволюцію та семантичне версії. Вони розробили інструмент під назвою Maracas з метою виявлення критичних змін між 2 версіями та того, на які частини клієнтського коду вони впливають.

У статті [15] обговорюють наслідки повторного використання програмного забезпечення у виробництві. Пізніше вони провели опитування на цю тему та опублікували результати. Це дає нам гарне розуміння походження програмних бібліотек і того, як вони з'явилися.

Існує багато методів представлення компонентів програмного забезпечення для повторного використання, і в роботі [16] ці методи оглядаються та класифікуються. Крім того, вони обговорюють системи, в яких вони використовувалися, і пропонують структуру представлення повторного використання програмного забезпечення, яка пов'язує ці методи. Пізніше вони провели емпіричне дослідження цього питання у [17].

У дослідженні [18] обговорюють труднощі побудови графів викликів для JavaScript. Графіки викликів фіксують зв'язок між функціями програмного забезпечення. Вони вирішують цю проблему, представляючи масштабований аналіз потоку на основі поля для побудови графіків викликів, який добре працює на практиці.

Робота [19] також представляє нове рішення для побудови графа викликів у JavaScript, спеціально для екосистеми node.js. Вони підходять до проблеми з точки зору безпеки. Крім того, вони враховують модульну структуру програм Node.js. Вони стверджують, що вони більш точні та ефективні порівняно з програмним забезпеченням конкурентів.

Дослідження [6] дає загальний вступ до методів візуалізації графів. Він дає загальний огляд кількох алгоритмів компонування та методів взаємодії.

## Висновки до розділу

У результаті дослідження та аналізу предметної області візуалізації програмного забезпечення було визначено ключові аспекти, що складають основу сучасних підходів до розробки та використання засобів візуалізації у сфері інформаційних технологій.

По-перше, опис області дослідження дав можливість окреслити межі вивчення, акцентуючи увагу на важливості візуалізації для спрощення процесів розуміння та аналізу архітектури програмних систем. Було показано, що сучасна архітектура програмного забезпечення є багаторівневою і вимагає застосування різноманітних технік для її ефективного аналізу.

По-друге, детальне вивчення поняття архітектури програмного забезпечення дало змогу ідентифікувати основні компоненти і підходи, що лежать в основі структури сучасних програмних рішень. Було визначено, що архітектура включає не тільки технічні аспекти, але й такі важливі фактори, як гнучкість, масштабованість та стійкість до змін.

Аналіз бібліотек програмного забезпечення показав їх важливу роль у прискоренні розробки та спрощенні повторного використання коду. Різноманітність бібліотек і їх значення в контексті візуалізації було також розглянуто з метою демонстрації можливостей для оптимізації процесів розробки програмних продуктів.

Нарешті, детальний огляд літератури по темі дослідження виявив існуючі тенденції та підходи до візуалізації програмного забезпечення. Це дозволило визначити найбільш перспективні напрямки розвитку цієї сфери, а також окреслити виклики, з якими стикаються дослідники та практики.

Таким чином, даний розділ заклав теоретичну основу для подальшого дослідження методів та інструментів візуалізації програмного забезпечення, визначивши ключові поняття, підходи та сучасні тренди у цій галузі.

# РОЗДІЛ 2. ДОСЛІДЖЕННЯ МЕТОДІВ, АЛГОРИТМІВ ТА ІНСТРУМЕНТІВ ВІЗУАЛІЗАЦІЇ СТРУКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 2.1. Аналіз існуючих інструментів візуалізації структури програмного забезпечення

Розглянемо два інструменти CodeGraph і HUNTER – це проекти, які спеціалізуються на процесах візуалізації. Дослідимо які методи для досягнення подібної мети вони використовують. Крім того, аналізовані проекти підтримують JavaScript.

### *2.1.1. Дослідження інструменту CodeGraph для візуалізації графу залежностей проектів JavaScript*

В роботі [33] було розроблено CodeGraph, який є «веб-додатком, який візуалізує граф залежностей проектів JavaScript.

CodeGraph - це потужний веб-додаток, спеціально розроблений для візуалізації графів залежностей у проектах JavaScript. Цей інструмент дозволяє розробникам отримати наочне уявлення про структуру їхніх проектів, ідентифікувати циклічні залежності, виявити потенційні проблеми з модульністю та оптимізувати архітектуру коду.

Основні функціональні можливості CodeGraph:

- Візуалізація графів залежностей. CodeGraph генерує інтерактивні графічні представлення залежностей між модулями JavaScript. Це дозволяє легко побачити, як модулі пов'язані між собою, і які модулі залежать від інших.

- Підтримка різних форматів. Додаток підтримує різні формати опису проектів, такі як package.json (Node.js), bower.json і інші. Це дозволяє аналізувати залежності проектів, створених на різних технологіях.

- Інтерактивність. Графи, створені за допомогою CodeGraph, є інтерактивними. Користувачі можуть збільшувати, зменшувати, переміщувати вузли та ребра, а також фільтрувати граф за різними критеріями.

- Виявлення циклічних залежностей. CodeGraph автоматично виявляє циклічні залежності між модулями, що може свідчити про проблеми в архітектурі проекту.

- Аналіз модульності. Додаток дозволяє оцінити модульність проекту, виявити модулі з високою зв'язністю і запропонувати рекомендації щодо покращення структури коду.

- Інтеграція з іншими інструментами. CodeGraph може інтегруватися з іншими інструментами розробки, такими як IDE, системи контролю версій і т.д.

Переваги використання CodeGraph є наступними:

- Покращення якості коду. Завдяки візуалізації залежностей розробники можуть виявляти і усувати проблеми в архітектурі проекту на ранніх етапах розробки.

- Збільшення продуктивності. Розуміння структури проекту дозволяє розробникам швидше знаходити і виправляти помилки, а також вносити зміни в код.

- Спрощення співпраці. CodeGraph може бути корисний для командних проектів, оскільки дозволяє всім учасникам проекту мати єдине уявлення про структуру коду.

Це дозволяє користувачам візуально досліджувати зв’язок між різними файлами в проекті. Крім того, він надає користувачеві статистичні дані про проект та інформацію про окремі файли. За допомогою деяких тестів на зручність використання було встановлено, що CodeGraph сприяє кращому розумінню проекту JavaScript і забезпечує більш приємний досвід користувача, ніж традиційні текстові інструменти». Зображення CodeGraph показано на рисунку 2.1.

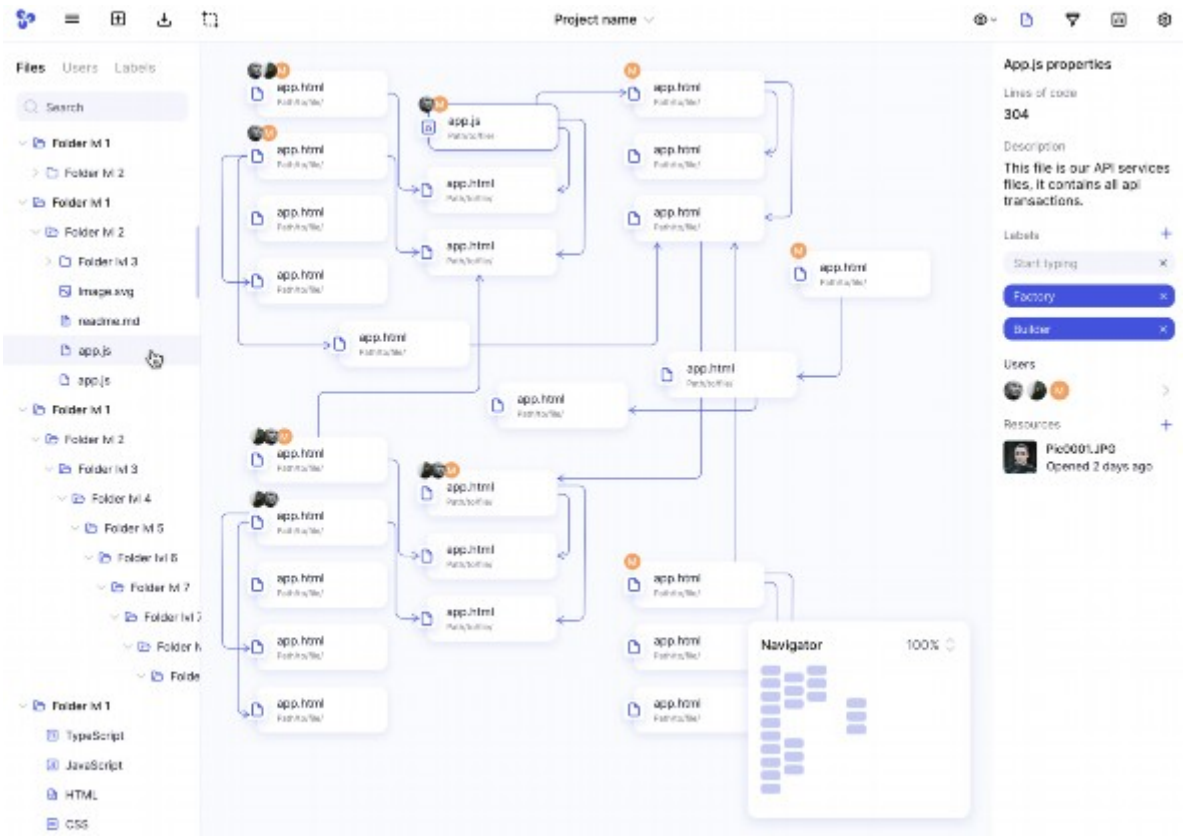


Рис. 2.1. Огляд платформи візуалізації графів CodeGraph

Даний інструмент візуалізує графік залежностей проекту JavaScript. CodeGraph є комерційним проектом. Можна сказати, що CodeGraph є трохи розширенішим ніж пропонований, пропонуючи більше налаштувань і різних функцій. Наприклад, CodeGraph пропонує багато макетів і налаштувань макета, показує додаткову статистику, має інтеграцію керування версіями git, міні-карту для графіка та додаткові параметри фільтрації. Однак, йому бракує можливостей згортання та розширення вузлів, кольорового кодування на графіку, динамічних розмірів вузлів (на основі розміру коду файлу) і, звичайно, виявлення внутрішньої архітектури вихідного коду за допомогою графів викликів функцій і вилучення API бібліотеки. У більшості випадків CodeGraph є надзвичайно надійним.

### *2.1.2. Огляд інструменту Hunter для візуалізації програм JavaScript*

Hunter [34] — це «інструмент для візуалізації програм JavaScript. Hunter візуалізує вихідний код за допомогою набору узгоджених

представлень, які включають діаграму зв’язку вузла, яка зображує залежності між компонентами системи, і карту дерева, яка допомагає програмістам орієнтуватися під час навігації по її структурі». Hunter – це потужний інструмент, спеціально розроблений для візуалізації та аналізу JavaScript-коду. Він дозволяє розробникам отримати глибоке розуміння структури та логіки своїх додатків, виявляти потенційні проблеми та оптимізувати код.

На рисунку 2.2 показано графічний інтерфейс інструменту Hunter, а його компоненти відповідно позначені і описані нижче.

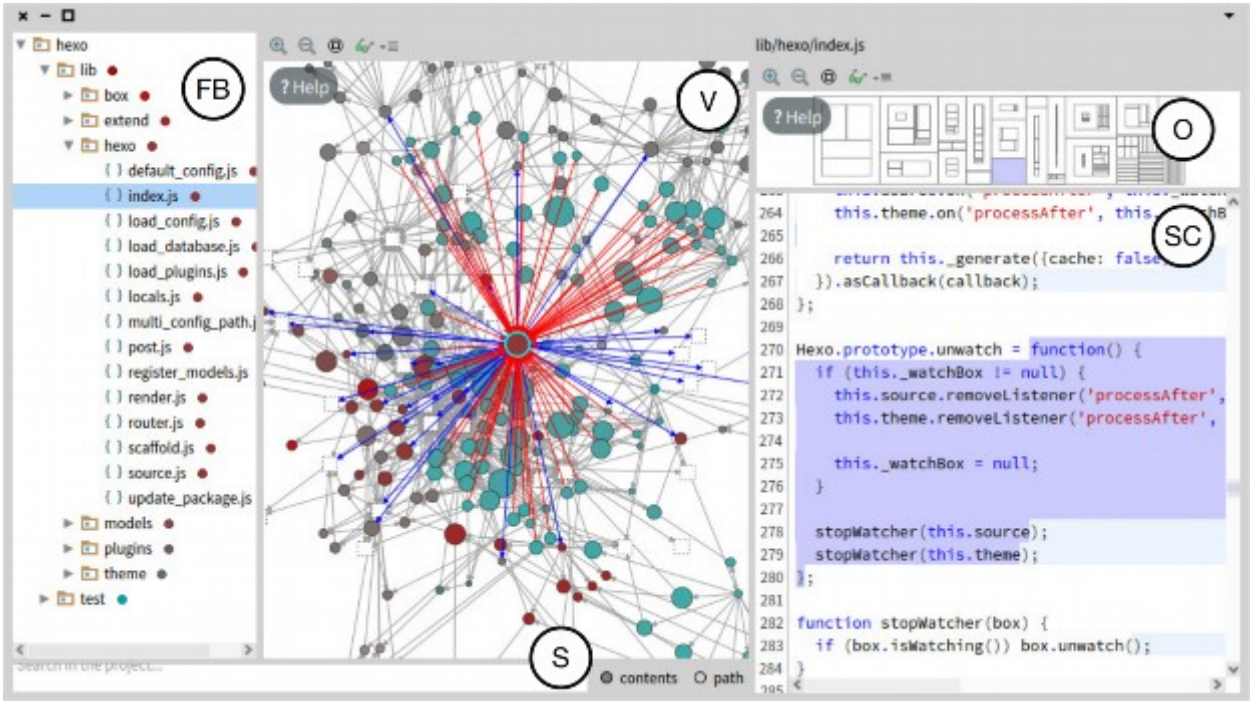


Рис. 2.2. Графічний інтерфейс Hunter

На рисунку 2.2: крайня ліва панель (FB) — це браузер файлів, також включений у більшість IDE. Центральна панель — це перегляд залежностей файлів (V), який пов’язує залежності між файлами вихідного коду JavaScript. Верхня права панель (O) показує структуру вибраного файлу з точки зору вкладеності функцій. Нижня ліва панель (S) — це вікно пошуку, за допомогою якого можна знайти певні файли або функції, розташовані у V. Нарешті, нижня права панель (SC) показує вихідний код вибраного файлу з виділеною функцією.

Алгоритм роботи Hunter наступний:

1. Парсинг коду: Hunter аналізує JavaScript-код і будує його абстрактне синтаксичне дерево (AST).

2. Побудова графа: На основі AST будується граф, який відображає структуру коду.

3. Візуалізація: Граф відображається у зручному для користувача інтерфейсі, де можна інтерактивно досліджувати різні частини коду.

4. Аналіз: Hunter проводить різні аналізи коду, такі як аналіз потоку даних, виявлення антипаттернів і профілювання продуктивності.

5. Представлення результатів: Результати аналізу відображаються у вигляді графіків, діаграм і таблиць, що полегшує їх інтерпретацію.

Основні функціональні можливості Hunter:

- Візуалізація дерева абстрактного синтаксису (AST): Hunter будує детальне графічне представлення AST JavaScript-коду, що дозволяє побачити, як код структурований і як різні елементи пов'язані між собою.

- Аналіз потоку даних: Інструмент аналізує потік даних у програмі, відстежуючи, як значення передаються між різними частинами коду. Це допомагає виявити потенційні помилки та неточності.

- Виявлення антипаттернів: Hunter може ідентифікувати поширені антипаттерни в JavaScript-коді, такі як занадто великі функції, циклічні залежності, невикористаний код і т.д.

- Профілювання продуктивності: Інструмент дозволяє профілювати JavaScript-додатки, щоб визначити вузькі місця і оптимізувати виконання коду.

- Інтеграція з IDE: Hunter часто інтегрується з популярними IDE (інтегрованими середовищами розробки), що дозволяє використовувати його безпосередньо під час написання коду.

Hunter часто використовують для наступних задач:

- Розуміння складного коду: Hunter допомагає розробникам краще зрозуміти складні і заплутані фрагменти коду.

- Виявлення помилок: Інструмент може виявити потенційні помилки і неточності в коді, які можуть призвести до проблем під час виконання програми.

- Оптимізація продуктивності: Hunter дозволяє виявити вузькі місця в коді і оптимізувати його виконання.

- Навчання: Hunter може бути корисним для навчання нових розробників, оскільки він дозволяє візуалізувати, як працює JavaScript-код.

- Рефакторинг: Інструмент може допомогти в процесі рефакторингу коду, дозволяючи побачити, які частини коду можуть бути спрощені або переписані.

Знову ж таки, зображення графіка залежностей служить ключовим компонентом Hunter. Однією з нових особливостей Hunter є кольорове кодування папок у файловому браузері. Колір батьківської папки файлу, який представляє кожен вузол, використовується для ідентифікації кожного вузла в графі залежностей. Це дає змогу користувачеві візуально пов’язати візуалізацію графіка залежностей і перегляд файлового браузера. Крім того, розмір кожного вузла в графі залежностей обернено пропорційний кількості рядків коду, присутніх у файлі. Даний інструмент немає можливості підключати функції через їх виклики, але отримує можливість спостерігати за структурою та вкладеністю функцій. Це компроміс, який варто розглянути.

### *2.1.3. Опис Eunice – інструменту статичного аналізу коду*

Eunice [35] — це інструмент, який може сканувати проекти C# і JavaScript і «покращує зв’язок, зв’язок і модульність програмного забезпечення за допомогою ієрархічної структури та спрощених однонаправлених залежностей. Eunice аналізує вихідний код, робить висновок про його структуру та показує, чи збігаються залежності».

Основні функціональні можливості Eunice наступні:

- Виявлення вразливостей: Eunice може виявляти відомі вразливості в коді, такі як SQL-ін'єкції, XSS, CSRF та інші.

- Статичний аналіз коду: Eunice аналізує код без його виконання, виявляючи помилки, які можуть бути пропущені компілятором або під час ручного тестування.

- Підтримка множинних мов: Інструмент підтримує як C#, так і JavaScript, що дозволяє використовувати його для сканування різноманітних проектів.

- Перевірка відповідності стандартам кодування: Інструмент дозволяє налаштувати правила перевірки відповідності коду корпоративним стандартам або загальноприйнятим рекомендаціям (наприклад, код стилю, номінації, використання певних конструкцій).

- Генерація звітів: Eunice генерує детальні звіти про результати сканування, що дозволяє легко ідентифікувати та усувати виявлені проблеми.

- Інтеграція з IDE: Багато інструментів статичного аналізу, таких як Eunice, інтегруються з популярними IDE (інтегрованими середовищами розробки), що дозволяє проводити аналіз коду в режимі реального часу.

На рисунку 2.3 показано звіт Eunice.

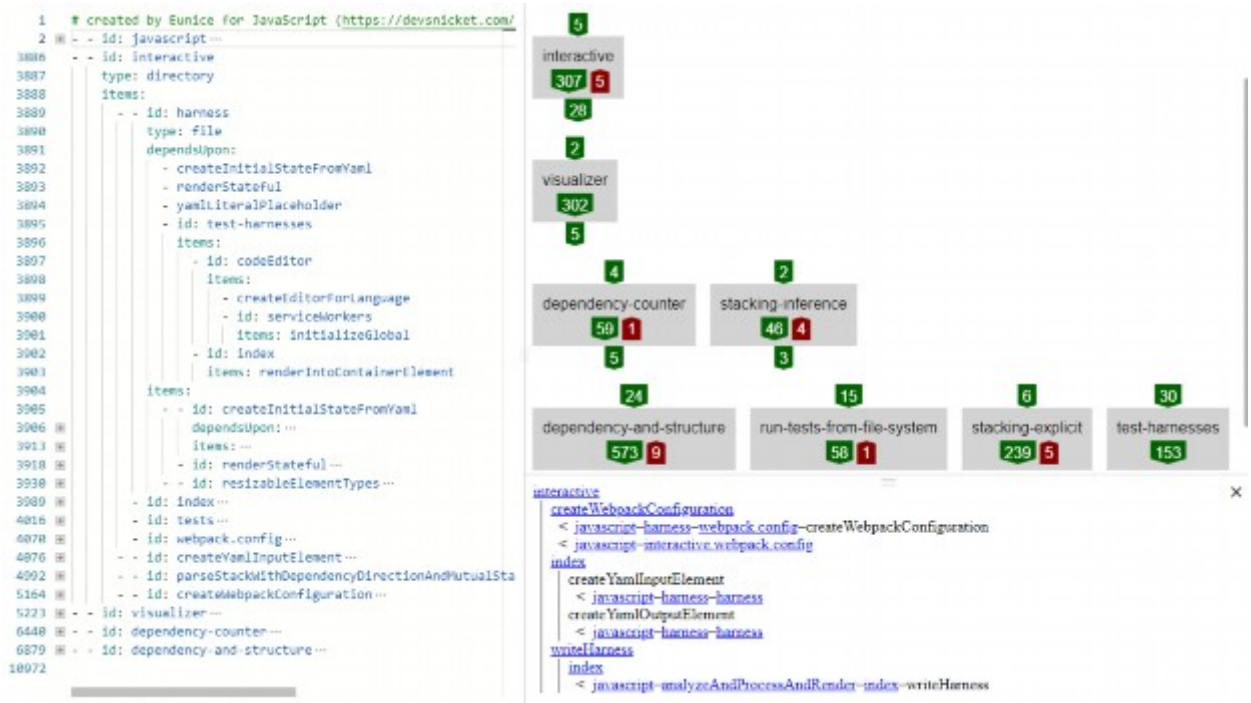


Рис. 2.3. HTML-звіт Eunice

Алгоритм роботи Eunice наступний:

1. Сканування проекту: Eunice аналізує код проекту, будуючи його абстрактне синтаксичне дерево (AST).

2. Аналіз AST: На основі AST інструмент проводить різні аналізи, такі як аналіз потоку даних, пошук потенційних помилок і вразливостей, перевірка відповідності стандартам кодування.

3. Генерація звіту: Результати аналізу відображаються у вигляді детального звіту, який містить інформацію про виявлені проблеми, їхню локалізацію в коді та рекомендації щодо усунення.

Для виконання Eunice у проектах JavaScript потрібна одна команда, а саме npx eunice. Потім він створює HTML-звіт із включеною візуалізацією. Eunice візуалізує вкладену деревоподібну структуру, і не візуалізує мережу графів, що згортається/розширюється. Слід визнати, що Eunice вражає та містить багато інформації, але також досить складний і має круту криву навчання.

Переваги використання Eunice наступні:

- Раннє виявлення проблем: Проблеми виявляються на ранніх етапах розробки, що дозволяє заощадити час і кошти.

- Збільшення надійності: Зменшується ризик виникнення помилок у виробництві.

- Покращення безпеки: Знижується ризик успішних атак на систему.

- Стандартизація коду: Забезпечується єдиний стиль кодування в проекті.

## 2.2. Дослідження алгоритму Breadthfirst для побудови візуалізацій

Пошук у ширину (Breadth-First Search, BFS) - це алгоритм обходу графа, який досліджує всі сусідів вершини на поточному рівні глибини перш ніж перейти до вершин на наступному рівні. Цей алгоритм часто

використовується для побудови дерев або графів, а також для вирішення задач, пов'язаних з пошуком найкоротших шляхів у незважених графах.

Опис алгоритму "Breadthfirst":

1. Ініціалізація:

- Обирається кореневий вузол графа, від якого почнеться побудова (якщо граф не має явного кореня, його можна вибрати довільно).

- Створюється порожня черга (Queue) для зберігання вузлів, які необхідно обробити.

2. Додавання кореневого вузла:

- Кореневий вузол додається в чергу, а також у список відвіданих вузлів.

- Початковий рівень для кореневого вузла встановлюється як 0.

3. Обробка вузлів у черзі:

- Поки черга не порожня, алгоритм вибирає вузол з черги для обробки:

- Витягує поточний вузол із черги.

- Усі суміжні вузли (сосіди) поточного вузла, які ще не відвідані, додаються в чергу.

- Для кожного сусіда фіксується рівень, який на 1 більше, ніж рівень поточного вузла.

4. Розміщення вузлів за рівнями:

- Вузли, які належать до одного рівня, розташовуються на одній горизонтальній лінії (або вертикальній, якщо напрям візуалізації обернений).

- Відстань між рівнями зазвичай фіксована, але може варіюватися залежно від кількості вузлів.

5. Завершення:

- Алгоритм продовжує обробляти вузли з черги до тих пір, поки всі вузли графа не будуть оброблені.

- В результаті отримуємо граф, у якому вузли рівнів розміщені в порядку їх віддаленості від кореневого вузла.

BFS може бути реалізовано на різних мовах програмування, таких як Python, C++, Java, JavaScript.

Застосування алгоритму наступні:

- Побудова дерев: Алгоритм BFS часто використовується для побудови дерев, таких як дерева пошуку або дерева відвідувань.

- Пошук найкоротших шляхів у незважених графах: Оскільки BFS досліджує вершини рівень за рівнем, він гарантує, що перший раз, коли буде знайдена цільова вершина, шлях до неї буде найкоротшим.

- Аналіз графів: BFS може бути використаний для визначення компонент зв'язності графа, перевірки графа на двочастковість та інших задач.

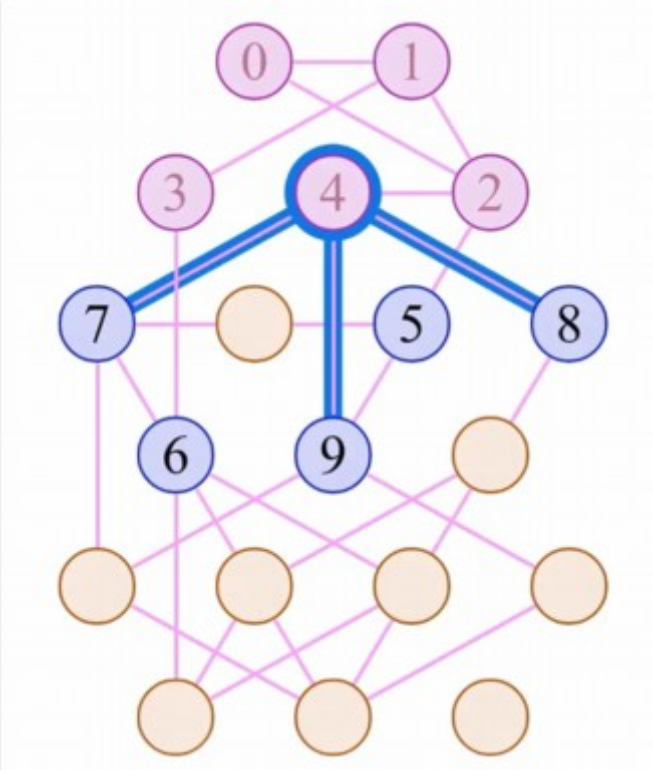


Рис. 2.4. Візуалізація роботи алгоритму Breadth-First Search, BFS

Існують наступні модифікації алгоритму:

- Пошук в глибину (Depth-First Search, DFS): Інший алгоритм обходу графа, який досліджує гілки графа якомога глибше перед тим, як повернутися назад.

- Алгоритм Дейкстри: Використовується для пошуку найкоротших шляхів у зважених графах.

- Алгоритм A:* Є евристичним алгоритмом пошуку, який використовується для знаходження найкоротших шляхів у графах.

Алгоритм BFS є фундаментальним алгоритмом для роботи з графами. Він має широке застосування в різних областях комп'ютерних наук, від розробки ігор до аналізу даних. Розуміння принципу роботи BFS є важливим для будь-якого розробника, який працює з графовими структурами даних.

## 2.3. Дослідження особливостей макету Cola в контексті побудови візуалізацій засобами JS

Макет Cola для бібліотеки cytoscape.js — це один із варіантів розміщення елементів графа на полотні. Він використовує механізм фізичного моделювання для автоматичного визначення позицій вузлів, враховуючи сили відштовхування між вузлами і сили притягання між ребрами. Макет базується на бібліотеці WebCola, яка підтримує кілька специфічних функцій для роботи з графами, таких як обмеження положень вузлів, розміщення вузлів за групами та інше.

Основні можливості макету Cola:

- Розташування вузлів з урахуванням відштовхування між ними, щоб уникнути їхнього накладання.

- Групування вузлів у кластери або окремі області графа.

- Можливість фіксувати вузли в певних позиціях, щоб інші вузли розміщувались відносно цих зафіксованих точок.

- Робота з неорієнтованими графами та можливість візуалізувати графи різної складності.

Лістинг 2.1. Приклад використання макету Cola в Cytoscape.js

```
var cy = cytoscape({
```

```
      container: document.getElementById('cy'), // Контейнер для
графа
      elements: [
        // Вузли та ребра
        { data: { id: 'a' } },
        { data: { id: 'b' } },
        { data: { id: 'c' } },
        { data: { id: 'd' } },
        { data: { id: 'e' } },
        { data: { id: 'ab', source: 'a', target: 'b' } },
        { data: { id: 'bc', source: 'b', target: 'c' } },
        { data: { id: 'cd', source: 'c', target: 'd' } },
        { data: { id: 'de', source: 'd', target: 'e' } },
      ],
      style: [ // Стилі для вузлів і ребер
        {
          selector: 'node',
          style: {
            'background-color': '#0074D9',
            'label': 'data(id)'
          }
        },
        {
          selector: 'edge',
          style: {
            'width': 2,
            'line-color': '#FF4136',
            'target-arrow-color': '#FF4136',
            'target-arrow-shape': 'triangle'
          }
        }
      ],

      layout: {
        name: 'cola', // Вказуємо макет Cola
        nodeSpacing: 50, // Відстань між вузлами
        edgeLengthVal: 45, // Довжина ребра
        maxSimulationTime: 1000, // Максимальний час моделювання
        fit: true, // Підігнати граф під розміри контейнера
        randomize: false, // Чи треба випадкове початкове
розташування вузлів
        avoidOverlap: true, // Уникнення накладання вузлів
        handleDisconnected: true, // Обробляти незв'язані вузли
        convergenceThreshold: 0.01 // Поріг збіжності для
припинення моделювання
      }
    });
```

Наведемо опис ключових параметрів макету Cola:

- **nodeSpacing**: Визначає мінімальну відстань між вузлами, щоб уникнути їх накладання.

- **edgeLengthVal**: Стандартна довжина ребер, що визначає відстань між вузлами, з’єднаними ребром.

- **maxSimulationTime**: Максимальний час симуляції, після якого розміщення графа завершується, навіть якщо не досягнуто збіжності.

- **fit**: Якщо значення true, то граф буде автоматично масштабовано під розміри контейнера.

- **randomize**: Якщо true, початкове розміщення вузлів буде випадковим.

- **avoidOverlap**: Якщо true, алгоритм намагатиметься уникати накладання вузлів.

- **handleDisconnected**: Забезпечує обробку незв'язаних компонентів графа.

- **convergenceThreshold**: Параметр, який визначає, коли моделювання можна зупинити, якщо система наближається до збіжності.

Макет Cola корисний для візуалізації графів з великою кількістю вузлів і складною структурою, оскільки дозволяє створювати більш зрозумілу і наочну картину, зменшуючи накладання вузлів і ребер.

## 2.4. Алгоритм побудови візуалізацій Cose-Bilkent

Макет Cose-Bilkent для cytoscape.js — це модифікований варіант силового макета, що базується на методі фізичного моделювання ієрархічних та неієрархічних графів. Він використовується для візуалізації великих графів з високою кількістю вузлів і ребер. Макет Cose-Bilkent був розроблений для покращення традиційного силового макета Cose, з акцентом на більшу стабільність, продуктивність і якість розташування.

Основні можливості макета Cose-Bilkent:

- Покращена продуктивність: ефективний розрахунок для великих графів.

- Уникнення накладання вузлів за допомогою фізичних моделей відштовхування і притягання.

- Підтримка ієрархічних графів: макет підтримує відображення графів, де певні групи вузлів мають підпорядковану структуру.

- Можливість налаштування параметрів для тонкого налаштування: можна керувати силою відштовхування, притягання, відстанями між вузлами, рівнем випадковості тощо.

Лістинг 2.2. Приклад використання макета Cose-Bilkent у Cytoscape.js

```
var cy = cytoscape({
  container: document.getElementById('cy'), // Контейнер для графа

  elements: [
    // Вузли та ребра
    { data: { id: 'a' } },
    { data: { id: 'b' } },
    { data: { id: 'c' } },
    { data: { id: 'd' } },
    { data: { id: 'e' } },
    { data: { id: 'ab', source: 'a', target: 'b' } },
    { data: { id: 'bc', source: 'b', target: 'c' } },
    { data: { id: 'cd', source: 'c', target: 'd' } },
    { data: { id: 'de', source: 'd', target: 'e' } },
  ],

  style: [ // Стилі для вузлів і ребер
    {
      selector: 'node',
      style: {
        'background-color': '#0074D9',
        'label': 'data(id)'
      }
    },
    {
      selector: 'edge',
      style: {
        'width': 2,
        'line-color': '#FF4136',
        'target-arrow-color': '#FF4136',
        'target-arrow-shape': 'triangle'
      }
    }
  ],

  layout: {
    name: 'cose-bilkent', // Вказуємо макет Cose-Bilkent
    idealEdgeLength: 100, // Бажана довжина ребер
    nodeRepulsion: 4500, // Сила відштовхування вузлів
    gravity: 0.25, // Сила тяжіння (для центрування)
```

```
        edgeElasticity: 0.45, // Еластичність ребер
        nestingFactor: 0.1, // Множник для обробки вкладених елементів
        numIter: 2500, // Кількість ітерацій для моделювання
        tile: true, // Розміщення із плиткою для поліпшення розташування
        animate: 'end', // Анімація в кінці побудови графа
        randomize: false, // Якщо true, початкове розміщення вузлів буде випадковим
        fit: true, // Масштабування графа під розмір контейнера
        padding: 30 // Відступи між графом і межами контейнера
      }
    });
```

Наведемо опис ключових параметрів макета Cose-Bilkent:

- **idealEdgeLength**: Визначає бажану довжину ребра між двома вузлами. Велика довжина збільшує відстань між вузлами.

- **nodeRepulsion**: Сила відштовхування між вузлами. Вищі значення призводять до більшої відстані між вузлами, щоб уникнути їхнього накладання.

- **gravity**: Визначає силу тяжіння до центру графа, яка допомагає утримувати граф у межах екрану.

- **edgeElasticity**: Визначає жорсткість ребер; чим більше значення, тим жорсткішими будуть ребра, що зменшить їхню гнучкість.

- **nestingFactor**: Використовується для обробки вкладених елементів (якщо є групи вузлів).

- **numIter**: Кількість ітерацій для завершення алгоритму. Чим більше ітерацій, тим краще оптимізується граф, але це може впливати на продуктивність.

- **tile**: Якщо true, вузли розміщуються за принципом плитки, щоб поліпшити загальну структуру.

- **animate**: Якщо анімація включена (наприклад, 'during' або 'end'), вузли плавно переходять у свої кінцеві позиції після остаточного завершення розміщення.

- **randomize**: Якщо true, початкові позиції вузлів будуть випадковими. Це може вплинути на кінцевий вигляд графа.

- **fit**: Якщо true, граф автоматично масштабується до розміру контейнера після завершення побудови.

- **padding**: Відступи між межами графа і контейнером.

Макет **Cose-Bilkent** ідеально підходить для роботи з великими графами, де необхідна стабільність і ефективність. Він використовується для створення структурованих візуалізацій складних мережевих систем, соціальних графів, організаційних ієрархій тощо.

## 2.5. Дослідження набору алгоритмів для компонування графів Eclipse Layout Kernel

ELK (Eclipse Layout Kernel) — це набір алгоритмів для компонування графів, розроблений спільнотою Eclipse Foundation. ELK надає потужні та гнучкі механізми для візуалізації графів у різних програмних середовищах, включаючи Cytoscape.js. Його головна мета — забезпечити якісне компонування графів, яке зберігає читабельність і підтримує широкий спектр типів графів.

ELK інтегрується в Cytoscape.js через спеціальні плагіни та адаптери, що дозволяють використовувати його алгоритми для організації графів у різних візуальних стилях.

Особливості ELK:

- Підтримка різних типів графів: ELK може обробляти як орієнтовані, так і неорієнтовані графи, ієрархічні структури, деревоподібні графи тощо.

- Висока гнучкість налаштувань: Підтримує безліч параметрів для налаштування розташування вузлів, таких як відстань між ними, структурування, вирівнювання та уникнення накладання.

- Підтримка ієрархій: Можна розташовувати вузли за рівнями, що дозволяє якісно відображати складні системи з багаторівневою структурою.

- Робота з великими графами: ELK спеціалізується на обробці графів із великою кількістю вузлів і ребер, оптимізуючи процес компонування.

Лістинг 2.3. Приклад використання макета ELK в Cytoscape.js

```
var cy = cytoscape({
 container: document.getElementById('cy'),
elements: [
// ... елементи графа
],
style: [
// ... стиль графа
], layout: {
name: 'elk',
algorithm: 'layered', // Вибір алгоритму
// ... інші параметри
}
});
```

Наведемо опис ключових параметрів макета ELK:

- **algorithm**: Алгоритм компонування. ELK підтримує різні алгоритми компонування, такі як:

- **layered**: для багаторівневих графів.

- **force**: силовий макет для орієнтованих та неорієнтованих графів.

- **box**: для компактних компоновок.

- **direction**: Вказує напрямок розташування (вгору, вниз, ліворуч, праворуч).

- **spacing**: Визначає відстань між вузлами графа.

- **nodePlacement**: Метод розміщення вузлів. Один із популярних методів — **BRANDES_KOEPF**.

- **edgeRouting**: Тип маршрутизації ребер (наприклад, **ORTHOGONAL** для прямокутних з'єднань).

Розглянемо перелік основних алгоритмів ELK для Cytoscape.js:

- **Layered**: Для ієрархічних графів, де вузли розташовуються рівнями, що дозволяє візуалізувати орієнтовані графи або дерева.

- **Force**: Алгоритм силового компонування, який базується на фізичних моделях відштовхування вузлів і притягання ребер.

- **Orthogonal**: Алгоритм, який мінімізує кількість перетинів між ребрами та використовує прямі кути для з'єднання вузлів.

- **Circular**: Розміщує вузли в колі, оптимізуючи відстань між ними.

Наведемо основні переваги використання ELK у Cytoscape.js:

- **Гнучкість.** ELK надає безліч варіантів компонування, які підходять для різних типів графів.

- **Якість розміщення.** Алгоритми ELK розроблені для покращення читабельності графів, мінімізуючи перетини ребер та накладання вузлів.

- **Підтримка великих графів.** ELK ефективно обробляє графи з тисячами вузлів та ребер, що робить його корисним для складних мереж.

Цей макет широко застосовується для візуалізації складних структур у таких сферах, як біоінформатика, соціальні мережі, мережі залежностей, ієрархії процесів тощо.

## Висновки до розділу

У другому розділі проведено детальне дослідження методів, алгоритмів та інструментів для візуалізації структури програмного забезпечення. Основну увагу приділено аналізу існуючих інструментів візуалізації, таких як CodeGraph, Hunter та Eunice, які застосовуються для роботи з проектами на JavaScript. Огляд цих інструментів показав їхні переваги та недоліки, зокрема у контексті візуалізації графів залежностей і статичного аналізу коду.

Розділ також охоплює дослідження популярних алгоритмів для побудови візуалізацій. Алгоритм Breadthfirst проаналізовано як базовий підхід для обходу та відображення графів, що дозволяє структурувати інформацію за рівнями. Особливу увагу приділено макету Cola, який надає можливість побудови інтерактивних графічних структур за допомогою JavaScript, забезпечуючи динамічне відображення взаємозв'язків між елементами.

Крім того, описано алгоритм Cose-Bilkent, що використовує фізичні моделі для оптимізації розташування елементів графу, забезпечуючи високу якість візуалізацій. На завершення, розглянуто набір алгоритмів Eclipse Layout Kernel (ELK), який забезпечує широкі можливості для компонування

графів із різними конфігураціями та застосовується для складних ієрархічних структур.

Отже, досліджені інструменти та алгоритми забезпечують ефективні методи візуалізації складних структур програмного забезпечення, що сприяє покращенню розуміння архітектури та залежностей у проектах на JavaScript.

# РОЗДІЛ 3. ПРЕДСТАВЛЕННЯ МЕТОДОЛОГІЇ ІНТЕРАКТИВНОЇ ВІЗУАЛІЗАЦІЇ СТРУКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 3.1. Опис підходу та розробка структури рішення

У цьому розділі ми досліджуємо методи, які використовуються для вирішення наших дослідницьких питань з першого розділу. Спочатку ми обговорюємо загальний підхід, а потім встановлені обмеження та цілі. Потім ми розглядаємо дизайн інструменту та прийняті дизайнерські рішення.

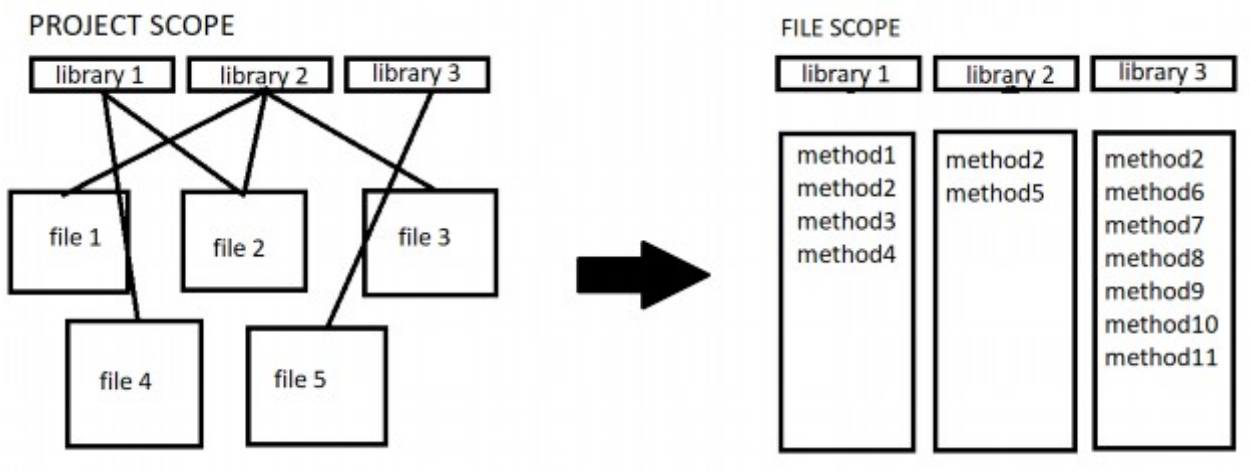


Рис. 3.1. Попередній макет інструменту візуалізації

На рисунку 3.1 - подання, яке з'єднує бібліотеки з файлами проекту (ліворуч), і подання, яке перераховує використання API бібліотек для кожного файлу (праворуч).

Для дослідницької проблеми ми хочемо знайти способи вказати використання бібліотек у будь-якому програмному проекті таким чином, щоб зробити його більш зрозумілим. Аналіз і візуалізація є найефективнішими способами досягти цього. Щоб фактично досягти цих двох компонентів, можна створити інструмент. У розділі 3.3 викладені цілі цього інструменту. В цьому розділі досліджується ітеративна розробка програмного

забезпечення, де воно ітеративно вдосконалюється, а інструмент називається L library U sage в аналізаторі N ode.js , скорочено LUNA.

LUNA (Library Usage in Node.js Analyzer) – це інструмент, спеціально розроблений для аналізу та візуалізації того, як бібліотеки використовуються в проектах на Node.js. Він призначений для спрощення розуміння складної структури сучасних програмних додатків, які часто побудовані на великій кількості взаємопов'язаних бібліотек.

Основні функції LUNA:

- Аналіз абстрактного синтаксичного дерева (AST): LUNA проходить через вихідний код проекту і будує абстрактне подання його структури. Це дозволяє інструменту точно визначити, які бібліотеки використовуються і як вони взаємодіють між собою.

- Візуалізація взаємодій бібліотек: Інструмент створює інтерактивний граф, який відображає всі залежності між різними бібліотеками в проекті. Це дозволяє розробникам швидко отримати загальне уявлення про структуру свого додатку та виявити потенційні проблеми.

- Відновлення архітектури програмного забезпечення: На основі аналізу AST і візуалізації взаємодій бібліотек, LUNA може допомогти відновити загальну архітектуру програми. Це особливо корисно для великих і складних проектів, де архітектура може бути не явно задокументована.

Особливості LUNA:

- Покращення розуміння проекту: LUNA допомагає розробникам краще зрозуміти, як різні частини їхнього проекту пов'язані між собою, особливо коли мова йде про використання сторонніх бібліотек.

- Виявлення потенційних проблем: Інструмент може допомогти виявити надмірні залежності, конфлікти між бібліотеками та інші проблеми, які можуть вплинути на продуктивність або стабільність програми.

- Спрощення процесу модернізації: Коли необхідно оновити або замінити бібліотеку, LUNA може допомогти оцінити вплив таких змін на інші частини проекту.

- Навчання нових розробників: LUNA може бути корисним інструментом для нових розробників, які приєднуються до проекту, оскільки він допомагає швидко ознайомитися з його структурою.

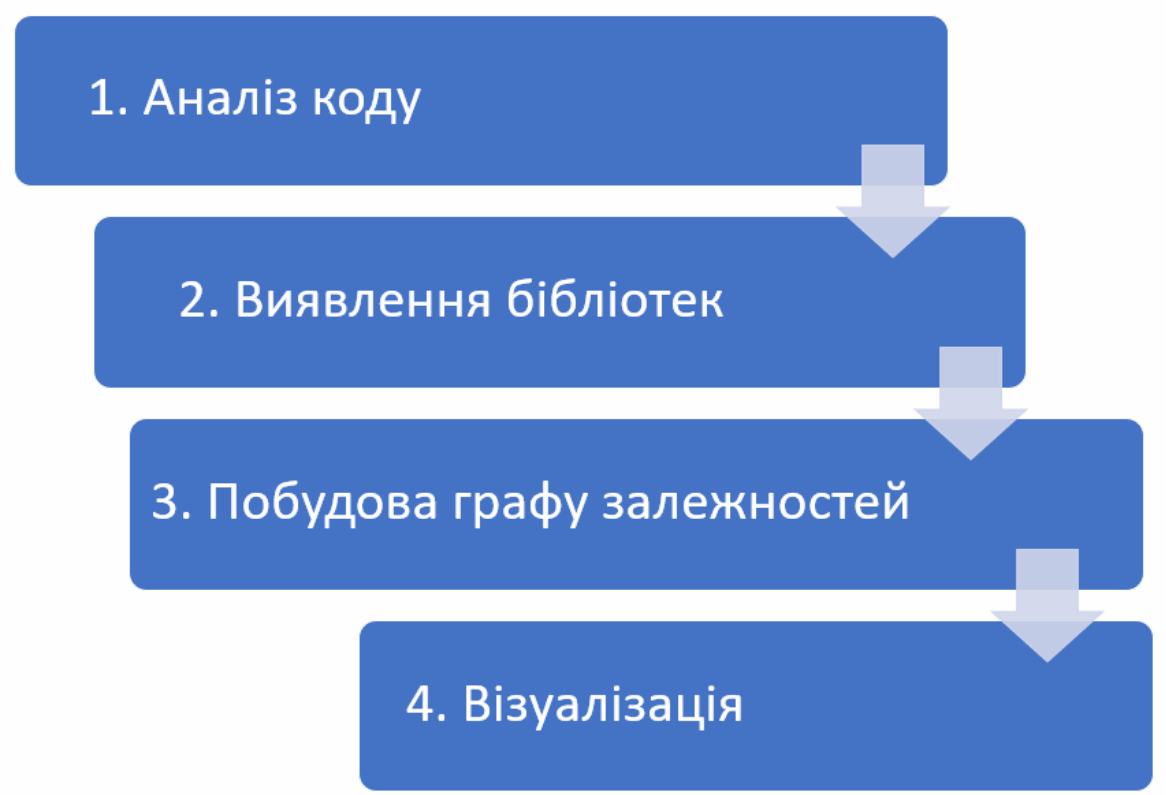


Рис. 3.2. Алгоритм виконання візуалізації структури ПЗ

Алгоритм роботи LUNA (рис. 3.2):

- Аналіз коду: LUNA аналізує вихідний код проекту і будує абстрактне синтаксичне дерево (AST), яке представляє структуру коду в пам'яті.

- Виявлення бібліотек: Інструмент виявляє всі використані в проекті бібліотеки і їхні версії.

- Побудова графу залежностей: LUNA будує граф, де кожна вершина представляє бібліотеку, а ребра відображають залежності між ними.

- Візуалізація: Інтерактивний граф відображається користувачеві, дозволяючи йому досліджувати структуру проекту і взаємодію між бібліотеками.

LUNA є потужним інструментом для розробників, які працюють з Node.js. Він допомагає краще зрозуміти структуру проекту, виявити потенційні проблеми і спростити процес розробки. Завдяки своїй здатності візуалізувати складні залежності, LUNA стає незамінним помічником для команд, які працюють над великими і складними проектами.

## 3.2. Розробка діаграми архітектури системи

JavaScript — це мова програмування, яка разом із HTML і CSS є однією з основних технологій Всесвітньої павутини. Сьогодні майже всі веб-сайти використовують JavaScript для керування поведінкою веб-сторінок. Код виконується спеціальним двигуном JavaScript, який є у всіх основних веб-переглядачах. JavaScript відповідає стандарту ECMAScript. Він високорівневий і має динамічну типізацію, об’єктну орієнтацію на основі прототипу та першокласні функції. Він мультипарадигмальний, підтримує керований подіями, функціональний та імперативний стилі програмування.

Я зосереджуся на аналізі мови програмування JavaScript, зокрема середовища виконання node.js. Тому що він добре відомий тим, що має велику кількість сторонніх бібліотек, і тому що він добре відомий мені (роки практичного досвіду). Бібліотеки в цій екосистемі більш відомі як модулі NPM (Node Package Manager), але ми залишимо їх називати бібліотеками або залежностями, тобто бібліотеками, від яких залежить проект або бібліотека.

Node.js - це потужне, багатоплатформне середовище виконання JavaScript з відкритим кодом, яке дозволяє розробникам створювати високопродуктивні мережеві додатки. Хоча Node.js зазвичай асоціюється з розробкою серверної частини, він також відіграє важливу роль у візуалізації даних та взаємодії з різноманітними бібліотеками програмного забезпечення.

Оскільки JavaScript є основною мовою для веб-розробки, використання Node.js дозволяє використовувати єдину мову для як клієнтської, так і серверної частини, що спрощує розробку.

Node.js володіє багатою екосистемою модулів: npm (Node Package Manager) надає доступ до величезної кількості готових модулів для візуалізації, таких як D3.js, Chart.js, Echarts та інших. Ці модулі забезпечують широкий спектр можливостей для створення інтерактивних візуалізацій.

Node.js використовує модель введення-виведення на основі подій, що дозволяє йому ефективно обробляти великі обсяги даних і створювати швидкі та плавні візуалізації. Особливість асинхронного введення-виведення дозволяє Node.js виконувати кілька завдань одночасно, не блокуючи виконання основного потоку. Це особливо корисно при роботі з базами даних та іншими ресурсами, що можуть бути повільними.

Як пояснюється в розділі 3.1, пропонований підхід включає створення сканера проекту для аналізу, першого основного компонента LUNA. Цей сканер виявить усі бібліотеки, які використовуються в проекті. Для кожної бібліотеки також буде вказано використання, тобто:

1. Скільки функціональних можливостей бібліотеки використовується для кожного файлу з точки зору доступного API (методи та дані);
2. Де він використовується, у яких файлах і в яких програмних компонентах (функціях і класах);
3. Чи є бібліотеки зовнішніми (від сторонніх розробників) чи внутрішніми

Методи виявлення архітектури програмного забезпечення використовуються для визначення місць в архітектурі, де використовуються бібліотеки. Детальніше про це можна буде описано в наступних розділах. Крім того, візуалізація буде другим основним компонентом LUNA. Ось короткий перелік програмних вимог для LUNA:

1. LUNA може бути доступний як пакет NPM, як Node.js Package або як окрема програма;
2. Звіти LUNA можна переглядати у веб-браузерах на основі хрому (Chrome, Edge, Brave), Firefox;

3. LUNA є виконуваним у Windows, Linux, Mac OC;

4. LUNA є виконуваним на Android, iOS;

5. LUNA може працювати з проектами Node.js, TypeScript, може сканувати онлайн-репозиторії проектів (наприклад, з GitHub).

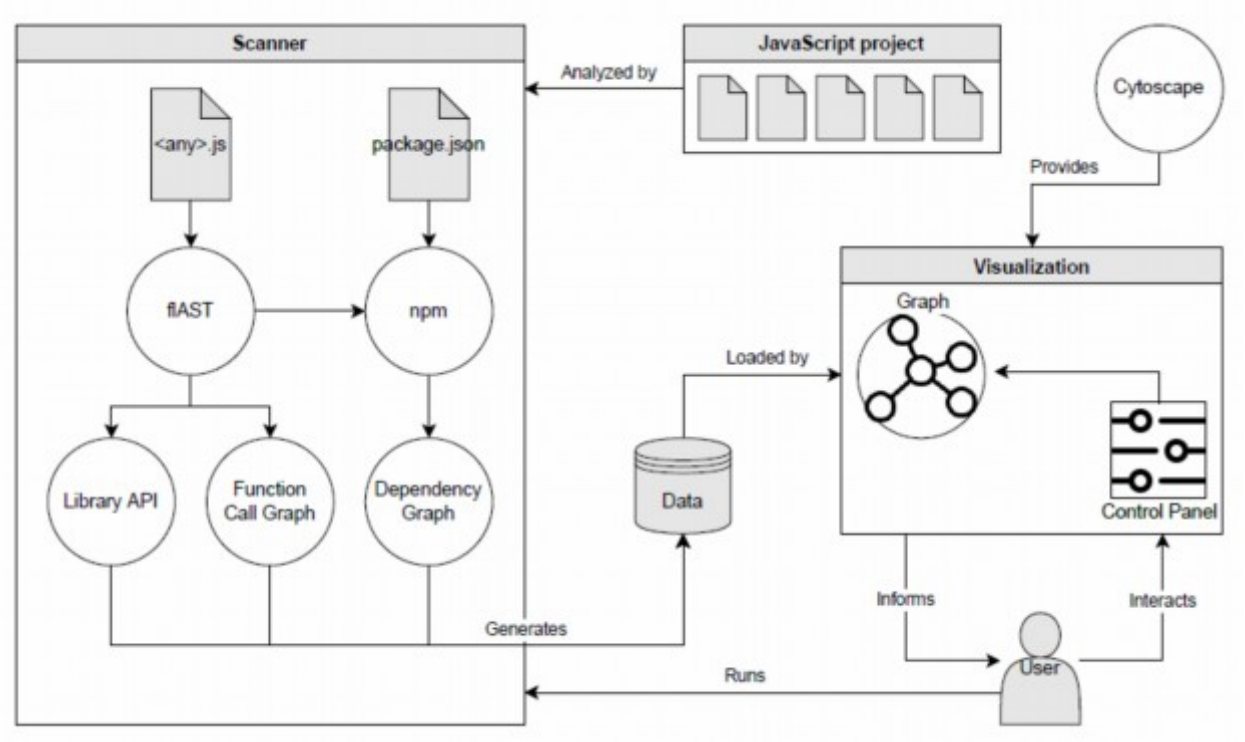


Рис. 3.3. Діаграма архітектури системи інтерактивної візуалізації

### 3.3. Принцип побудови абстрактного синтаксичного дерева

Для належного аналізу вихідного коду надзвичайно важливим є абстрактне синтаксичне дерево (AST), яке представляє собою деревовидне представлення вихідного коду. Кожен вузол дерева позначає конструкцію, що зустрічається у вихідному коді. AST є статичним представленням коду, що означає, що він не змінюється під час виконання.

Абстрактне синтаксичне дерево (AST) - це потужний інструмент, який використовується в комп'ютерних науках для представлення структури програми. Воно є ідеальним кандидатом для побудови візуалізацій, оскільки забезпечує глибоке розуміння внутрішньої логіки коду.

AST ідеально підходить для візуалізації з наступних причин:

- Точне відображення структури: AST точно відображає синтаксичну структуру коду, включаючи функції, змінні, умови, цикли та інші елементи. Це дозволяє створювати візуалізації, які точно відповідають коду.

- Абстрагування від синтаксису: AST абстрагується від конкретного синтаксису мови програмування, що робить його універсальним інструментом для візуалізації коду на різних мовах.

- Гнучкість: AST може бути модифіковано та розширено для підтримки різних видів візуалізацій, від простих діаграм до складних графічних інтерфейсів.

- Автоматизація: Процес побудови AST може бути автоматизований за допомогою парсерів, що дозволяє створювати візуалізації динамічно під час розробки.

Для роботи з AST використовують гаступні інструменти:

- Парсери: Для створення AST з коду (наприклад, ANTLR, Python's ast).

- Бібліотеки візуалізації: Для створення графіків та діаграм (D3.js, Graphviz).

- Інтегровані середовища розробки (IDE): Багато сучасних IDE підтримують візуалізацію коду на основі AST.

Це спосіб представлення структури програми у спосіб, який легко аналізувати. Для створення AST було обрано Acorn [20], оскільки це добре налагоджений парсер для JavaScript і все ще активно підтримується. Однією з ключових функцій, яка мені потрібна для належного аналізу, була здатність знаходити посилання на змінну в коді. Для цього було використано сторонню бібліотеку scope-analyser [21]. На жаль, під час розробки було виявлено, що scope-analyzer [21] був неточним, і це не буде виправлено найближчим часом. Приблизно в той же час я звернув увагу на новий багатообіцяючий проект, який пропонував такі ж функції, але з використанням іншого базового аналізатора AST. Незважаючи на те, що проект був досить пізнім, було вирішено переписати деякі частини кодової бази, щоб реалізувати цю нову

бібліотеку під назвою flast [22]. Це, по суті, оболонка для eslint-scope [23], яка виводить зведений об’єкт AST, який включає область і довідкову інформацію для кожного вузла. Він використовує сучасніший синтаксичний аналізатор, побудований на основі Acorn [20], який називається espree [24].

Лістинг 3.1. Фрагмент scanner.js, що показує, як файли та бібліотеки імпортуються через RequireJS

```
1  // Internal dependencies
2  const { constructString, constructTemplateLiteral, findReferences } =
       require("./common");
3  const { extractCalls } = require("./call-graph");
4  const { extractLibs } = require("./library-api");
5  const { getNodeModules } = require("./dependency-graph");
6
7  // External dependencies / libraries
8  const { basename, dirname, extname, relative, resolve } = require("
       path");
9  const { generateFlatAST } = require("flast");
10 const { promisify } = require("util");
11 const { readFile, stat } = require("fs/promises");
12 const glob = require("glob");
13 const randomColor = require("randomcolor");
```

## 3.4. Аналіз викликів функцій і використання бібліотек для побудови візуалізацій

Для аналізу необхідно відстежувати ланцюжки викликів функцій і використання API бібліотеки програмного забезпечення, а також дерево залежностей. Спочатку виявлення на GitHub і NPM проводилося для пошуку існуючих рішень. На жаль, багато з них були відсутні, застарілі, несумісні або просто не працювали. У попередньому розділі перераховано рішення, які в кінцевому підсумку були використані, але більшість завдань аналізу все одно довелося розробляти вручну. Давайте пройдемося по цій реалізації.

Під час сканування проекту він спочатку створює об’єкт дерева залежностей, або викликаючи власний метод NPM [25] для переліку всіх встановлених залежностей, або як запасний варіант (якщо ці залежності не

інстальовано локально), він відтворює цей об’єкт дерева залежностей, рекурсивно запитуючи веб-API для кожної батьківської залежності, який містить список її залежностей. Після цього він читає всі файли JavaScript проекту. Однак він ігноруватиме файли встановлених залежностей, розташовані в папці node_modules. Потім він створить AST для кожного прочитаного файлу. Цей AST використовується для отримання інформації про структуру коду та використовувані бібліотеки. Він генерує дерево викликів функцій у файлі, виявляючи будь-які виклики функцій і відстежуючи їх джерело та призначення/ціль, тобто визначення функції. Крім того, він намагається виявити всі імпортовані бібліотеки та надані ними функції/функції, тобто API бібліотеки. Вони обчислюються на основі вузлів декларації вимог і імпорту AST. Це виявить як оголошення, зроблені на початку файлів JavaScript, так і оголошення, вкладені у вихідний код. Нарешті, уся зібрана інформація перетворюється на об’єкт даних, який можна імпортувати за допомогою графіка, розташованого у звіті. Цей процес зображено на рисунку 3.3.

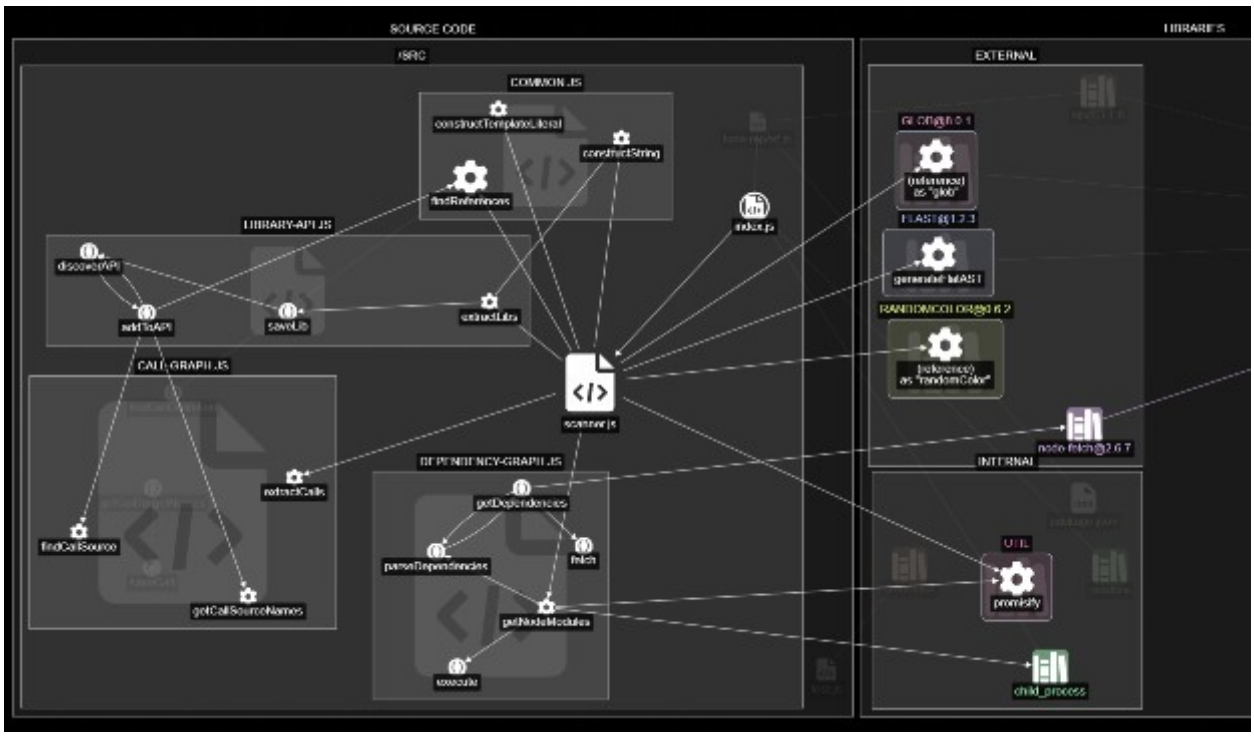


Рис. 3.4. Імпорт scanner.js, показаний у візуалізації

Як приклад, лістинг 3.1 - це фрагмент коду, показує його AST, і, нарешті, рисунок 3.4 відображає його візуалізацію з LUNA.

## 3.5. Методи формування звітів і графіків як способів візуалізації

Після завершення сканування проекту та створення даних, необхідних для графіка, LUNA створює звіт. Формат звіту – HTML, і його можна переглядати в будь-якому веб-переглядачі на основі Chromium, наприклад Google Chrome. Цей звіт також містить вбудовані зображення на основі SVG, сценарії JavaScript і таблиці стилів CSS. HTML звіту відображається за допомогою механізму шаблонів під назвою EJS [26]. Це рішення було вибрано для вбудовування всіх даних і коду у звіт. Це зробило створений звіт портативним, тож його можна було спільно використовувати між машинами (або навіть розмістити на статичному веб-сайті).

Основним способом візуалізації використання бібліотеки в проекті було обрано графік. Вузли можуть представляти різні елементи вихідного коду, такі як каталоги, файли, бібліотеки, класи та виклики функцій. Ребра використовуються для представлення використання або виклику функції, тобто стрілка від A до B, коли A використовує/викликає B. Давайте ближче розглянемо, як побудований цей графік і з яких компонентів він складається.

### *3.5.1. Використання бібліотеки cytoscape.js*

LUNA використовує бібліотеку під назвою cytoscape.js [27], яка є бібліотекою з відкритим кодом для повнофункціональних графіків, написаних на чистому JavaScript.

Cytoscape.js є потужною бібліотекою JavaScript, призначеною для створення інтерактивних візуалізацій графів. Вона надає широкий набір функцій для маніпуляції, стилізації та аналізу графів, що робить її незамінним інструментом для дослідників, розробників та аналітиків. Це потужна бібліотека, яка дозволяє розширювати та багато налаштовувати.

Крім того, він має гарну документацію, що полегшує роботу. Ця бібліотека використовується для створення графіка у звіті. Графік створюється з використанням даних, які генерує аналітична частина LUNA. Графік відображає 3 компоненти як підграфи: вихідний код, бібліотеки та графік залежностей. Кожен компонент візуалізує щось своє. Компоненти також пов'язані між собою.

### *3.5.2. Розширення*

Як згадувалося, cytoscape.js [27] підтримує розширення. LUNA використовує деякі з цих розширень. Одне з цих розширень використовується для додавання додаткових функцій до графіка, усі інші використовуються для макета графіка. Для підтримки згортання та розширення вузлів LUNA використовує розширення cytoscape-expand-collapse [28] для cytoscape.js. Це корисно для великих графіків, оскільки дозволяє отримати кращий огляд шляхом згортання нерелевантних частин. Таким чином, це дозволяє краще зосередитися на певній області графіка.

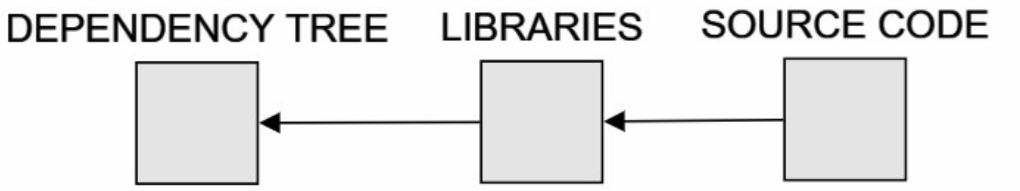


Рис. 3.5. Три компоненти візуалізації

### *3.5.3. Компонент вихідного коду*

Цей компонент візуалізує структуру програмного проекту. Він не тільки відображає структуру файлів і каталогів, що часто ретельно розроблено архітектором проекту, але також дозволяє розгортати файли в графи викликів функцій. Однак вузли, що представляють файли, за замовчуванням згорнуті. Крім того, розмір вузлів файлу визначається кількістю рядків коду у файлі. Це допомагає користувачам швидко визначати

найважливіші файли в проекті або принаймні файли з найбільшою кількістю коду.

### *3.5.4. Компонент бібліотек*

Цей компонент містить список усіх бібліотек, підключених до проекту програмного забезпечення. Бібліотечні вузли можуть бути розширені, щоб показати використаний ними API, тобто функції або об'єкти/класи, що містять методи. Вони поділяються на 3 групи. Одна група складається з усіх внутрішніх бібліотек, таких як нативні модулі, надані node.js, або бібліотеки внутрішньої розробки. Інша група містить список усіх зовнішніх бібліотек, які є сторонніми бібліотеками. Остання група включає бібліотеки, які безпосередньо не використовуються самою кодовою базою, а ймовірно використовуються лише для розробки проекту. Кожна бібліотека має свій власний рандомізований колір, тому легше відстежувати конкретну бібліотеку. Все, що підключено до цієї бібліотеки, має той самий колір.

### *3.5.5. Компонент графа залежностей*

'Цей компонент включає ланцюжок залежностей усіх бібліотек від компонента бібліотек. Оскільки ці залежності є вузлами, підключеними до вузла бібліотеки, вони мають однаковий колір. Весь ланцюжок залежностей має однаковий колір, якщо він не з'єднаний з іншим ланцюжком залежностей, оскільки одна залежність може входити до кількох ланцюжків залежностей. Цей компонент часто є великою складною мережею вузлів залежностей, тому, щоб не перевантажувати користувача, цей компонент графіка згорнуто за замовчуванням.

### *3.5.6. Оптимізація*

З додаванням графів викликів файлів і бібліотечних вузлів API під час розробки інструменту візуалізації кількість вузлів і ребер для завантаження в граф експоненціально зросла. У деяких великих проектах навіть стало

неможливо завантажити. Значить, потрібна була якась оптимізація. На щастя, значна частина вузлів прихована за згорнутими групами вузлів або підграфів, як обговорювалося в попередніх розділах про компоненти графа. Тому під час початкового завантаження графа LUNA ці вузли та ребра можна опустити. Завдання полягало в тому, щоб опустити правильні вузли та ребра та додати правильні вузли та ребра назад після події розширення їх батьківського вузла. Крім того, вузли, які з'єднуються з іншими вузлами за межами батьківського вузла, не можуть бути пропущені без порушення функціональності згортання. На щастя, ця техніка завантаження за вимогою була успішно реалізована та значно покращила час завантаження порівняно з неоптимізованою версією.

### 3.6. Представлення та опис алгоритмів компонування

Положення вузлів, які відображаються на графіку, обчислюється за допомогою алгоритму макета. У цьому розділі ми обговоримо алгоритми компонування, які були розглянуті в LUNA. Таблиця 3.1 показує короткий виклад різних макетів, а також їхні сильні та слабкі сторони на основі особистого досвіду роботи з макетами графів.

Таблиця 3.1.

Порівняння різних макетів графіків, які використовуються для візуалізації

| Layout | Edge crossing | Node overlap | Hierarchy & ranking | Grouping & clustering |
|---|---|---|---|---|
| Breadthfirst | 0 | − | + | 0 |
| Cola | − | − | − | + |
| Cose-bilkent | − | 0 | 0 | 0 |
| Elk: Layered | − | + | − | − |
| Elk: Mr. Tree | 0 | − | + | 0 |

Потужність алгоритму макета позначається знаком +, слабкість позначається знаком -, а 0 означає нейтральність.

Dagre — це дискретний макет, який розміщує вузли в ієрархічному порядку та мінімізує кількість перехресних посилань. Загальна реалізація натхненна «Технікою малювання орієнтованих графів» [29], а метод мінімізації кількості перехресних зв’язків базується на «Мінімізації двошарового прямолінійного перетину» [30].

Після спробування декількох макетів цей виявився найкращим, оскільки він створює чіткий і зручний макет. Тому цей макет спочатку був обраний для графіка. Пізніше в розробці в цьому алгоритмі макета було помічено кілька критичних помилок. Це призвело до виявлення того, що основна бібліотека, на яку він спирався, застаріла. Тому було вирішено відмовитися від підтримки цього алгоритму макета.

Натомість було вирішено підтримувати набір алгоритмів компонування. Цю колекцію було підібрано та налаштовано на найефективнішу з усіх підтримуваних макетів на графіку. Однак жоден із них не був ідеальним для всіх варіантів використання, тому в інструмент було додано можливість переключатися між макетами в цій колекції. Таким чином користувач має можливість використовувати найкращий макет для свого випадку використання або переваг.

Макет «Breadthfirst» підтримується cytoscape.js. Він поміщає вузли в ієрархію на основі обходу графа в ширину. Цей алгоритм макета найкраще підходить для графіків дерева/лісу в режимі зверху вниз за замовчуванням і для DAG у режимі кола.

Алгоритм побудови графіків "Breadthfirst" (алгоритм обходу в ширину) застосовується для візуалізації графів таким чином, щоб рівні вузлів відображалися відповідно до їх віддаленості від кореневого вузла. Це особливо корисно для візуалізації ієрархічних структур або деревоподібних графів.

Основні особливості алгоритму "Breadthfirst":

- Вузли, які ближче до кореня, розташовуються на верхніх рівнях, а більш віддалені — на нижніх (або навпаки, залежно від напрямку побудови).

- Алгоритм добре підходить для візуалізації дерев, ієрархій або слабко зв'язаних графів, де вузли розташовані рівномірно за рівнями.

- Під час обходу в ширину не виникає перекриття вузлів одного рівня, що забезпечує структуровану й зрозумілу візуалізацію.

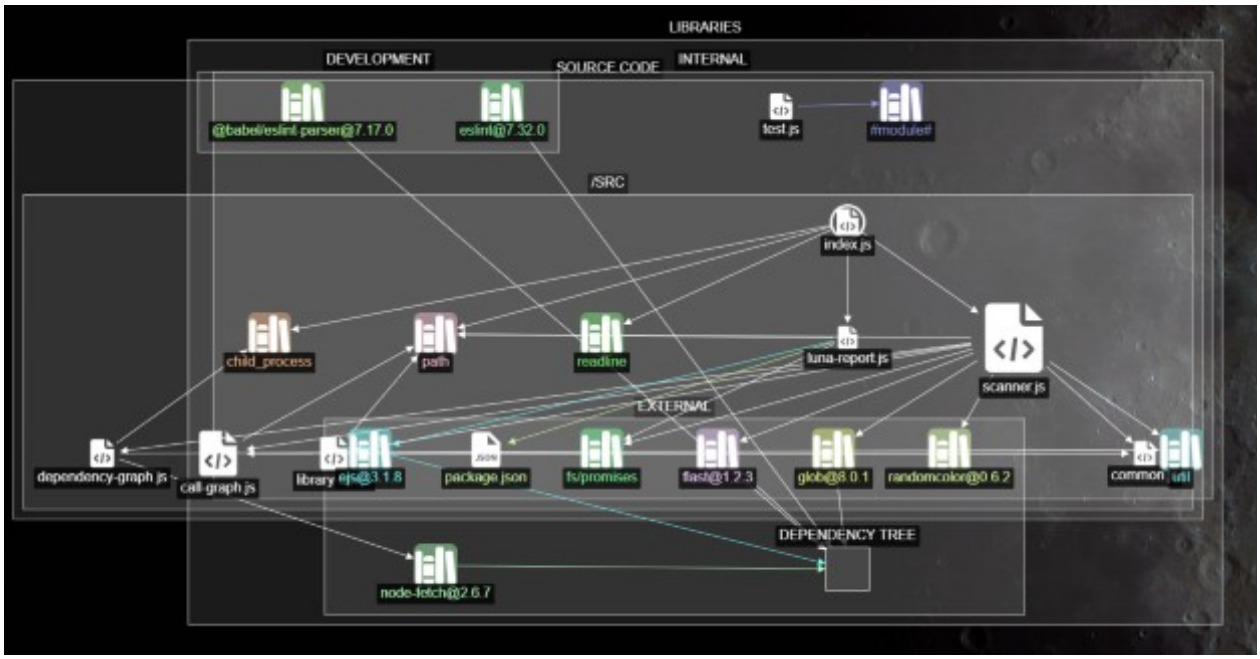


Рис. 3.6. Алгоритм макета для графіка у звіті, створеному згідно алгоритму «Breadthfirst»

Макет Cola для cytoscape.js використовує симуляцію фізики, спрямованої силою, з декількома складними обмеженнями.

Cola - це фізичний симуляційний макет для Cytoscape.js, який використовує сили тяжіння та відштовхування для розміщення вузлів і ребер графа в естетично привабливому та інформативному вигляді. Цей макет особливо корисний для візуалізації складних мереж, де автоматичне розподілення вузлів може значно покращити читабельність графа.

Макет Cola базується на фізичних принципах, таких як:

- Сили тяжіння: Притягують вузли один до одного, створюючи кластери.

- Сили відштовхування: Відштовхують вузли один від одного, запобігаючи перекриттю.

- Константи пружини: Визначають жорсткість зв'язків між вузлами.

- Ітерації: Макет виконує кілька ітерацій, щоб досягти стабільного стану.

Основні параметри макета Cola:

- nodeSpacing: Мінімальна відстань між вузлами.
- edgeLength: Бажана довжина ребер.
- avoidOverlap: Чи слід уникати перекриття вузлів.
- handleDisconnected: Як обробляти незв'язані вузли.
- unconstr: Чи дозволити незв'язаним вузлам рухатися вільно.
- gravity: Сила тяжіння, що притягує вузли до центру графа.
- alignment: Вирівнювання вузлів (горизонтальне, вертикальне).

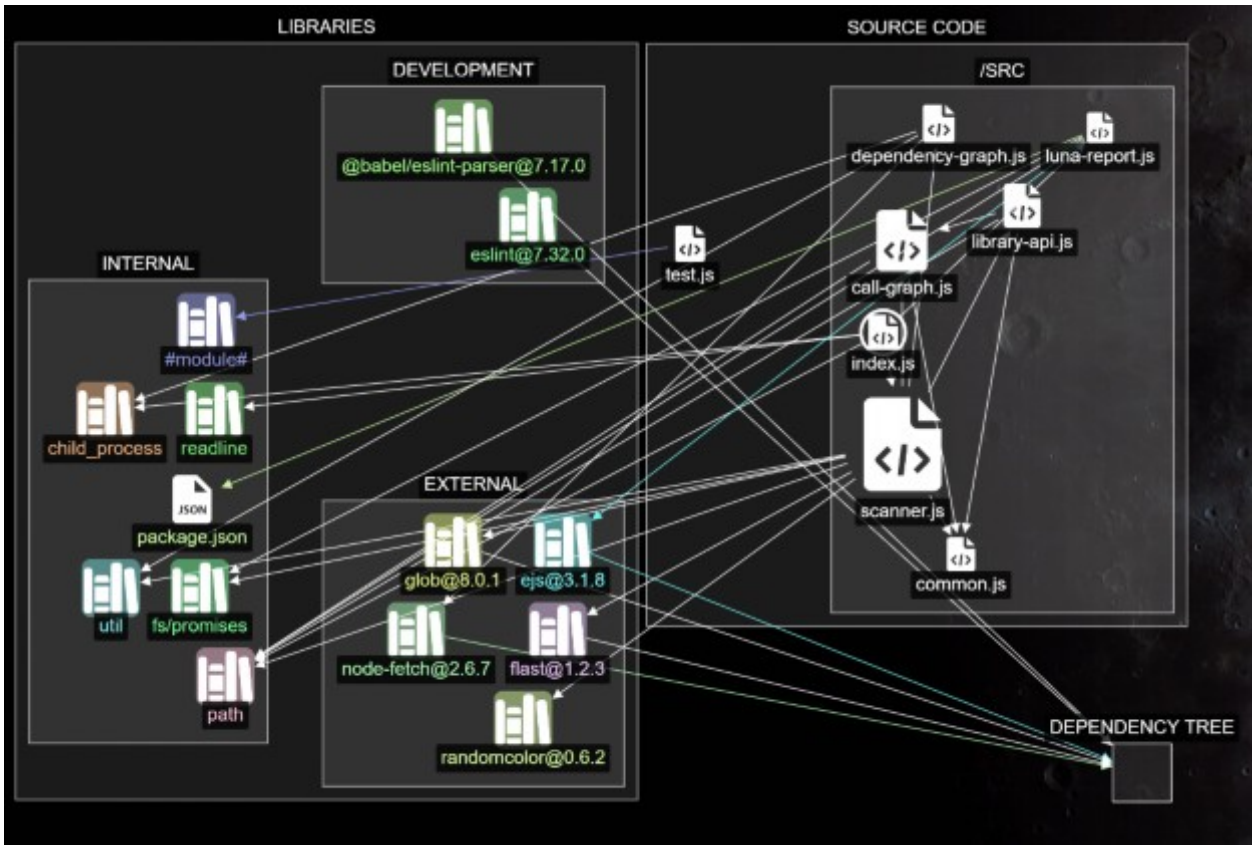


Рис. 3.7. Алгоритм макета для графіка у звіті, створеному на основі Cola

Cose-Bilkent — це пружинний макет для вбудовування для cytoscape.js із підтримкою складених графів (вкладених структур) і змінних (нерівномірних) розмірів вузлів.

Cose-Bilkent - це популярний макет для Cytoscape.js, який використовує фізичний симуляційний підхід для розміщення вузлів і ребер графа. Він відрізняється від макета Cola деякими особливостями, що можуть бути корисними в певних ситуаціях.

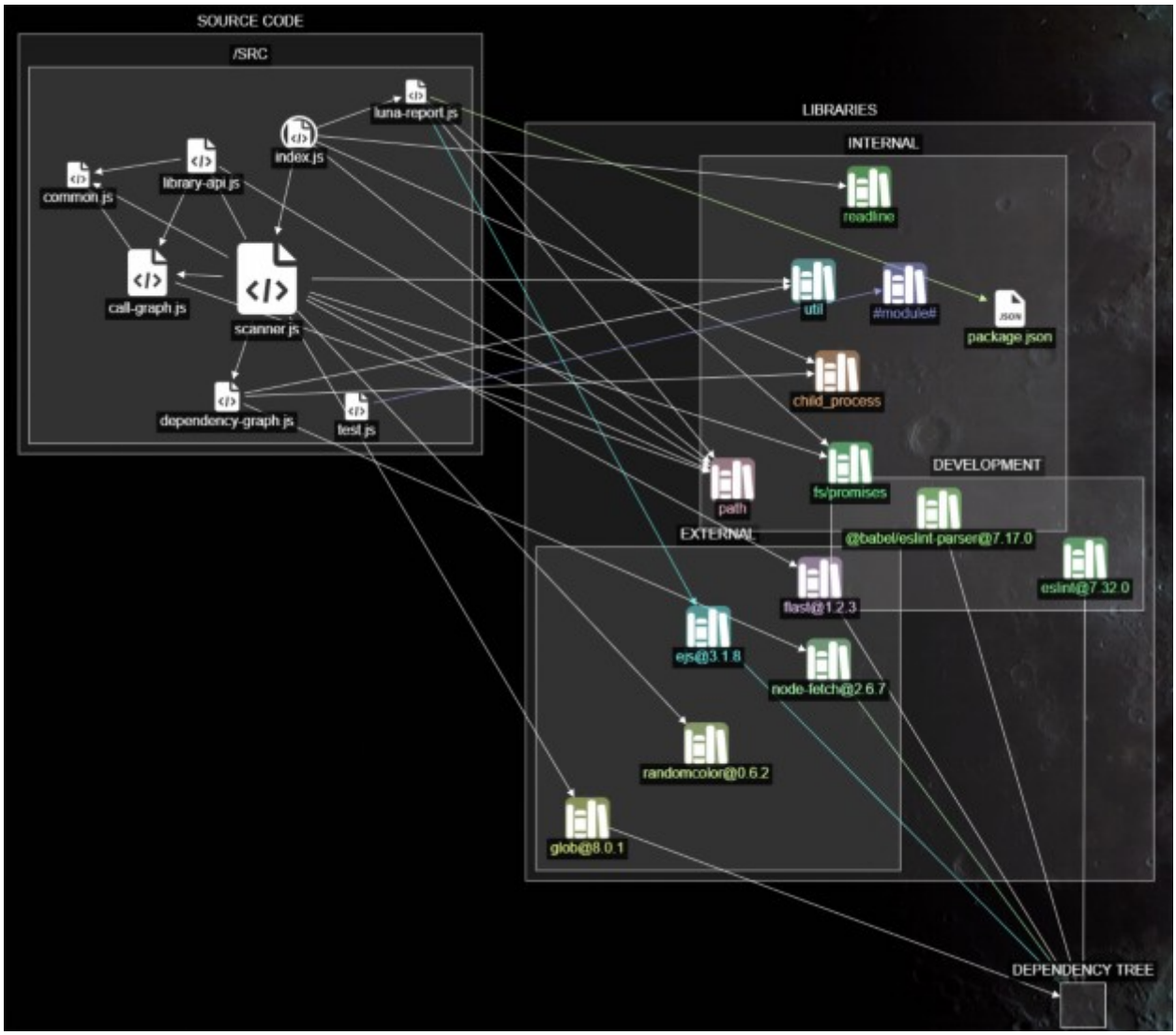


Рис. 3.8. Алгоритм макета для графіка у звіті, створеному на основі Cose-Bilkent

Наведемо основні відмінності Cose-Bilkent від Cola:

- Метод розміщення: Cose-Bilkent використовує метод "concentric layout", який розміщує вузли в концентричних колах, що може бути корисним для візуалізації ієрархічних структур.

- Оптимізація: Cose-Bilkent використовує оптимізаційні алгоритми для покращення якості розміщення вузлів.

- Параметри: Cose-Bilkent має деякі відмінні параметри, такі як nodeRepulsion, idealEdgeLength та gravity, які можна налаштувати для отримання бажаного результату.

Переваги використання макета Cose-Bilkent:

- Ієрархічна структура: Підходить для візуалізації ієрархічних даних.

- Оптимізація розміщення: Забезпечує більш оптимальне розміщення вузлів.

- Гнучкість: Може бути налаштований за допомогою різних параметрів.

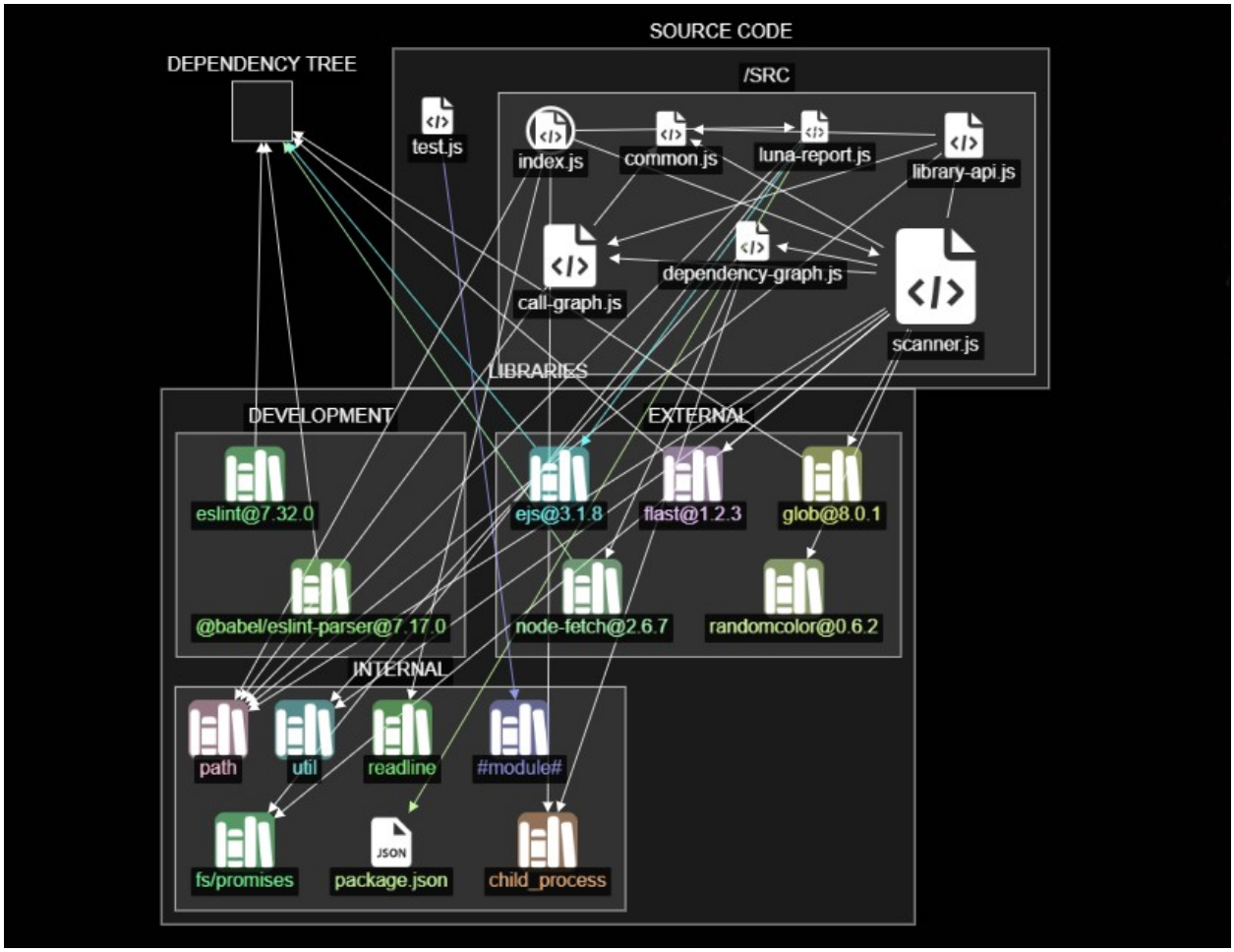


Рис. 3.9. Алгоритм макета для графіка у звіті, створеному на основі Eclipse Layout Kernel (ELK) Layered

ELK (Eclipse Layout Kernel) - це потужний набір алгоритмів компонування графів, інтегрований в Cytoscape.js. Він пропонує широкий спектр можливостей для створення високоякісних і інформативних візуалізацій складних мереж. На відміну від фізично орієнтованих макетів, таких як Cola і Cose-Bilkent, ELK використовує більш формальний підхід, що дозволяє досягти більш точного і контрольованого компонування.

Переваги використання макета ELK:

- Гнучкість: Широкий спектр алгоритмів і параметрів для налаштування.

- Якість компонування: Висока якість візуалізації навіть для складних графів.

- Контроль: Можливість детального контролю над процесом компонування.

- Підтримка різних типів графів: Ефективна робота з різними типами графів.

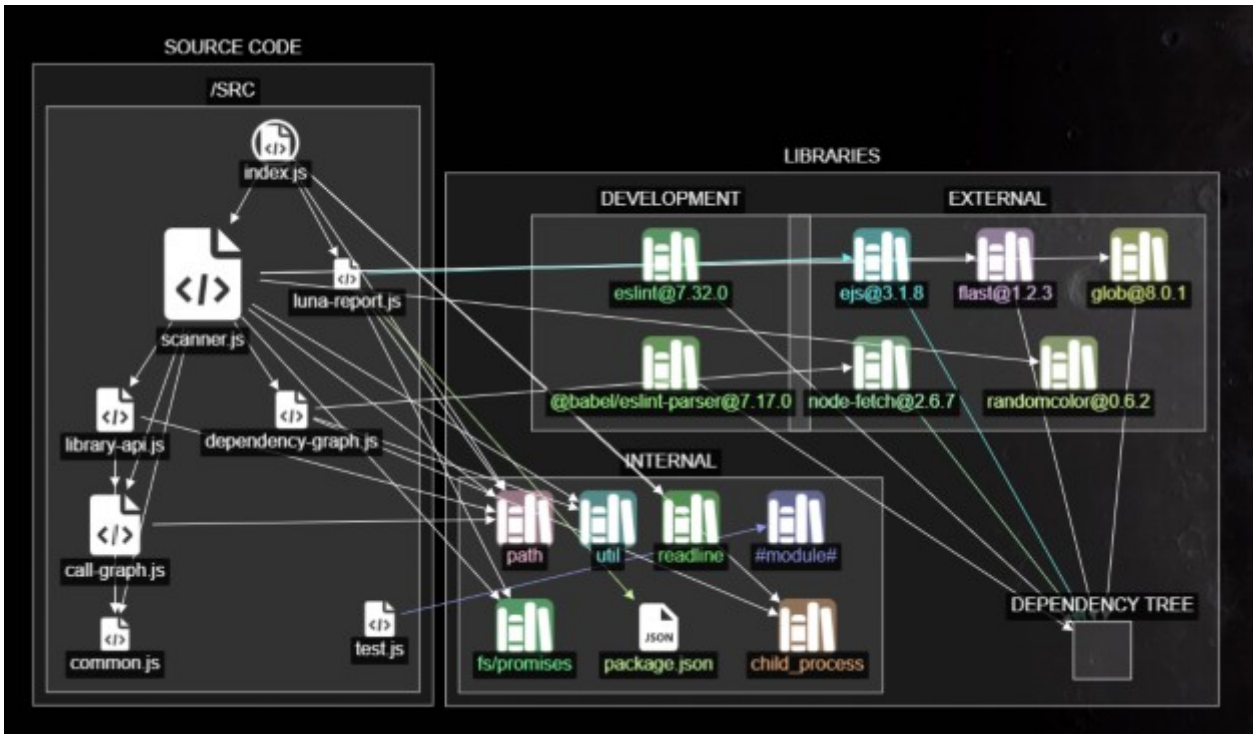


Рис. 3.10. Алгоритм макета для графіка у звіті, створеному на основі Eclipse Layout Kernel (ELK) Mr. Tree

## 3.7. Інтерфейс інструменту інтерактивної візуалізації програмного забезпечення на основі JavaScript

Інтерфейс складається з двох частин: сканера та візуалізації. Сканер пропонує інтерфейс командного рядка (CLI), що означає, що ви можете взаємодіяти з ним лише через термінал або консоль. Він виконує лише одне: сканує проект і створює звіт про нього за допомогою команди «npx luna-scanner». Інтерфейс візуалізації значно більш функціональний. Крім графіків, які описуються в попередньому розділі, основний спосіб взаємодії з візуалізацією — це два меню. Давайте поглянемо на них.

Меню графіків містить параметри взаємодії з усім графіком. На рисунку 3.11 показано меню графіка та пояснення доступних у ньому параметрів.

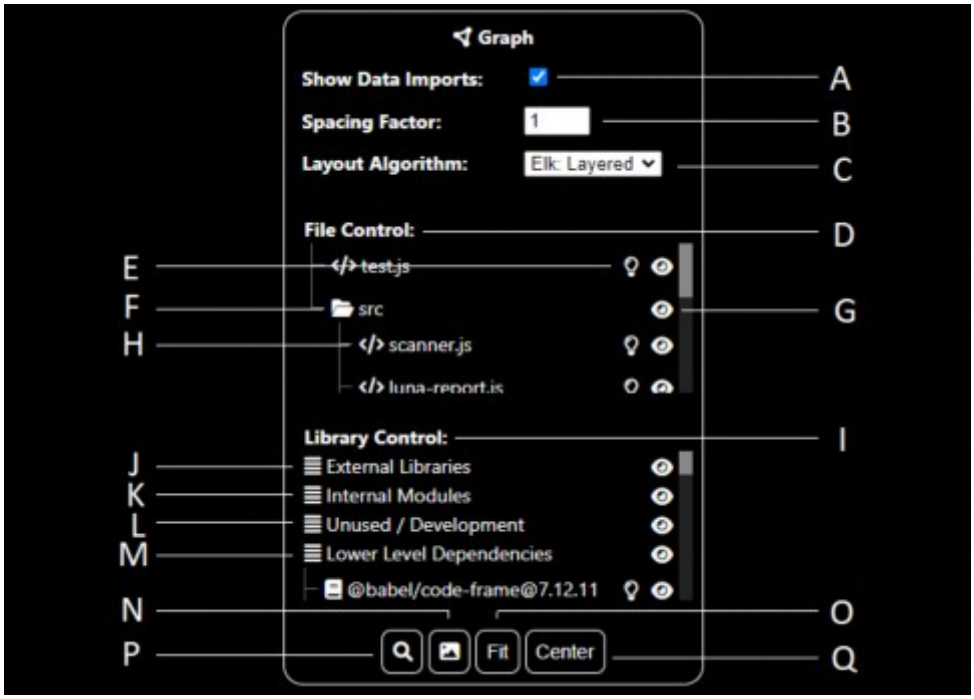


Рис. 3.11. Меню графіків

На рисунку 3.11 показано наступні елементи меню:

A) Перемикання видимості всіх файлів даних JSON, які імпортуються;

B) Зміна відстані між вузлами через цей коефіцієнт інтервалу;

C) Зміна положення вузлів на графі, продиктовані вибраним алгоритмом макета;

D) Розділ для керування всіма вузлами файлів, які існують у графі;

E) Перемикання  підсвічування вузла на графіку;

F) Папка, яку можна згорнути (подвійним клацанням) у розділі керування файлами;

G) Перемикання видимості вузла на графіку;

H) Файл у розділі керування файлами;

I) Розділ для керування всіма бібліотеками та вузлами залежностей, які існують у графі;

J) Група зовнішніх бібліотек;

K) Група внутрішніх бібліотек;

Л) Група розвитку бібліотек;

M) Група графів залежностей;

N) Зробити скріншот графіка, який знаходиться у вікні перегляду;

O) Розмістити весь графік у вікні перегляду;

P) Пошук вузла в графі за міткою (незалежно від регістру);

Q) Розміщення положення вікна перегляду до центру.

Меню вузла показує інформацію про вибраний вузол і містить параметри для керування ним. Рисунки 3.12 і 3.13 показує меню вузла та пояснюють доступні в ньому параметри.

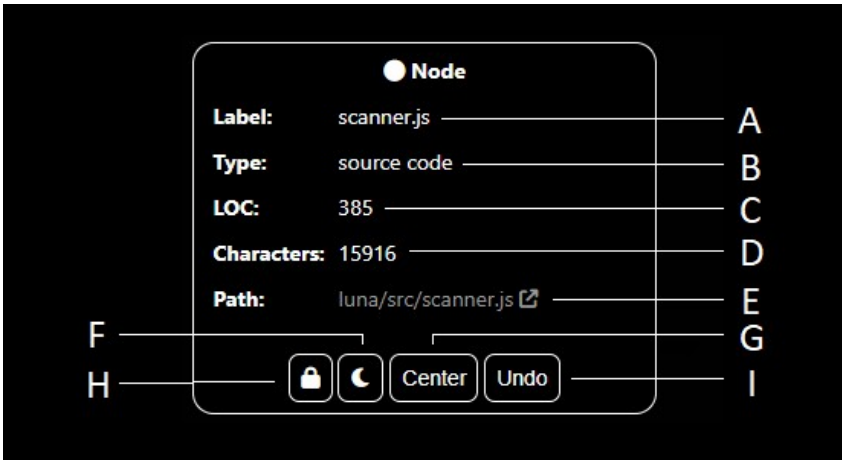


Рис. 3.12. Меню вузла файлу

На рисунку 3.12 представлено наступні елементи меню:

A) Мітка вузла;

B) Категорія, яку представляє цей вузол (API, клас, файл, папка, виклик функції, дані JSON або бібліотека/залежність);

C) Загальна кількість рядків коду в цьому файлі;

D) Загальна кількість символів коду в цьому файлі;

E) Абсолютний шлях до цього файлу;

F) Не використовується;

Ж) Фокус на вузлі;

H) Перемикання функції наведення миші;

I) Увімкнення підсвічування вузла.

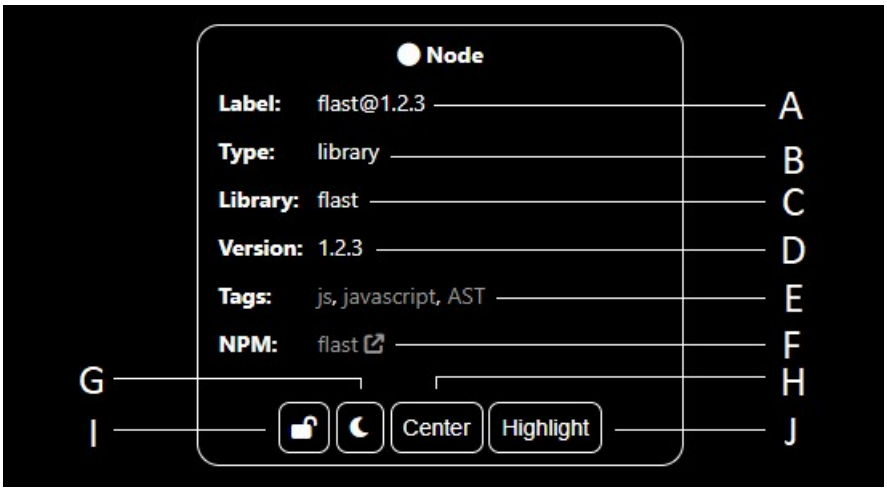


Рис. 3.13. Меню вузла бібліотеки.

На рисунку 3.13 показано елементи меню:

A) Мітка вузла;

B) Категорія, яку представляє цей вузол (API, клас, файл, папка, виклик функції, дані JSON або бібліотека/залежність);

B) Назва бібліотеки;

Г) Версія бібліотеки;

E) Ключові слова, пов’язані з бібліотекою;

F) Посилання на список бібліотек на npmjs.com;

G) Кнопка, яка показує користувачеві команду bash для створення звіту з цієї бібліотеки;

H) Фокус на вузлі;

I) Перемикання функції наведення миші;

J) Увімкнення підсвічування вузла.

### 3.8. Тестування інструменту інтерактивної візуалізації

Під час розробки були проведені тести для забезпечення належної функціональності. Було додано режим налагодження, щоб надати додаткові журнали, а також увімкнути обмежену функцію гарячого перезавантаження для створеного звіту, що дозволило застосувати зміни до коду клієнта та стилю після перезавантаження звіту без його повторного створення.

Зазвичай звіт має весь код і дані, вбудовані в один статичний файл. Натомість у режимі налагодження певні сценарії та стилі не вбудовуються, а пов’язуються з файлами розробки. Більшість тестів було виконано на самому проекті, але щоб забезпечити хорошу продуктивність в усіх проектах JavaScript, у набір тестів було включено кілька додаткових проектів. Ці проекти були свідомо обрані, щоб охопити широкий спектр простору розробки JavaScript.

Наприклад, одним із проектів, включених до набору тестів, була популярна бібліотека jQuery [31], яка має багато внутрішніх бібліотек, але не має зовнішніх бібліотек як залежностей. Іншим проектом, який часто використовувався як стрес-тест, був вихідний код для програми CLI NPM [25], яка включала понад п’ять тисяч файлів JavaScript і 2500 папок під час тестування.

Наразі інструмент здатен впоратися з цими великими проектами, хоча з неабиякими труднощами та тривалим часом завантаження.

Для тестування інструменту інтерактивної візуалізації було виконано наступні завдання:

**Task 1**

Використовуючи інструмент візуалізації треба знайти бібліотеку або бібліотеки, що використовуються, уражені вразливістю безпеки в залежності «estraverse@5.2.0».

**Task 2**

Використовуючи інструмент візуалізації треба щоб визначити, які файли та функції у вихідному коді потрібно змінити під час видалення або заміни бібліотеки flast.

**Task 3**

Використовуючи інструмент візуалізації треба знайти, які 5 функцій API внутрішнього «шляху» бібліотеки використовуються внутрішнім кодом.

**Task 4**

Використовуючи інструмент візуалізації треба визначити топологічний порядок, у якому 7 файлів js у папці «src» використовують один одного, використовуючи відповідний макет, починаючи з src/index.js.

**Task 5**

Використовуючи інструмент візуалізації треба показати хороше зображення лише внутрішньої роботи src/call-graph.js із вихідного коду.

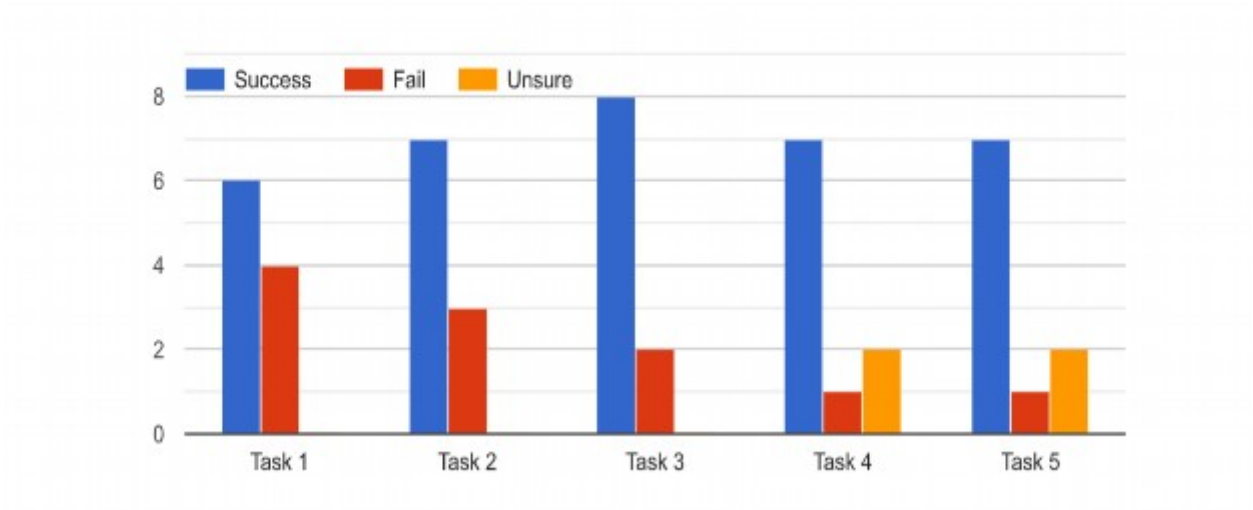


Рис. 3.14. Графік, що показує чи вдалось виконати кожне із завдань

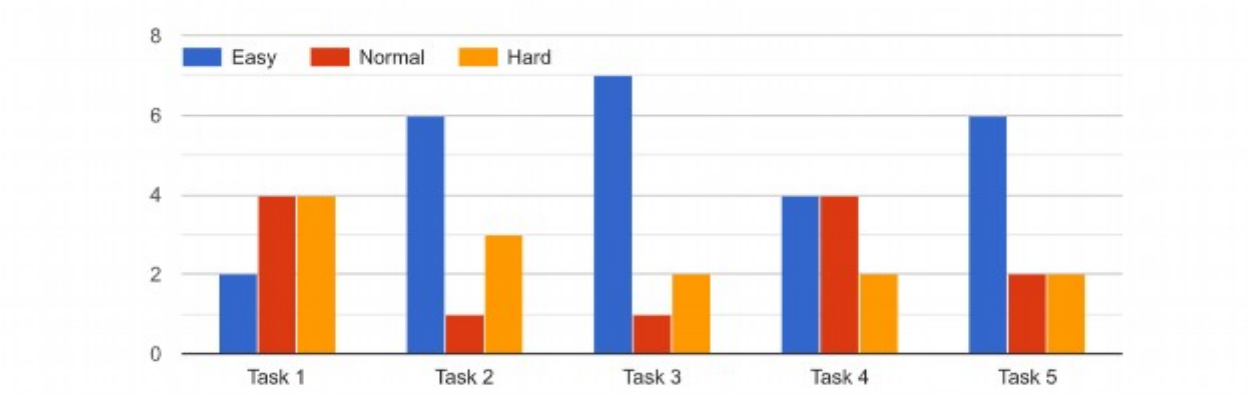


Рис. 3.15. Графік, що показує складність кожного із завдань

Щодо питань які були подані в першому розділі, то можна вказати наступне. Як відновити архітектуру програмного забезпечення ? Це було реалізовано в даному інструменті, який може робити наступне для відновлення архітектури програмного забезпечення:

- Відновити області вихідного коду.
- Відновити класи та їхні методи.
- Відновити виклики функцій та їх визначення.
- Відновити каталог і структуру файлів.
- Відновити бібліотеки, використані у вихідному коді.
- Відновити імпортований API з бібліотек.
- Відновити відношення залежності між бібліотеками.
- Відновити відношення залежності між файлами.

Як пояснюється в третьому розділі, метод відновлення архітектури програмного забезпечення базується на аналізі вихідного коду. Абстрактне синтаксичне дерево (AST) було побудовано з вихідного коду для вилучення вищевказаної інформації шляхом відвідування кожного вузла з AST та аналізу самого себе, його обсягу та посилань.

Завдання 1 було додано для перевірки зв’язків залежності бібліотеки в архітектурі. Він по суті просив користувача визначити всі підключені залежності для певної бібліотеки. Лише 6 із 10 спроб були успішними, оскільки вони вважали його відносно складним. Основною проблемою цього

завдання було обмеження, де функція пошуку не враховувала згорнуті вузли. Отже, коли вузол згорнуто, він не шукатиме вузли в цьому вузлі. Через це важко знайти потрібну бібліотеку.

Завдання 4 стосується структури архітектури та просить вибрати певний алгоритм компонування, щоб розташувати архітектурні елементи, тобто вузли на графі, таким чином, щоб топологічний порядок став зрозумілим. В трьох випадках не вдалося цього зробити, оскільки не змогли повністю використати інструмент через технічні проблеми, а один із них не знав, що робити. Це завдання було визначено як середньої складності.

Останнє завдання, завдання 5, пропонує користувачам створити образ внутрішньої архітектури певного компонента в рамках проекту, це включає виклики функцій та їх визначення, а також класи та їхні методи. Це не тільки тестує користувачів на використання функції знімка екрана, але також перевіряє їх на визначення правильної архітектури та приховування всіх невідповідних частин. Знову сім спроб були успішними, а дві ні. Проте це завдання вважалося досить простим.

Отже, ми можемо вважати, що даний інструмент успішно здатен відновити архітектуру програмного забезпечення.

Друге питання дослідження полягає в тому, як виявити бібліотеки та їх використання? У відповіді на попереднє дослідницьке запитання уже описано, як відновити архітектуру програмного забезпечення, яка включає бібліотеки, що використовуються у вихідному коді. Це робиться шляхом перегляду метаданих проекту, а також будь-якої бібліотеки, імпортованої у вихідний код. У першій відповіді також зазначено, що він відновлює API, імпортований з бібліотек, що означає будь-які функції, об’єкти чи дані, які бібліотека пропонує у вихідний код. Крім того, аналіз AST також відстежує, де цей API використовується у вихідному коді. Це означає, що даний інструмент може виявити бібліотеки та їхнє використання.

Запитання дослідження № 3 стосується, як візуалізувати інформацію корисним способом? Звіт, який генерує інструмент візуалізації в даній роботі, містить візуалізацію всієї зібраної інформації.

Відповідаючи на попередні запитання дослідження, ми можемо відповісти на основне питання дослідження. Основне питання дослідження: як полегшити розуміння використання бібліотеки в архітектурі програмного забезпечення ?

Відповіддю на це запитання було використано інструмент візуалізації, що описано в даній роботі. Цей інструмент можна використовувати для відновлення архітектури програмного забезпечення, виявлення бібліотек і їх використання, а також візуалізації інформації в корисний спосіб, як ми бачили з наших попередніх питань. Візуалізація була обрана як найкращий спосіб передачі інформації про використання бібліотеки в програмному забезпеченні.

## Висновки до розділу

У третьому розділі було представлено методологію інтерактивної візуалізації структури програмного забезпечення, що базується на JavaScript. Першочергово було описано підхід до розробки рішення та створення архітектури системи, що включає ключові компоненти для ефективної візуалізації. Детально розглянуто побудову діаграми архітектури, принципи формування абстрактного синтаксичного дерева, що забезпечує зрозуміле відображення програмних структур.

Важливою складовою є аналіз викликів функцій та бібліотек, який слугує основою для побудови візуалізацій. Основна увага приділена методам формування звітів та графіків, зокрема через використання бібліотеки cytoscape.js, яка дозволяє створювати інтерактивні графи залежностей. Було розроблено й описано кілька компонентів для роботи з вихідним кодом,

бібліотеками та залежностями, а також здійснено оптимізацію їх функціонування.

Окрему увагу приділено алгоритмам компонування для побудови візуалізацій, що забезпечують високу продуктивність та якість розташування елементів. В результаті, створено інтерфейс інструменту інтерактивної візуалізації, який був протестований на практиці для підтвердження його ефективності. Тестування показало, що розроблений інструмент забезпечує високий рівень інтерактивності та дозволяє користувачам гнучко працювати з графічними представленнями програмних структур.

# ВИСНОВКИ

У магістерській роботі проведено комплексне дослідження методів інтерактивної візуалізації структури програмного забезпечення на основі JavaScript. Метою роботи є ілюстрація використання програмних бібліотек та архітектурних елементів у програмних системах, що допомагає розробникам краще розуміти архітектуру проектів. Для досягнення цієї мети було досліджено процес інтерактивної візуалізації шляхом сканування проектів JavaScript, створення абстрактного синтаксичного дерева (AST) з вихідного коду та вилучення з нього інформації про архітектуру програмного забезпечення та використання бібліотек. Зібрані дані візуалізуються у вигляді веб-звітів, які забезпечують інтерактивний перегляд структур проекту.

Інструмент візуалізації складається з трьох основних підграфів та панелі взаємодії. Перший підграф представляє структуру каталогу та файлів проекту, кожен з яких містить граф викликів функцій. Другий підграф демонструє використання бібліотек, поділяючи їх на внутрішні, зовнішні та невикористовувані. Третій підграф відображає залежності між бібліотеками. Відношення між елементами проекту представлені як у межах окремих підграфів, так і між ними. Панель взаємодії дозволяє користувачам інтерактивно взаємодіяти з візуалізаціями, що забезпечує зручний інструмент для аналізу структури програмного забезпечення.

У першому розділі роботи досліджено та проаналізовано предметну область візуалізації програмного забезпечення. Було розглянуто поняття архітектури програмного забезпечення, бібліотек та візуалізацій. Окрему увагу приділено огляду літератури, що дозволило систематизувати сучасні підходи та інструменти, що використовуються у цій сфері.

Другий розділ присвячено дослідженню методів, алгоритмів та інструментів візуалізації структури програмного забезпечення. Було проведено аналіз таких інструментів, як CodeGraph, Hunter, та Eunice, що дозволило виявити їхні переваги та обмеження у контексті роботи з

проектами на JavaScript. Окрім того, вивчено алгоритми побудови візуалізацій, зокрема Breadthfirst, Cola, Cose-Bilkent та алгоритми з набору Eclipse Layout Kernel, які забезпечують ефективне компонування графів різної складності.

У третьому розділі детально представлено методологію інтерактивної візуалізації. Було описано розробку структури рішення, створення архітектурної діаграми системи, принципи побудови абстрактного синтаксичного дерева, а також методи аналізу викликів функцій та використання бібліотек. Крім того, було розроблено різні компоненти системи, такі як компонент вихідного коду, компонент бібліотек та компонент графа залежностей, що дозволяє створювати комплексні візуалізації. Описано алгоритми компонування та тестування розробленого інструменту, що підтвердило його ефективність у відновленні архітектури програмного забезпечення та візуалізації залежностей.

Отже, результати роботи продемонстрували здатність інтерактивної візуалізації ефективно відображати архітектуру програмного забезпечення, виявляти бібліотеки та їх використання. Інструмент візуалізації дозволяє корисно відобразити інформацію для подальшого аналізу, сприяючи покращенню розуміння архітектурних структур програмного забезпечення.

# ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ


1. Wikipedia contributors, “Software package — Wikipedia, the free encyclopedia.” https://en.wikipedia.org/w/index.php?title=Software package &oldid=1104525135, 2022.
2. Wikipedia contributors, “Software architecture — Wikipedia, the free encyclopedia.” https://en.wikipedia.org/w/index.php?title=Software architecture&oldid=1115479169
3. L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice. SEI Series in Software Engineering, Boston, MA: Addison Wesley, 4 ed., Oct. 2021.
4. R. Agarwal, R. Deshmukh, P. Borhade, S. Murarka, and D. Datta, “Software architecture recovery techniques,” International Journal of Engineering and Advanced Technology, vol. 9, p. 4, 04 2020.
5. J. Garcia, I. Ivkovic, and N. Medvidovic, “A comparative analysis of software architecture recovery techniques,” in 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 486–496, IEEE, 2013.
6. R. M. Tarawaneh, P. Keller, and A. Ebert, “A General Introduction To Graph Visualization Techniques,” in Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering - Proceedings of IRTG 1131 Workshop 2011 (C. Garth, A. Middel, and H. Hagen, eds.), vol. 27 of OpenAccess Series in Informatics (OASIcs), (Dagstuhl, Germany), pp. 151–164, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012.
7. R. G. Kula, A. Ouni, D. M. German, and K. Inoue, “On the impact of micro-packages: An empirical study of the npm javascript ecosystem,” arXiv preprint arXiv:1709.04638, 2017.
8. A. Zerouali, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, “On the diversity of software package popularity metrics: An empirical study of

npm," in 2019 IEEE 26th international conference on software analysis, Evolution and Reengineering (SANER), pp. 589–593, IEEE, 2019.

9. J. ˇZitn`y, "Npm a javascript," Master's thesis, ˇCesk´e vysok´e uˇcen´ı technick´e v Praze. Vypoˇcetn´ı a informaˇcn´ı centrum., 2017.
10. B. Taraghi, A. Azmi, and O. Yusop, "Producing software engineering documents through software reverse engineering for node.js web application," in Advanced Research in Engineering and Information Technology International Conference (AVAREIT), 2018.
11. A. Nurwidyantoro, T. Ho-Quang, and M. R. V. Chaudron, "Automated classification of class role-stereotypes via machine learning," in Proceedings of the Evaluation and Assessment on Software Engineering, EASE '19, (New York, NY, USA), p. 79–88, Association for Computing Machinery, 2019.
12. L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 138–147, IEEE, 2017.
13. L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, "Breaking bad? semantic versioning and impact of breaking changes in maven central," arXiv preprint arXiv:2110.07889, 2021.
14. H. Mili, F. Mili, and A. Mili, "Reusing software: Issues and research directions," IEEE transactions on Software Engineering, vol. 21, no. 6, pp. 528–562, 1995.
15. W. B. Frakes and P. Gandel, "Representing reusable software," Information and Software Technology, vol. 32, no. 10, pp. 653–664, 1990.
16. W. B. Frakes and T. P. Pole, "An empirical study of representation methods for reusable software components," IEEE transactions on software engineering, vol. 20, no. 8, pp. 617– 630, 1994.
17. A. Feldthaus, M. Sch¨afer, M. Sridharan, J. Dolby, and F. Tip, "Efficient construction of approximate call graphs for javascript ide services," in 2013

35th International Conference on Software Engineering (ICSE), pp. 752–761, IEEE, 2013.

18. B. B. Nielsen, M. T. Torp, and A. Møller, “Modular call graph construction for security scanning of node. js applications,” in Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 29–41, 2021.
19. PerimeterX, “flAST - FLat Abstract Syntax Tree.” https://github.com/PerimeterX/flast, 2022.
20. npm, “npm - a JavaScript package manager.” https://github.com/npm/cli, 2010.
21. M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader, “Cytoscape.js: a graph theory library for visualisation and analysis,” Bioinformatics, vol. 32, pp. 309–311, 2015.
22. U. Dogrusoz, A. Karacelik, I. Safarli, H. Balci, L. Dervishi, and M. C. Siper, “Efficient methods and readily customizable libraries for managing complexity of large networks,” PloS one, vol. 13, no. 5, p. e0197238, 2018.
23. E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo, “A technique for drawing directed graphs,” IEEE Transactions on Software Engineering, vol. 19, no. 3, pp. 214–230, 1993.
24. M. J¨unger and P. Mutzel, “2-layer straightline crossing minimization: Performance of exact and heuristic algorithms,” in Graph Algorithms And Applications I, pp. 3–27, World Scientific, 2002.
25. R. Van Barlingen, “CodeGraph: an Interactive Dependency Analyzer for JavaScript Projects,” master’s thesis, Aalto University. School of Science, 2021.
26. M. Dias, D. Orellana, S. Vidal, L. Merino, and A. Bergel, “Evaluating a visual approach for understanding javascript source code,” in Proceedings of the 28th International Conference on Program Comprehension, pp. 128–138, 07 2020.

27. Bostock, M., Ogievetsky, V., & Heer, J. (2011). "D3: Data-Driven Documents". IEEE Transactions on Visualization and Computer Graphics, 17(12), 2301–2309.
28. Murray, S. (2013). Interactive Data Visualization for the Web. O'Reilly Media.
29. McCandless, D. (2010). The Visual Miscellaneum: A Colorful Guide to the World's Most Consequential Facts. Harper Design.
30. Tufte, E. R. (2006). Beautiful Evidence. Graphics Press.
31. Heer, J., & Shneiderman, B. (2012). "Interactive Dynamics for Visual Analysis". Communications of the ACM, 55(4), 45–54.
32. Wittenburg, K., & Shaw, D. (2020). Introduction to Data Visualization and Storytelling. Springer.
33. Roberts, J. C. (2007). "State of the Art: Coordinated & Multiple Views in Exploratory Visualization". Fifth International Conference on Coordinated and Multiple Views in Exploratory Visualization, 61–71.
34. Aigner, W., Miksch, S., Schumann, H., & Tominski, C. (2011). Visualization of Time-Oriented Data. Springer.
35. Wattenberg, M. (2005). "A Note on Space-Filling Visualizations and Space-Filling Curves". IEEE Symposium on Information Visualization, 181-186.
36. Holten, D., & van Wijk, J. J. (2009). "Force-Directed Edge Bundling for Graph Visualization". Computer Graphics Forum, 28(3), 983–990.
37. Heer, J., Card, S. K., & Landay, J. A. (2005). "Prefuse: A Toolkit for Interactive Information Visualization". Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 421–430.
38. Kosara, R., & Mackinlay, J. (2013). "Storytelling: The Next Step for Visualization". Computer, 46(5), 44–50.
39. Lau, L., & Lee, J. (2015). Interactive Data Visualization with D3.js. Packt Publishing.
40. Ward, M. O., Grinstein, G. G., & Keim, D. A. (2015). Interactive Data Visualization: Foundations, Techniques, and Applications. CRC Press.

41. Fry, B. (2008). Visualizing Data: Exploring and Explaining Data with the Processing Environment. O'Reilly Media.
42. Few, S. (2012). Show Me the Numbers: Designing Tables and Graphs to Enlighten. Analytics Press.
43. Norman, D. A. (2013). The Design of Everyday Things: Revised and Expanded Edition. Basic Books.
44. Shneiderman, B. (1996). "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations". Proceedings of the IEEE Symposium on Visual Languages, 336–343.
45. Battelle, J. (2005). The Search: How Google and Its Rivals Rewrote the Rules of Business and Transformed Our Culture. Portfolio Hardcover.
46. Harris, R. L. (1999). Information Graphics: A Comprehensive Illustrated Reference. Oxford University Press.
47. Cairo, A. (2012). The Functional Art: An Introduction to Information Graphics and Visualization. New Riders.
48. Shneiderman, B., & Plaisant, C. (2010). Designing the User Interface: Strategies for Effective Human-Computer Interaction. Addison-Wesley.
49. Kirk, A. (2016). Data Visualisation: A Handbook for Data Driven Design. Sage Publications.
50. Riche, N. H., & Lee, B. (2010). "Data Sketching as a Practice for Data Visualization". ACM CHI Conference on Human Factors in Computing Systems, 13-18.
51. Steele, J., & Iliinsky, N. (2010). Beautiful Visualization: Looking at Data through the Eyes of Experts. O'Reilly Media.
52. Meeks, E. (2019). Hands-On Data Visualization: Interactive Storytelling from Spreadsheets to Code. O'Reilly Media.
53. Thomas, J., & Cook, K. (2005). Illuminating the Path: The Research and Development Agenda for Visual Analytics. IEEE Computer Society Press.

54. Li, W., Luo, J., & Su, H. (2018). "Enhancing Interactive Data Visualization with Real-Time Web". International Conference on Data Science and Advanced Analytics (DSAA), 82–91.
55. Weber, M. A., & Isenberg, T. (2019). "Visualizing Large Graphs in JavaScript: Web-Based Solutions". EuroVis Workshop on Graph Visualization, 14-21.