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

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

«На правах рукопису» УДК 004.04

«До захисту допущено»
Завідувач кафедри
_____Едуард ЖАРІКОВ
«___»_____2022 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-професійною програмою «Інженерія програмного забезпечення інформаційних систем»
зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Засіб для моделювання екосистем інженерії програмного забезпечення»

Виконав:
студент ІІ курсу, групи ІП-13мп
Скригун Владислав Олександрович _____

Керівник:
д.т.н., професор кафедри
інформатики та програмної інженерії
Сидоров Микола Олександрович _____

Рецензент:
к.т.н., доцент
Ткач Михайло Мартинович _____

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

Київ – 2022 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-професійна програма «Інженерія програмного забезпечення інформаційних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____Едуард ЖАРІКОВ

«___»_____2022 р.

ЗАВДАННЯ

на магістерську дисертацію студенту

Скригуну Владиславу Олександровичу

1. Тема дисертації «Засіб для моделювання екосистем інженерії програмного забезпечення», науковий керівник дисертації Сидоров Микола Олександрович, д.т.н., професор, затверджені наказом по університету від «09» листопада 2022 р. № 4115-с
2. Термін подання студентом дисертації «06» грудня 2022 р.
3. Об’єкт дослідження: екосистеми програмного забезпечення.
4. Предмет дослідження: методи моделювання екосистем інженерії програмного забезпечення.
5. Перелік завдань, які потрібно розробити: на основі поглибленого порівняльного аналізу існуючих теоретичних положень уточнити сутність екосистеми програмного забезпечення; проаналізувати існуючі впровадження програмних продуктів-аналогів задля виявлення їхніх основних переваг і недоліків; визначити шляхи вирішення основних задач моделювання екосистем інженерії програмного забезпечення; розробити засіб для моделювання екосистем інженерії програмного забезпечення.
6. Орієнтовний перелік графічного (ілюстративного) матеріалу – 3 плакати
7. Орієнтовний перелік публікацій: 1. Скригун В.О. Екосистеми програмного забезпечення: сутність та типи. *Світ наукових досліджень. Випуск 13*: матеріали Міжнародної мультидисциплінарної наукової інтернет-конференції, (м. Тернопіль, Україна – м. Переворськ, Польща, 25-26 жовтня 2022 р.) / [редкол. :

О. Патряк та ін.]; ГО “Наукова спільнота”; WSSG w Przeworsku. – Тернопіль: ФО-П Шпак В.Б., 2022. С. 82-83.
2. Скригун В.О. Моделювання екосистеми інженерії програмного забезпечення. Матеріали Всеукраїнської науково-практичної конференції молодих вчених та студентів «Інженерія програмного забезпечення і передові інформаційні технології» (SoftTech2022). Секція кафедри інформатики та програмної інженерії. Матеріали конференції. Київ. 2022. 23-25 листопада 2022 р.

8. Консультанти розділів дисертації

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

9. Дата видачі завдання «6» жовтня 2022 р.

Календарний план

№ п/п	Назва етапів виконання магістерської дисертації	Термін виконання	Примітка
1	Ознайомлення з предметною галуззю	6.10.2022	
2	Визначення структури магістерської дисертації; пошук та вивчення літератури	10.10.2022	
3	Робота над першим розділом магістерської дисертації	26.10.2022	
4	Робота над другим розділом магістерської дисертації	3.11.2022	
5	Робота над третім розділом магістерської дисертації	10.11.2022	
6	Розроблення стартап-проєкту	18.11.2022	
8	Оформлення текстової та графічної частини магістерської дисертації	20.11.2022	
9	Подання дисертації на попередній захист	22.11.2022	
10	Подання дисертації на захист	06.12.2022	

Студент Владислав СКРИГУН

Науковий керівник Микола СИДОРОВ

РЕФЕРАТ

Розмір пояснювальної записки – 90 аркушів, містить 15 ілюстрацій, 24 таблиці, 1 додаток, 32 джерела.

Актуальність теми. Екосистема програмного забезпечення набуває все більшої актуальності в наш час. Оскільки за допомогою моделювання екосистем інженерії програмного забезпечення стає можливим вирішення не складних відносин між продуктами компаній в індустрії програмного забезпечення, але й поліпшення розроблення безпосереднього програмного забезпечення. З огляду на зазначене виявлено потребу в розробленні засобу для моделювання екосистем інженерії програмного забезпечення.

Мета дослідження. Основною метою є розроблення засобу для моделювання екосистем інженерії програмного забезпечення, який буде доступний як вебзастосунок, що дозволить уникнути додаткової розробки для різних операційних систем.

Для реалізації поставленої мети **сформульовані такі завдання**:

– на основі поглибленого порівняльного аналізу існуючих теоретичних положень уточнити сутність екосистеми програмного забезпечення;

– проаналізувати існуючі впровадження програмних продуктів-аналогів задля виявлення їхніх основних переваг і недоліків;

– визначити шляхи вирішення основних задач моделювання екосистем інженерії програмного забезпечення;

– розробити засіб для моделювання екосистем інженерії програмного забезпечення.

Об’єкт дослідження: екосистеми програмного забезпечення.

Предмет дослідження: засіб моделювання екосистем інженерії програмного забезпечення.

В ході здійснення дослідження було використано метод systematic mapping study (систематичний огляд літератури) задля вивчення та аналізу предметної області дослідження з текстових джерел інформації та метод case study (метод

ситуативного аналізу) задля аналізу розробленого методу побудови програмного забезпечення.

Наукова новизна результатів магістерської дисертації полягає в тому, що запропоновано засіб для моделювання екосистем інженерії програмного забезпечення.

Практичне значення отриманих результатів полягає в розробленні програмного забезпечення для моделювання екосистем інженерії програмного забезпечення.

Зв'язок з науковими програмами, планами, темами. Робота виконувалась на кафедрі інформатики та програмної інженерії Національного технічного університету України "Київський політехнічний інститут імені Ігоря Сікорського".

Апробація. Наукові положення дисертації пройшли апробацію на Міжнародній мультидисциплінарній науковій інтернет-конференції (м. Тернопіль, Україна – м. Переворськ, Польща, 25-26 жовтня 2022 р.)

Публікації. 1. Скригун В.О. Екосистеми програмного забезпечення: сутність та типи. *Світ наукових досліджень. Випуск 13*: матеріали Міжнародної мультидисциплінарної наукової інтернет-конференції, (м. Тернопіль, Україна – м. Переворськ, Польща, 25-26 жовтня 2022 р.) / [редкол.: О. Патряк та ін.]; ГО "Наукова спільнота"; WSSG w Przeworsku. Тернопіль: ФО-П Шпак В.Б., 2022. С. 82-83. 2. Скригун В.О. Моделювання екосистеми інженерії програмного забезпечення. Матеріали Третьої Всеукраїнської науково-практичної конференції молодих вчених та студентів *«Інженерія програмного забезпечення і передові інформаційні технології» (SoftTech2022)*. Секція кафедри інформатики та програмної інженерії. Київ. 2022. 23-25 листопада 2022 р. Київ: 2022.

Ключові слова: ЕКОСИСТЕМА, МОДЕЛЮВАННЯ, ЕКОСИСТЕМИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ВЕБДОДАТОК.

ABSTRACT

Explanatory note size – 90 pages, contains 15 illustrations, 24 tables, 1 application, 32 references.

Topicality. The software ecosystem is becoming more and more relevant nowadays. Because with the help of modeling software engineering ecosystems, it becomes possible to solve not only the complex relationships between the products of companies in the software industry, but also to improve the development of direct software. In view of the above, the need to develop a tool for modeling software engineering ecosystems was identified.

The purpose of the dissertation research is developmenting of a tool for modeling software engineering ecosystems that will be available as a web application, which will avoid additional development for different operating systems.

In order to realize the set goal, the following tasks have been formulated:

- on the basis of an in-depth comparative analysis of the existing theoretical provisions, clarify the essence of the software ecosystem;

analyze existing implementations of similar software products to identify their main advantages and disadvantages;

- to determine ways to solve the main problems of modeling software engineering ecosystems;

- to develop a tool for modeling software engineering ecosystems.

Research object: software ecosystems.

The subject of research: a software engineering ecosystem modeling tool.

In the course of the research, the method of systematic mapping study (systematic literature review) was used to study and analyze the subject area of research from textual sources of information and the case study method (method of situational analysis) was used to analyze the developed method of software construction.

The scientific novelty of the results of the master's thesis lies in the fact that

a tool for modeling software engineering ecosystems is proposed.

The practical significance of the obtained results lies in the development of software for modeling software engineering ecosystems.

Relationship with working with scientific programs, plans, topics. Work was performed at the Department of Informatics and Software Engineering of the National Technical University of Ukraine «Kyiv Polytechnic Institute. Igor Sikorsky».

Approbation. The scientific provisions of the dissertation were tested at the International Multidisciplinary Scientific Internet Conference (Ternopil, Ukraine - Perevorsk, Poland, October 25-26, 2022); Third Ukrainian Scientific and Practical Conference of Young Scientists and Students "Software Engineering and Advanced Information Technologies"

Publications. 1. Skryhun V.O. Software ecosystems: essence and types. The world of scientific research. Issue 13: materials of the International Multidisciplinary Scientific Internet Conference, (Ternopil, Ukraine - Perevorsk, Poland, October 25-26, 2022) / [edited by: O. Patryak and others]; NGO "Scientific Community"; WSSG in Przeworsk. – Ternopil: FO-P Shpak V.B., 2022. P. 82-83. 2. Skryhun V.O. Modeling the software engineering ecosystem. Materials of the Third All-Ukrainian Scientific and Practical Conference of Young Scientists and Students "Software Engineering and Advanced Information Technologies" (SoftTech2022). Section of the Department of Informatics and Software Engineering. Kyiv. 2022. November 23-25, 2022. Kyiv: 2022.

Keywords: ECOSYSTEM, MODELING, SOFTWARE ENGINEERING ECOSYSTEMS, WEB APPLICATION.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення

DOM – document object model

SECO – екосистема програмного забезпечення

SSN – постачання програмного забезпечення

OSS – Open Sound System, відкрите програмне забезпечення

ВСТУП

Актуальність теми викликана постійним розвитком індустрії програмного забезпечення, а експерименти із новими бізнес-моделями призводить до фундаментальних змін в галузевих структурах та взаємозв'язках фірми та клієнта. Окрім того, що продукти та технології швидко розвиваються, багато інноваційних компаній експериментують із новими бізнес-моделями, що час від часу призводить до фундаментальних зрушень у цілих галузевих структурах і взаємозв'язках між компаніями та клієнтами. Останнім часом багато компаній прийняли стратегію використання платформи для залучення розробників програмного забезпечення, а також кінцевих користувачів, створюючи навколо себе цілі екосистеми програмного забезпечення (SECO).

Програмні екосистеми стають все більш популярною формою організації галузі, яку просувають провідні постачальники програмного забезпечення. Через свою новизну та складність багато питань залишаються неясними для різних сторін, залучених до екосистеми. Перехід від більш звичайних галузевих структур і відносин до екосистеми, ймовірно, матиме глибокий вплив на бізнес, а також на вибір технічного дизайну. Тому особливої актуальності набувають засоби для моделювання екосистеми інженерії програмного забезпечення.

Метою дослідження є розроблення засобу для моделювання екосистем інженерії програмного забезпечення, який буде доступний як вебзастосунок, а тому буде відсутня потреба у додатковій розробці для різних операційних систем.

Для реалізації поставленої мети сформульовані такі *завдання*:

− вивчення та аналіз наявних впроваджених програмних продуктів-аналогів задля виявлення їхніх основних переваг і недоліків;

- аналіз наукових досліджень з метою дослідження шляхів вирішення основних задач моделювання екосистем інженерії програмного забезпечення;

- розроблення засобу для моделювання екосистем інженерії програмного забезпечення;

- написання програмного забезпечення на основі запропонованого методу, аналіз його основних переваг та недоліків, та доведення його ефективності.

Об'єкт дослідження: екосистеми програмного забезпечення.

Предмет дослідження: засіб моделювання екосистем інженерії програмного забезпечення.

В ході здійснення дослідження було використано метод systematic mapping study (систематичний огляд літератури) задля вивчення та аналізу предметної області дослідження з текстових джерел інформації та метод case study (метод ситуативного аналізу) задля аналізу розробленого методу побудови програмного забезпечення.

Наукова новизна результатів магістерської дисертації полягає в тому, що запропоновано засіб для моделювання екосистем інженерії програмного забезпечення.

Практичне значення отриманих результатів полягає в розробленні програмного забезпечення для моделювання екосистем інженерії програмного забезпечення.

Апробацію результатів досліджень, проведених у рамках дисертації, було здійснено на Міжнародній мультидисциплінарній науковій інтернет-конференції (м. Тернопіль, Україна – м. Переворськ, Польща, 25-26 жовтня 2022 р.)

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ. АНАЛІЗ ВИМОГ ДО ЗАСОБУ ДЛЯ МОДЕЛЮВАННЯ ЕКОСИСТЕМ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Загальні положення

Програмне забезпечення стало глобальним явищем, яке включає в себе велику кількість тісно взаємодіючих соціально-технічних систем. Сукупність сучасного програмного забезпечення характеризується обсягом і складністю, глобальним поширенням, тісною та складною взаємодією систем, появою нових видів взаємодій при розробленні, супроводі, обміні, використанні та поширенні, а також широкою інтеграцією в соціальні відносини.

Сучасне програмне забезпечення характеризується великим обсягом і складною взаємодією систем, що обумовлює необхідність розроблення додаткових понять та концепцій для його вивчення та опису. Одним з напрямів такого пошуку вбачаємо у використанні концепцій біологічних екосистем. Подальше вивчення та побудова засобу для моделювання екосистем інженерії програмного забезпечення наразі є актуальним. Екосистема програмного забезпечення поєднує в собі програмне забезпечення та середовище його використання.

1.2 Огляд літератури із використанням методу Systematic Mapping Study

Предметна область екосистем інженерії програмного забезпечення досить широка і вимагає ретельного розгляду, оскільки наявна значна кількість наукових праць, публікацій та інших джерел інформації. Задля того, аби дослідити та ретельно проаналізувати значну кількість інформації, особливо з мережі Інтернет, необхідно витратити значний часовий ресурс.

Для вирішення вищезазначеної проблеми та ефективного дослідження предметної області в представленому науковому дослідженні автором було використано спеціальний метод для пошуку та фільтрації найбільш значимих для конкретного випадку джерел інформації - Systematic Mapping Study.

Systematic Mapping Study включає аналіз інформації для створення загального уявлення про область дослідження предмета; це працює як попередній крок аналізу до систематичного огляду літератури або мета-аналізу. Метою систематичного картографування є класифікувати, аналізувати та ідентифікувати бази даних публікацій наукових звітів, щоб знайти прогалини в знаннях. Тематичний аналіз визначає кількісну кількість публікацій. Таким чином, систематичне відображення пропонує структуру щодо типу наукових звітів та їх результатів. Цей метод допомагає визначити сучасний стан, зібрати інформацію про тенденції та визначити авторів, журнали та провідні публікації з теми.

Методологічний процес систематичного картографування дотримується протоколу, який включає деякі визначені кроки для роботи над науковими звітами щодо галузі дослідження.

Алгоритм роботи Systematic Mapping Study наведено на рис. 1.1.

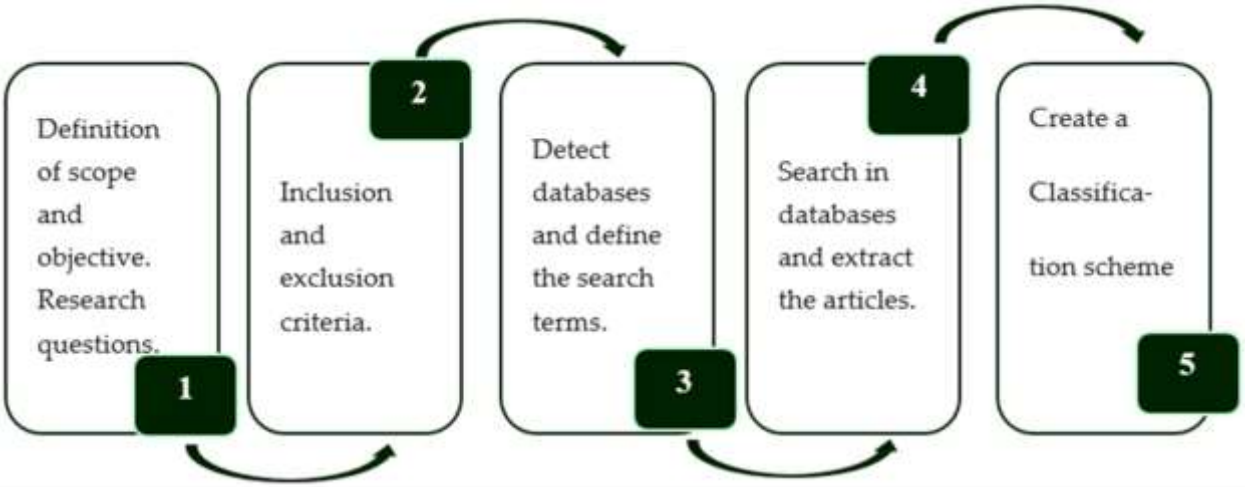

Рисунок 1.1 – Алгоритм роботи Systematic Mapping Study

Розглянемо кожен з етапів детальніше.

Systematic Mapping Study починається з дослідницьких запитань, відповіді на які дають огляд первинних досліджень, і такі запитання зведені в таблиці за категоріями. Методологія планується відповідно до обсягу та мети дослідження, що має вказувати на можливість виявлення прогалин у дослідженні та можливості для майбутніх досліджень. Для цього розроблено запитання, що наведено в табл. 1.1.

Таблиця 1.1 – Перелік запитань дослідження

№	Запитання
1.	What is the software engineering ecosystem?
2.	What is the main method for modeling the software engineering ecosystem?
3.	Which tools are there for modeling the software engineering ecosystem?
4.	What are the main approaches of implementation of software engineering ecosystem?

Наступним етапом є побудова критеріїв включення та виключення. Критерії включення (табл. 1.2) та виключення використовуються для виключення досліджень, які не є релевантними для відповідей на запитання дослідження.

Таблиця 1.2 – Критерії включення

IC1	Тільки англомовні дослідження
IC2	Дослідження, що стосуються і посилаються на будь-які теми, пов’язані з екосистемою інженерії програмного забезпечення
IC3	Дослідження за останні 7 років

У методологічних кроках зазначено доцільність виключення досліджень, які знижують точність результатів. Тому необхідно визначити критерії з урахуванням часу, типу документа, мови, актуальності теми дослідження. Для цього дослідження умови включення були такими: дослідження відкритих інновацій, статті, опубліковані лише в журналах, які можна знайти в двох базах даних: Web of Science (WOS), IEEE. Критеріями

виключення були такі: повторювані статті, ESCI (Emerging Source Citation Index), книги, розділи книг і виступи, тези, а критерії якості охоплювали показники щодо внеску цінної інформації про відкриті інновації та узгодженості між метою, методом і результати (табл. 1.3).

Таблиця 1.3 – Критерії виключення

EC1	Повторні дослідження, які знайдені в різних пошукових системах. У даному випадку було розглянуто тільки одне дослідження
EC2	Дубльовані дослідження, які повідомляють про подібні результати. В даному випадку було розглянуто лише найповніше дослідження
EC3	Недоступні книги та роботи

Наступний етап – ідентифікація баз даних і пошукових термінів. Для створення пошукових запитів використовуються бази даних WOS, IEEE з метою перегляду наукових звітів. Рядок пошуку було визначено таким чином, аби однаково застосовуватися в обох базах даних відповідно до питань дослідження та критеріїв включення та виключення. Інакше результати пошуку неможливо було б порівняти. Для пошуку матеріалів, проаналізованих у цьому дослідженні, булеві вирази подібним чином використовувалися в обох базах даних. Використані пошукові рядки наведено в табл. 1.4. Як можливу оцінку ризику зазначимо, що відбір доказів статей проводився за допомогою методу вибірки зазначених пошукових ланцюгів відповідно до сфери інтересів. Результати селективності були основою для роботи.

Таблиця 1.4 – Пошукові стрічки

Q1	"modeling the software engineering ecosystem"
Q2	software engineering ecosystem
Q3	"software ecosystem" AND ("tools" OR "main approach for")

Наступний етап – ідентифікація баз даних і пошукових термінів. Для створення пошукових запитів використовуються бази даних WOS і IEEE,

щоб переглядати наукові звіти, які розкривають емпіричні докази щодо відкритих інновацій. Рядок пошуку було визначено таким чином, аби однаково застосовуватися в обох базах даних відповідно до питань дослідження та критеріїв включення та виключення. Інакше результати пошуку неможливо було б порівняти. Для пошуку матеріалів, проаналізованих у цьому дослідженні, булеві вирази подібним чином використовувалися в обох базах даних. Використані пошукові рядки наведено в таблиці. Як можливу оцінку ризику слід зазначити, що відбір доказів статей проводився за допомогою методу вибірки зазначених пошукових ланцюгів відповідно до сфери інтересів для дослідження. Результати селективності були основою для роботи.

Таблиця 1.5 – Результати пошуку

Пошукова стрічка	Кількість знайдених публікацій
"modeling the software engineering ecosystem"	3289
“software engineering ecosystem”	5671
"software ecosystem" AND ("tools" OR "main approach for")	2 694

Наступний етап – створення схеми класифікації (табл. 1.6).

Таблиця 1.6 – Категорія та її опис

Категорія	Класифікаційний опис
1	2
Evaluation Research	Метод впроваджується на практиці, проводиться оцінювання методу. Це означає, що показано, як цей метод реалізується на практиці (впровадження рішення), і якими будуть наслідки його впровадження з точки зору переваг та недоліків (оцінка впровадження). Це також включає виявлення проблем в індустрії.
Validation Research	Досліджені методи є новими та ще не були впроваджені на практиці. Методи можуть бути використані, наприклад, для експериментів.

Кінець таблиці 1.6

1	2
Solution Proposal	Запропоновано рішення проблеми, яке може бути або новим, або представляти собою суттєве розширення наявного методу. Потенційні переваги та придатність рішення підкріплені невеликим прикладом або переконливою аргументацією.
Philosophical Papers	Ці роботи представляють новий спосіб розгляду існуючих понять шляхом структурування галузі у формі таксономії або концептуальної основи.

Таблиця 1.7 – Класифікація відібраних публікацій за галуззю

Назва галузі	Публікації	Загальна кількість
Data stream processing	1,4,7,9,12,15,22	7
Live data, real-time data gathering	10,11,16	3
Data extraction/analysis	3,5,8,13,14,17	6
Big data storage	2,6,18,19,20,21	6

Отже, з-поміж масиву досліджених публікацій є небагато наукових статей, які пропонують нові методи вирішення задачі щодо вибору засобу для моделювання екосистем інженерії програмного забезпечення. Це ще раз підтверджує доцільність і актуальність проведення даного дослідження.

1.3 Огляд літератури

Використовуючи метод Systematic Mapping Study в попередньому розділі, було відібрано такі наукові праці в області екосистем інженерії програмного забезпечення, а саме:

− «An Exploratory Study on the Need for Modeling Software Ecosystems: The Case of SOLAR SECO» [7];

− «A Tool for Supporting the Teaching and Modeling of Software Ecosystems Using SSN Notation» [12];

− «Understanding Software Ecosystems: A Strategic Modeling Approach» [11];

− «A SECO meta-model: A common vocabulary of the SECO research domain» [29]

− «How to Evaluate the Productivity of Software Ecosystem: A Case Study in GitHub» [32].

У роботі «Екосистеми програмного забезпечення» [9] автори описують типові елементи екосистем та їхній контекст, намагаючись на якісному рівні спрогнозувати характеристики розвитку екосистеми програмного забезпечення з точки зору подальшого підвищення ефективності технологій розроблення, появи та розвитку нових областей застосування програмного застосування.

Екосистеми характеризуються таким [4]:

− розмірами (чим вона більше, тим менше залежить від зовнішніх частин);

− інтенсивністю обміну (чим він більший, тим більше приплив і відтік);

− збалансованістю процесів (чим сильніше порушено рівновагу, тим більше повинні бути зміни ззовні для його відновлення);

− стадією розвитку системи.

Окремі вчені пропонують [2] використання екологічного підходу до дослідження програмного забезпечення. Основні положення екології програмного забезпечення вчений пропонує розглядати як частини інженерії програмного забезпечення і наводить такі напрямки досліджень програмного забезпечення: «зелене» програмне забезпечення, сталий розвиток, цифрові екосистеми.

Важливим кроком для побудово екосистеми програмного забезпечення є розуміння взаємозв’язків між певними вже існуючими екосистемами, оскільки необхідно чітко бачити взаємозв’язки між вже існуючими екосистемами задля успішної інтеграції новоствореної екосистеми. Частково

вирішується це питання вченими [7] у запропонованій ними концептуальній карті (рис. 1.2), де прослідковується те, що SECO може бути пов’язане з іншим SECO. SECO має платформу, на якій можуть надаватися продукти та послуги бути включені, змінені або розширені як програмні артефакти. SECO також складається зі спільноти хабів, тобто головні агенти в SECO (наприклад, провідні організації) і нішевих гравців, тобто всіх зацікавлених сторін, які колективно впливають на SECO від індивідуальних дій до платформи (наприклад, кожен із них може впливати, зобов’язуватися, сприяти, просувати або розширювати платформу). Обидва типи центральні гравці в SECO пов’язані з певною роллю (наприклад, розробник, посередник, кінцевий користувач тощо).

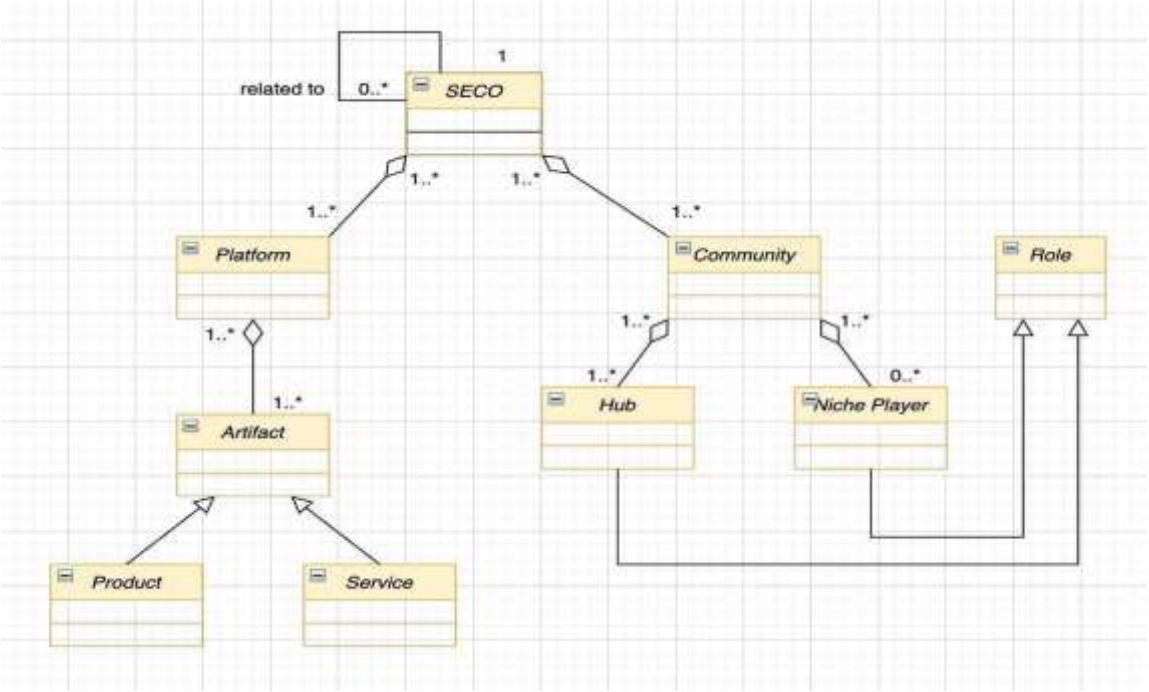

Рисунок 1.2 – Концептуальна карта SECO [7]

Огляд взаємозв’язків екосистем програмного забезпечення виявив необхідність визначення SECO, його основних ролей та проблематику моделювання екосистем програмної інженерії.

Цінність роботи вчених [12] у тому, що вони пропонують:

- запровадження визначення SECO;
- виокремлення основних ролей в SECO моделі;

− підкреслення проблематики моделювання екосистем програмної інженерії.

Екосистему інженерії програмного забезпечення – SECO можна визначити як набір акторів, які функціонують як єдине ціле, що взаємодіє з розподіленим ринком між програмним забезпеченням і послугами. Ці взаємодії є портованою технологічною платформою або спільним ринком і здійснюються шляхом обміну інформацією, артефактами та ресурсами.

SECO може розглядати в трьох вимірах:

- архітектура. Передбачає платформи (технології чи інфраструктури), на якій SECO буде вставлене, а також питання архітектури програмного забезпечення;

- бізнес: включає знання про ринок, рішення, які повинні приймати актори про бізнес-моделі, визначення продукту SECO портфоліо, ліцензування та стратегії продажів;

- соціальні: де визначається, як мережа акторів буде відноситись до SECO досягти поставлених цілей і сприяти зростанню SECO пропозиція, де кожен може отримати вигоду.

У табл. 1.8 наведено роді та їхні описи у SECO. Деякі приклади SECO: MySQL/PHP, Eclipse, Microsoft.

Таблиця 1.8 – Ключові ролі їхні описи у SECO

Ключові ролі	Опис у SECO
Основа (Keystone)	Опис чи мала група, що певним чином керує розробкою основної частини програмного забезпечення технології
Кінцевий користувач (end-user)	Ключова роль для основи програмного забезпечення, оскільки він визначає, хто потребує взаємодії з бізнесом будь якого типу.
Сторонні організації (third-party organization)	Технології програмного забезпечення, як базис для створення відповідних рішень чи сервісів (включаючи сторонніх розробників)

Однією з проблем, з якою стикаються при моделюванні SECO, є відсутність стандартизації, тому вченими [12] було запропоновано спосіб стандартизації моделювання SECO за допомогою стратегії мережі

постачання програмного забезпечення (SSN). SSN представляє собою серію пов’язаних організацій програмного забезпечення, обладнання та послуг, які співпрацюють, щоб задовольнити потреби ринку. SSN представляє головних учасників та їхню взаємодію в рамках SECO за допомогою ключових елементів. На рис. 1.3 показано елементи, які використовуються в нотації SSN для моделювання SECO.

Компанія, яка вас цікавить	Компанія, яка вас цікавить / продукт, який вас цікавить - це безпосередньо сама компанія / продукт, яка розповсюджує продукт в бізнес-моделі, визначеній для навколишнього середовища.
Постачальник	Постачальник: компанія або постачальник продуктів і / або послуг, який надає один або більше необхідних продуктів чи послуг.
Клієнт	Клієнт: суб'єкт, який прямо чи опосередковано отримує або використовує продукт.
Посередник	Посередник: компанія, продукт або послуга, які працюють між двома учасниками, щоб поширювати продукт або послугу. Це можуть бути дистриб'ютори, продавці, перекупники тощо.
Агрегатор	Агрегатор: компанії, продукти або послуги, які працюють між двома агентами, щоб додати цінність продукту або послуги, а також розповсюджувати чи перепродавати його.
Клієнт клієнта	Клієнт клієнта: у клієнта можуть бути власні клієнти, яким надаються продукти чи послуги прямо чи опосередковано. Приклади: підтримка продукту, оновлення тощо.
X.Y	Торгові відносини: представляє собою артефакт або потік послуг від одного актора до іншого. Це можуть бути дані, програмне забезпечення, послуги, гроші тощо.
	Потік: з'єднує двох учасників. Стосунки можуть бути складними, що складаються з багатьох потоків довільних напрямків.
	OR Gateway: дозволяє виконувати один або більше потоків між входами.
	XOR Gateway : дозволяє виконувати лише один поточний вхід.

Рисунок 1.3 – Елементи нотації SNN [12]

На рис. 1.3 наведено короткий приклад використання моделювання SECO нотації SSN, що показує відносини постачальників і посередників з цільовою компанією. Відношення кожного елемента (їх може бути більше ніж один) може складатися з фінансових значень, типу даних або будь-якого іншого побічного продукту, що передається між елементами.

Для розроблення екосистем програмного забезпечення необхідно мати мета-модель, що може вирішити проблему опису та структуризації SECO. Вченими у праці [29] представлена мета-модель SECO, яка допомагає описати та структурувати SECO. Мета-модель покращує комунікацію щодо досліджень SECO, полегшуючи обмін тематичними дослідженнями або порівнюючи SECO та дослідження про SECO. Також авторами була створено мета-модель, яку можна використовувати для різних аспектів в межах домену SECO. Тепер дослідники можуть пов’язати свою роботу з мета-моделлю. Пов’язуючи свою роботу з певною сутністю в моделі, кожному досліднику стає зрозуміло, як його робота пов’язана з дослідженням SECO домену. Таким чином, модель може бути використана як основа для зв’язку майбутніх досліджень SECO з знаннями, які вже є в цій галузі.

Крім того, тепер є можливість робити узагальнення певних типів SECO, оскільки є можливим моделювання різних випадків типу SECO мета-модель. Потім можна ідентифікувати спільні елементи в цих моделях, що полегшує формалізацію типу або теорії SECO. Це допомагає формалізувати домен, оскільки більшість досліджень зараз проводяться в кожному конкретному випадку.

Таким чином, створення об’єднуючої структури шляхом поєднання існуючих сутностей моделі може допомогти у створенні послідовної та однозначної моделі, що полегшує міркування про SECO.

SECO мета модель - це модель, що забезпечує об’єднуючу структуру для забезпечення та перевірки послідовності, водночас надаючи засоби для розрізнення коректних та хибних моделей.

Для подальшої роботи з мета моделлю необхідно розглянути вимоги,

яким вона повинна відповідати. Вчені [29] виокремили чотири основні вимоги, яким повинна відповідати мета модель, а саме:

– метамодель має бути застосовною до кожного SECO;

– сутності, включені в мета-модель, повинні бути похідними з наукових джерел;

– метамодель має бути простою у використанні та зрозумілою;

– мета-модель повинна надавати розширений список універсальних термінів, щоб полегшити дослідникам обговорення SECO.

При побудові метамоделі екосистеми виокремлено такі її основні складові:

– актори та їх ролі – вченими [29] було запроваджено 46 сутностей акторів, кожен з яких може мати відповідні ролі. Загалом всіх можна поділити на два окремих класи - це Індивіди (Individuals) та Організації (organizations);

– продукт та платформа. Платформа в екосистемі є центральною сутністю, на якій діють різні актори. У більшості випадків сама платформа є певною формою продукту, який можна розширити за допомогою програм або розширень, створених іншими учасниками (акторами);

– стратегія (Strategy) – спосіб, яким актор в той чи інший спосіб взаємодіє з SECO;

– стан екосистеми (Ecosystem health) – Стан SECO є ключовим фактором, оскільки це основний спосіб вимірювання поточного стану екосистеми. До основних показників можна віднести надійність та продуктивність;

– межі (Boundaries) – Межі SECO, є визначальними властивостями, які описують що є частиною SECO, а що ні. Можна описати деякі початкові межі, наприклад ринок, на якому працює SECO і тип екосистеми, який має дана екосистема, який визначається за чотирма факторами: базова технологія, предметна область, ринок розширення та доступність.

Слід звернути увагу на доволі важливе питання, що було піднято вченими [15], а саме SECO та його життєвий спосіб. Те, як SECO розвивається, здебільшого регулюється постачальником платформи, центром SECO. Життєвий спосіб SECO також поділяється на три рівні:

- стратегічний
- тактичний
- оперативний

Стратегічний рівень SECO охоплює чотири ключові характеристики:

- бачення SECO;
- стратегія платформи;
- стабільність;
- управління репутацією.

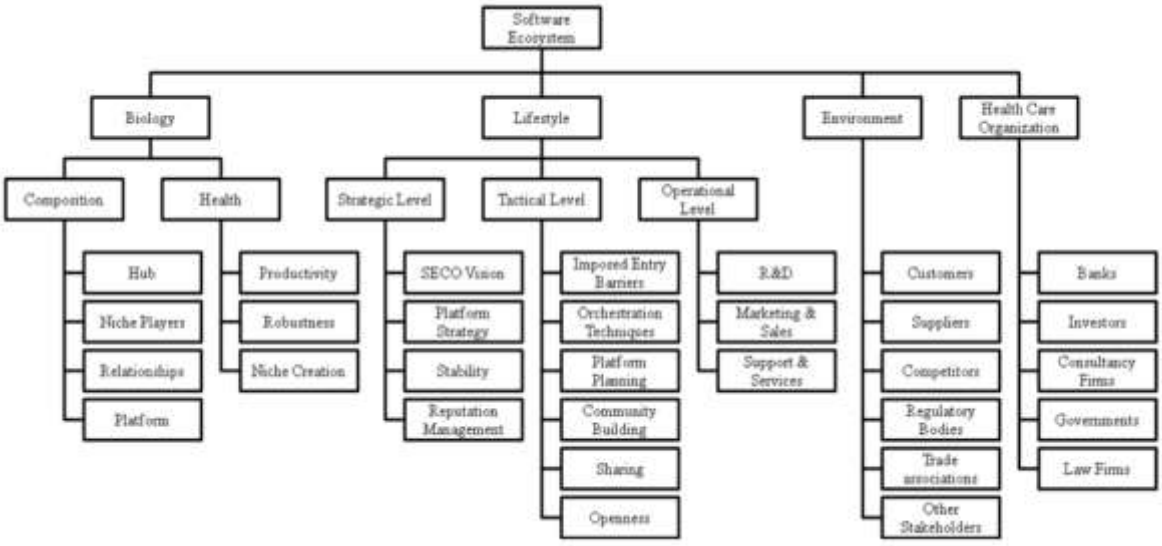

Рисунок 1.5 – Життєвий спосіб (lifestyle) SECO [15]

Після огляду життєвого способу SECO звернемо увагу на продуктивність екосистеми програмного забезпечення.

Оцінювання продуктивності SECO здійснювали вчені [32], які запропонували визначення продуктивності екосистеми програмного забезпечення. Подібно до первинної та вторинної продуктивності природних екосистем, продуктивність програмної екосистеми розкладається на

первинну та вторинну продуктивність відповідно до різних впливів діяльності користувачів на екосистеми. Крім того, продуктивність була кількісно визначена в різних екосистемах програмного забезпечення. На думку вчених, відповідно до джерела продуктивності кількість, масштаб і активність учасників відіграють найбільш безпосередню та важливу роль у продуктивності екосистеми. У процесі перевірки зв'язку між кількістю та активністю учасників і продуктивністю програмної екосистеми ними було виявлено, що первинна продуктивність сильно корелює з кількістю учасників, тоді як вторинна продуктивність менше корелює з кількістю учасників. Вчені представили модель продуктивності екосистеми програмного забезпечення, справедливість якої перевірена експериментами.

Також вчені вчені [32] виділили відповідні фактори продуктивності екосистеми програмного забезпечення та модель композиції, що їх можна використати для подальших досліджень екосистеми програмного забезпечення, вимірювання стану екосистеми програмного забезпечення, вимірювання ефективності розробки тощо, досліджено вплив непрямих факторів, таких як відгуки користувачів і фактори зовнішнього середовища, на продуктивність екосистеми програмного забезпечення.

Деякі загрози впливають на валідність методів вимірювання, включаючи зовнішні загрози та внутрішні загрози. Основною зовнішньою загрозою є валідність наборів даних. Таким чином, якість моделі визначається якістю використаного набору даних. Верифікаційні набори даних Stack Overflow і Bugzilla були скинуті з їхніх платформ, а набори даних GitHub були отримані через службу даних GitHub API V3. Маленька екосистема та невелика кількість даних призведуть до неточного отримання параметрів у моделі, що загрожуватиме дійсності висновку. Внутрішня загроза полягає в тому, що модель враховує лише кількість користувачів і вплив дій користувачів на продуктивність. Однак на точність моделі можуть впливати й інші фактори. Через особливості великої кількості даних, які використовуються для побудови моделі, характеристики деяких конкретних

проєктів в екосистемі могли бути проігноровані. Ця модель розглядає продуктивність екосистеми програмного забезпечення з кількісної, а не якісної точки зору, ігноруючи вплив різних подій на екосистему. Наприклад, в екосистемі на рівні проєкту деякі основні розробники часто надсилають код протягом періоду спостереження.

Аналіз показав, що також залишається відкритим питанням засобів моделювання для різних аспектів розгляду екосистем SECO. В роботі [28] вчені наголошують, що моделі програмної екосистеми можуть мати три рівні обсягу (рис. 1.6).

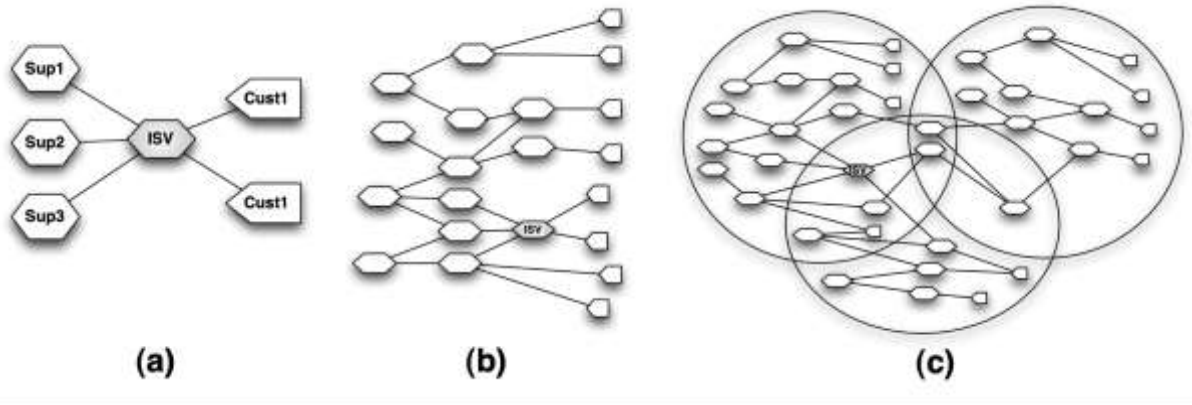

Рисунок 1.6 – Рівні обсягу моделі програмної екосистеми

На кожному з цих рівнів обсягу розглядаються різні сутності. На рівні обсягу мережі постачання програмного забезпечення (a) об’єктами дослідження є суб’єкти та їхні стосунки. На рівні обсягу SECO (b) об’єктами дослідження є мережі постачання програмного забезпечення (SSN) та їхні різні взаємозв’язки. на рівні обсягу SECO (c) об’єктами дослідження є самі SECO та зв’язки між ними. На кожному рівні існують різні дослідницькі проблеми, починаючи від впливу архітектурних змін на SECO і закінчуючи розробкою загальних показників для будь-якого SECO.

Моделювання SECO, на думку [21], базується на таких аспектах:

– проєктоцентричні аспекти стосуються проєктів, які складають екосистему, їхніх властивостей, їхніх взаємозв’язків та їхньої еволюції;

– аспекти, орієнтовані на розробника – стосуються діяльності, співпраці та взаємозалежності розробників, які роблять внесок в екосистему.

Для кожного з цих аспектів екосистема може відігравати одну з двох ролей в аналізі:

– фокус. Мета полягає в тому, щоб зрозуміти екосистему з цілісної точки зору. Цей тип аналізу починається з передумови, що проблеми, пов’язані з екосистемою, відрізняються від проблем, пов’язаних з окремими системами;

– контекст. Мета полягає в тому, аби зрозуміти окремі елементи, тобто проєкти екосистеми. Цей тип аналізу починається з передумови, що можна краще зрозуміти елемент, якщо вивчати його в контексті.

Розглянемо можливі типи для моделювання екосистем програмного забезпечення:

− проєктоцентричний з фокусуванням на екосистему – надання цілісного розуміння екосистеми з точки зору її проєктів (складові);

− проєктоцентричний з екосистемою в контексті – має на меті надати розуміння окремих проєктів у контексті екосистеми;

− аспект, орієнтований на розробника з фокусуванням на екосистему – надання цілісного розуміння екосистеми на основі структури співпраці її розробників;

− аспект, орієнтований на розробника з екосистемою в контексті –дає розуміння внеску кожного окремого розробника в екосистему.

Вважаємо, що доречним є порівняння екосистеми програмного забезпечення з екосистемою в біосфері. Аналіз наукового доробку вчених [10] дозволив виділити такі вимоги до екосистеми програмного забезпечення:

– наявність чітко виділених взаємозв’язків між її компонентами. Подібно до харчових ланцюгів і біогеохімічних циклів у природних екосистемах, програмна екосистема може існувати завдяки поширенню знань між її компонентами. Це означає, що передача знань між різними учасниками (наприклад, від постачальника до споживача, від розробника до користувача) повинна бути посилена шляхом правильного визначення «носіїв знань»;

– наявність різноманітності учасників екосистеми. Успішна екосистема повинна підтримувати спільноту шляхом залучення широкого кола учасників – це дозволяє екосистемі процвітати, навіть якщо вона втрачає частину своїх компонентів, оскільки інші учасники, що залишилися, продовжують виконувати основні ролі. Наприклад, платформа Eclipse є успішною, тому що вона не визначає себе як IDE. Його часто описують як платформу, яка включає широкий спектр користувачів;

– здатність до рефлексії екосистеми. Здорова екосистема має бути стійкою, придатною для життя, життєздатною та процвітаючою. Мета полягає в тому, щоб створити взаємозв'зки між компонентами, які допомагають один одному залишатися в прийнятній формі для виконання свого цільового призначення;

– можливість приймати рішення поза межами організації чи системи. У природних екосистемах різні особини або види в екосистемі взаємодіють один з одним за допомогою спільних ресурсів. Індивіди (учасники) самі приймають рішення (місцеві рішення), а загальний ефект є сумою рішень окремої людини. Можливо, екосистеми потребуватимуть відновлення або прийняття нових правил після того, як вони вже функціонуватимуть. У таких випадках зміна вимагає від особи, яка приймає рішення, запровадити нові правила поза межами організації. Сам процес прийняття рішень повинен враховувати кілька аспектів (технічних, ділових, соціальних тощо), щоб не порушити баланс екосистем;

– адаптивність артефактів, процесів і зацікавлених сторін. Щоб залучити в екосистему велику кількість постачальників, розробників і користувачів, екосистеми повинні забезпечувати відповідне середовище для всіх різних груп. Варіативність артефактів, процесів, інструментів тощо є важливою вимогою для підтримання різноманітності.

Проводячи аналогії екосистем програмного забезпечення та природніх екосистем, також можна виокремити загальні характеристики екосистеми (табл. 1.9).

Таблиця 1.9 – Загальні характеристики екосистеми

Біологічні системи	Системи програмного забезпечення
Харчові зв’язки	Ціннісні зв’язки
Розглядаються у своїх природніх межах	Розглядаються у межах моделювання взаємопов’язаних компонентів
Онтогенез	Вдосконалення (доповнення) системи
Природній добір	Умови ринку

Власне термін software ecosystem (екосистема програмного забезпечення) використовується для відображення надзвичайно різноманітних екосистем програмного забезпечення, хоча всі вони є насправді екосистемами інженерії програмного забезпечення. Розуміння самого терміну «екосистема інженерії програмного забезпечення» розкрито у праці [24], в якій було вперше було запроваджено термін «software engineering ecosystems» (екосистема інженерії програмного забезпечення). Ключовим фактором для розгляду екосистеми інженерії програмного забезпечення є ландшафт. Тобто в контексті інженерії програмного забезпечення термін «ландшафт» є свого роду метафорою, що використовується для виокремлення систем програмного забезпечення.

1.4 Порівняльний аналіз існуючих наукових досліджень засобів для моделювання екосистем інженерії програмного забезпечення

Результатом використання методу Systematic Mapping Study є 4 джерела, враховуючи їх актуальність, завершеність, доцільність і структурованість. Саме ці чинники стали вирішальними при прийнятті рішення щодо подальшого опрацювання та порівняння цих праць.

Вчені [13] запропонували модель екосистеми програмного забезпечення з відкритим кодом. В роботі було зосереджено увагу на певному типі екосистем програмного забезпечення, тобто тих, які побудовані

навколо ініціативи програмного забезпечення з відкритим вихідним кодом (OSS) (наприклад, екосистеми Android, Gnome та Eclipse), а саме екосистеми OSS. Було визначено три основні якості в екосистемах OSS:

− перший вимір – це якість програмної платформи, на якій будуються проєкти екосистеми (наприклад, екосистема Android надає платформу Android, яку використовують усі Android мобільні додатки);

− другий вимір – це якість спільнот OSS, які розвиваються всередині екосистеми та проєктів екосистеми (наприклад, сама спільнота Gnome, тобто спільнота платформи, а також спільноти проєктів, які належать до екосистеми, такі як спільноти Anjuta, Banshee та Abi Word);

− третій вимір якості властивий самим екосистемам, тобто якість, що випливає з розуміння екосистеми OSS як мережі взаємопов’язаних елементів (наприклад, кількість плагінів Eclipse та залежності між ними можна використовувати для оцінки взаємозв'язок екосистеми).

Оцінка якості екосистем OSS має життєво важливе значення, оскільки забезпечення якості – це спосіб запобігти неправильним рішенням, уникнути проблем, а також дозволяє перевірити відповідність вимогам і бізнес-цілям. Його також можна використовувати для якісного систематичного моніторингу для забезпечення зворотного зв’язку та виконання превентивних дій. Наприклад, перед тим, як прийняти рішення про інтеграцію проєкту у встановлену екосистему OSS, важливо провести якісне оцінювання якості, аби уникнути таких проблем, як неактивні спільноти користувачів, низький рівень згуртованості спільноти або навіть проблеми синергетичної еволюції, тобто відсутність співпраці між ключовими розробниками.

Аналіз показав, що наразі в літературі немає жодної моделі якості для екосистем OSS, за винятком деяких показників, розподілених між багатьма документами.

Щоб заповнити цю прогалину, вченими [13] було представлено QuESo – модель якості для оцінки якості екосистем OSS. Щоб отримати цю модель якості, по-перше, було проведено пошук в літературі всіх доступних

показників, що пов’язані з екосистемами OSS, по-друге, розроблено модель якості, використовуючи як стратегію «знизу вгору», класифікуючи знайдені показники, так і стратегію «зверху вниз». Наприклад, для оцінки активності спільноти можна порахувати кількість змін у вихідному репозиторії або кількість повідомлень у списках розсилки за останній період часу.

Модель “QuESo QUALITY MODEL” складається з двох типів взаємопов’язаних елементів: характеристик якості та продуктивності. Характеристики якості відповідають атрибутам екосистеми програмного забезпечення з відкритим кодом, які вважаються релевантними для оцінювання.

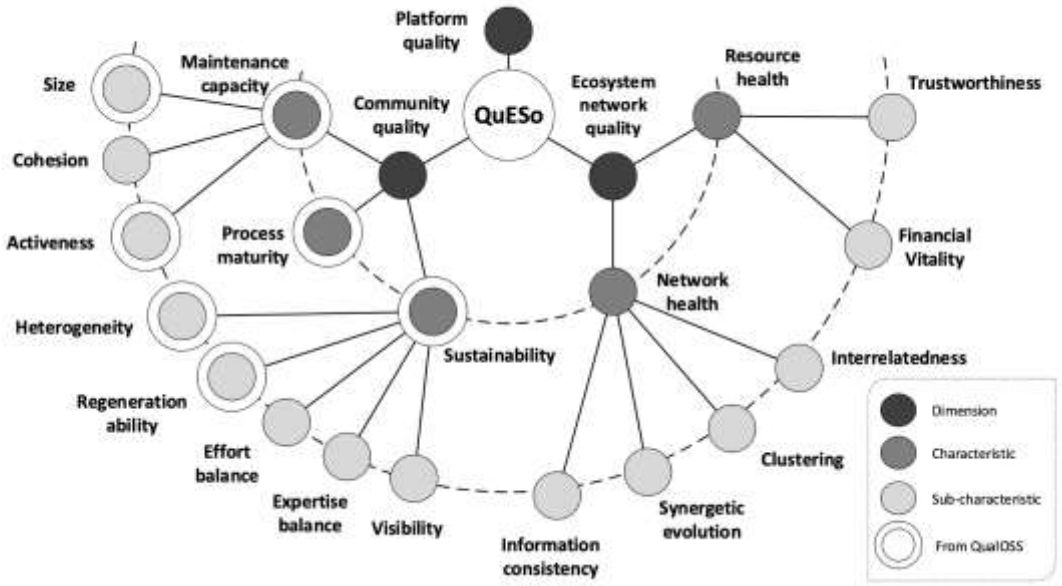

Рисунок 1.7 – QuESo QUALITY MODEL [13]

Цю модель якості можна використовувати як відправну точку для оцінки екосистеми OSS.

Інші вчені [12] представили інструмент моделювання ECOS, що його було розроблено відповідно до клієнт-серверної архітектури, яка є веб-додатком для моделювання SECO з нотацією SSN, призначеної для усунення одного з недоліків у літературі, який полягає у відсутності належного середовища та підтримки для моделювання SECO. Технології Vue.js і Mxgraph.js були використані в розробках інструменту. Його основні функції:

створювати модель SECO з використанням нотації SSN, функції редагування моделі (змінити розмір, перемістити, копіювати, вставити, видалити, згрупувати), зберегти/експортувати модель у форматах зображень (PNG, SVG), експортувати модель у формати, які дозволяють імпортувати модель в інструмент для можливих змін та/або обслуговування (XML, JSON) або в інших інструментах для можливої інтеграції та друку.

На рис. 1.8 представлено головний і унікальний екран ECOS інструменту моделювання, запропонований із мінімалістичним і простим дизайном, аби зробити його максимально легким для користувача і зрозумілим під час використання.

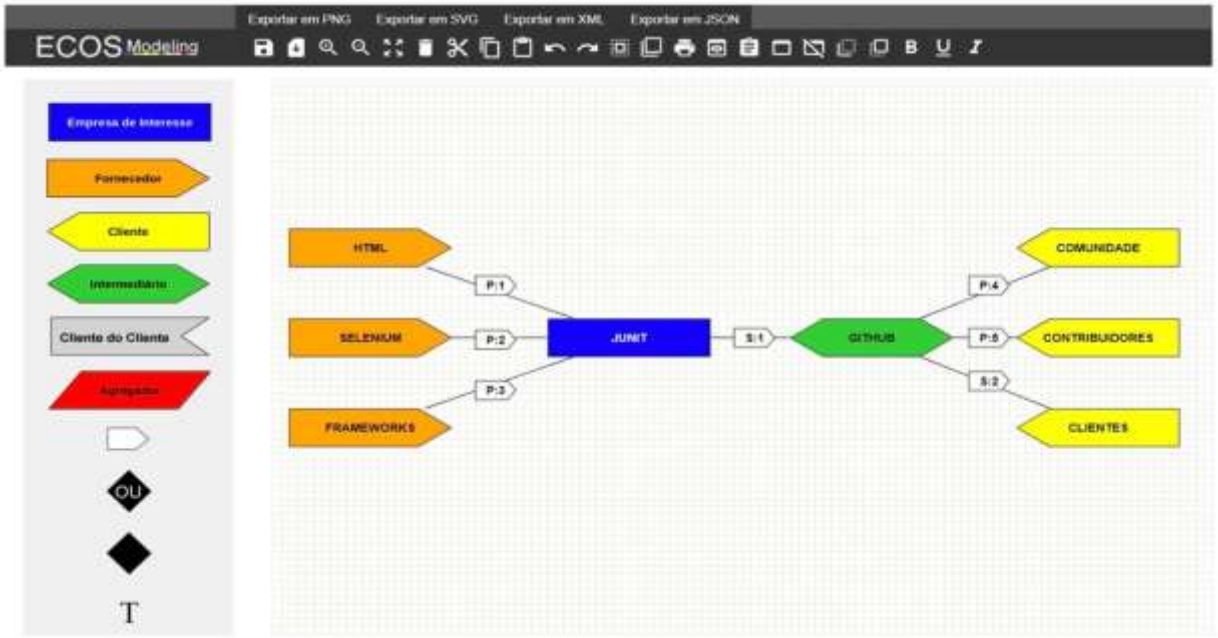

Рисунок 1.8 – Головний екран ECOS

Головний екран має верхнє меню з піктограми, які дозволяють користувачеві маніпулювати компонентами та створеною моделлю SECO, на додаток до опцій для експорту моделі. Меню зліва містить компоненти нотації SSN. У центральній області створюються моделі за допомогою SSN позначення. Можливість натискати та перетягувати компоненти полегшує моделювання, редагування та обслуговування моделі. Визначення зв’язків між компонентами також можливо. Немає жодних обмежень кількості елементів моделі та ступеня деталізації – це рішення моделіста.

В роботі [15] вчені з’ясовують, що саме потрібно моделювати в контексті SECO та яким чином на прикладі екосистем Google Anroid та Apple iOS. Основною метою обох екосистем є надання програмної платформи, а також додаткових програмних додатків і послуг для ринку. Таким чином, компанії Google і Apple створили мережу співавторів (або партнерів), що складається з розробників додатків, компаній-розробників програмного забезпечення та постачальників контенту. У той час як Google і Apple розробляють ключову програмну платформу (тобто мобільну операційну систему), інші розробники додатків і компанії, що займаються розробленням програмного забезпечення, надають додаткові програми та послуги для відповідних операційних систем. Крім того, постачальники контенту надають такий контент, як дані, музика та ігри. Google і Apple роблять програми та вміст, розроблені зовнішніми сторонами, доступними для ринку через онлайн-магазини App Store і Google Play.

Вчені [15] в контексті розроблення екосистем програмного забезпечення чітко виокремлюють кроки, які необхідно виконати для моделювання екосистеми програмного забезпечення. Передусім виокремлено набір вимог для опису SECO:

− учасники (Collaborator) – визначення членів екосистеми програмного забезпечення та їхніх ролей;

− взаємодії (Interaction) – виявлення стосунків між членами SECO;

− зона відповідальності – визначення ресурсів, діяльності та зобов'язань учасників (акторів).

Набір аналітичних вимог до екосистем програмного забезпечення:

− аналіз стимулів і мотивацій учасників;

− аналіз довіри та надійності;

− аналіз для розподілу та децентралізації обов'язків і ресурсів.

Існує декілька технік для моделювання SECO, зокрема – техніка i* моделювання (рис. 1.9).

i* - загальна техніка соціального моделювання, яка описує відносини між

акторами з різних ділових, технічних та організаційних точок зору. i* пояснює відносини між членами екосистеми програмного забезпечення в термінах стратегічної залежності між стратегічними акторами. Модель може бути використана для пояснення цілей і міркувань учасників щодо розроблення або приєднання до екосистеми програмного забезпечення.

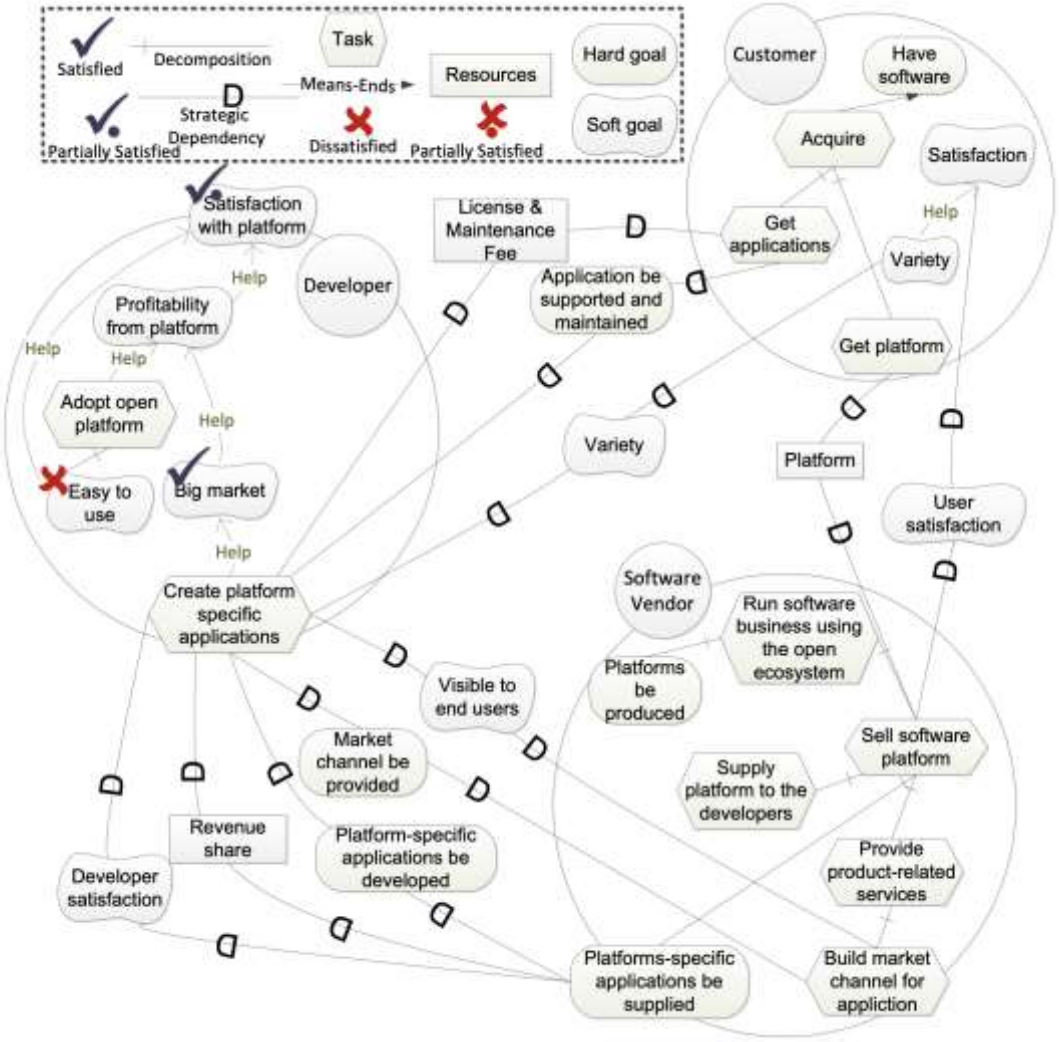

Рисунок 1.9 – Приклад SECO моделювання з використанням i*

В i* стратегічні залежності вказують на те, що один актор покладається на іншого актора задля досягнення мети, виконання завдання або надання ресурсів. Різні типи відкритих, відданих і критичних стратегічних залежностей пояснюють різні ступені контролю та вразливості залежного у відносинах між двома акторами. Крім того, як показує вибіркова оцінка цілей, хоча платформа непроста у використанні, вона адекватно задовольняє

розробників приєднатися до екосистеми програмного забезпечення, оскільки має великий ринок.

1.5 Аналіз вимог до методу побудови засобу для моделювання екосистем інженерії програмного забезпечення

Для успішного розроблення засобу для моделювання екосистем інженерії програмного забезпечення потрібно скласти нефункціональні та функціональні вимоги до даного забезпечення.

Маючи на меті розробити успішний продукт, особливу увагу слід звернути на вимоги. Вони повинні відповідати таким критеріям:

− атомарність. Відповідно до цього критерія вимоги повинні бути описані так, аби їх не можна було уточнити ще детальніше або розбити одну вимогу на декілька;

− завершеність. Вимоги до одного функціоналу повинні бути описані в одному пункті специфікації. Не можна допускати ситуацію, коли один і той самий функціонал описаний в різних частинах документа;

− послідовність. Вимога не повинна використовувати інші вимоги та обмеження системи;

− відстеження (трасування). Критерій, який дозволяє зрозуміти, чому було прописано саме таку вимогу;

− актуальність. Критерій, який перевіряє відповідність вимогам до сучасних реалій (наприклад, із законодавчого/технічного боку або з боку зручності використання продукту, актуальність його користувацького інтерфейсу);

− здійсненність. Перевірка на те, що вимоги можливо реалізувати за допомогою актуальних існуючих на даний момент технологій. зрозумілість (доступність);

− верифікованість. Критерій, за якими всі, можна порівняти готовий продукт із вимогою і перевірити, чи виконано її. Якщо вимога описана дуже розмито, перевірити її не вийде;

− обов'язковість: Визначення того, наскільки важливе та пріоритетне виконання даної вимоги. Усім вимогам у специфікації повинні бути призначені важливість, стабільність та перевага. За допомогою цього вибираються першочергові значення для виконання командою розробки завдання.

1.5.1 Розроблення не функціональних вимог до засобу для моделювання екосистем інженерії програмного забезпечення

Враховуючи перелічені пункти, можемо виокремити критерії, яким повинні відповідати нефункціональні вимоги:

- структурованість
- детальність
- повнота
- простота.

1.5.2 Розроблення функціональних вимог до засобу для моделювання екосистем інженерії програмного забезпечення

Основною метою даного дослідження є створення засобу для моделювання екосистем інженерії програмного забезпечення, що повинно мати наступний функціонал:

– можливість візуалізації складових екосистем інженерії програмного забезпечення;

– можливість редагувати отриману візуалізацію;

– зберігати готову роботу у форматі .jpeg;

– візуалізація різних взаємозв’язків між компонентами;

– можливість виокремити певні групи компонентів;

– зміна розмірів складових екосистеми в їх візуальному відображені;

– редагування наявних компонентів.

Залежно від обраного типу моделювання екосистеми інженерії програмного забезпечення система автоматично повинна надавати той чи

інший набір інструментів. Під час роботи з проєктоцентричним типом з фокусуванням на екосистему інструментом повинні бути елементи (nodes), що в групі представляють собою - Project Dependency Map.

Натомість під час роботи з проєктоцентричним типом з екосистемою в контексті ми повинні надати можливість виокремлювати роль певного проєкту в контексті всієї екосистеми.

При моделюванні екосистеми з аспектом, орієнтованим на розробника з фокусуванням на екосистему засіб повинен надати можливість для моделювання відносин між учасниками екосистеми.

1.6 Висновки до розділу 1

В даному розділі було розглянуто екосистему програмного забезпечення, вимоги до нею. На основі аналізу наукових праць вчених, що піднімають питання взаємозв’язків SECO між собою, визначено SECO, виокремлено основні ролі в моделі екосистеми програмного забезпечення, підкреслено проблематику моделювання екосистем програмної інженерії, одна з яких полягає у відсутності стандартизації моделювання SECO. Також розглянуто мета-моделі для побудови екосистем програмного забезпечення та основні вимоги, яким мета-модель повинна відповідати. З’ясовано життєвий спосіб SECO та рівні життєвого способу. Встановлено, що залишається відкритим питання засобів моделювання для різних аспектів екосистем SECO. В роботі дістав подальшого розвитку термін «екосистема інженерії програмного забезпечення» як новітня галузь науки.

2 ПРОЄКТУВАННЯ СИСТЕМИ

2.1 Модель програмного забезпечення для візуалізації екосистем інженерії програмного забезпечення

Програмне забезпечення представляє собою Single Page Application, що дозволяє візуалізувати екосистему інженерії програмного забезпечення.

Безпосередньо користувач даного засобу, web-застосунку, матиме змогу моделювати екосистему на тому чи іншому рівні, а саме:

- рівні взаємодії акторів екосистеми;
- рівні екосистеми програмного забезпечення;
- рівні взаємодії екосистем програмного забезпечення.

Окрім можливості моделювання екосистем користувач даного програмного забезпечення повинен мати змогу виконати наступні кроки, які є необхідними для моделювання екосистеми інженерії програмного забезпечення:

- визначення членів екосистеми програмного забезпечення та їхніх ролей;
- візуалізація стосунків між членами SECO;
- визначити ресурси та зобов'язань учасників (акторів).

Також слід зазначити, що залежно від обраного типу моделювання екосистеми користувачеві буде доступний різний набір інструментів для роботи.

Окрему увагу в моделі слід звернути на те, що засіб представляє собою web-застосунок. Це тягне за собою цілу низку переваг:

– відсутність необхідності інсталяції програмного забезпечення на певний пристрій, а й отже кросплатформеність, тобто можливість працювати на будь-якому пристрої чи операційній системі;

– необхідність оновлювати програмне забезпечення безпосереднім користувачем;

– відсутність вимог до пристрою, на якому буде відбуватися робота, оскільки для взаємодії достатньо лише браузера.

Для полегшення імплементації програмного забезпечення розробимо модель майбутнього засобу. Опишемо функціонал та базові властивості основних класіі.

Клас FlowCanvas відповідає за відображення всього наявного функціоналу та містить основні методи для взаємодії з програмним забезпеченням.Відображення наявних елементів, а й отже маніпуляція з DOM деревом відбувається в межах певного блоку, що містить в собі такі компоненти як: Layout, SideBar,MiniMap, Controls та FlowRender. Для ініціалізації основних елементів для побудови екосистеми використовується метод useMemo в якому передаємо об’єкт значеннями якого є класи відповідних типів нод. Основні методи та їхнє призначення наведено в табл. 2.1.

Таблиця 2.1 – Основні методи та їхнє призначення

Назва методу	Призначення
onConnect	відповідає за встановлення з’єднань між елементами
onDragOver	переміщення певних елементів (нод) на екран
setNodes	актуалізація наявних елементів при додаванні нових чи видаленні вже існуючих
onDrop	візуалізація додавання нових елементів

Допоміжні класи та їх функціонал наведено у табл. 2.2.

Таблиця 2.2 – Допоміжні класи та їх функціонал

Назва класу	Призначення
1	2
CustomEdge	з’єднання елементів між собою. Візуалізація штрих-пункти
ConnectionLine	з’єднання елементів між собою. Візуалізація пряма лінія

Кінець таблиці 2.2

1	2
DecisionNode	Відображення певного елементу в екосистемі з певними властивостями та їх зміною
CircleNode	Відображення певного елементу в екосистемі з певними властивостями та їх зміною
ReactangleNode	Відображення певного елементу в екосистемі з певними властивостями та їх зміною

Наведемо загальну діаграму класів майбутнього програмного забезпечення (рис. 2.1).

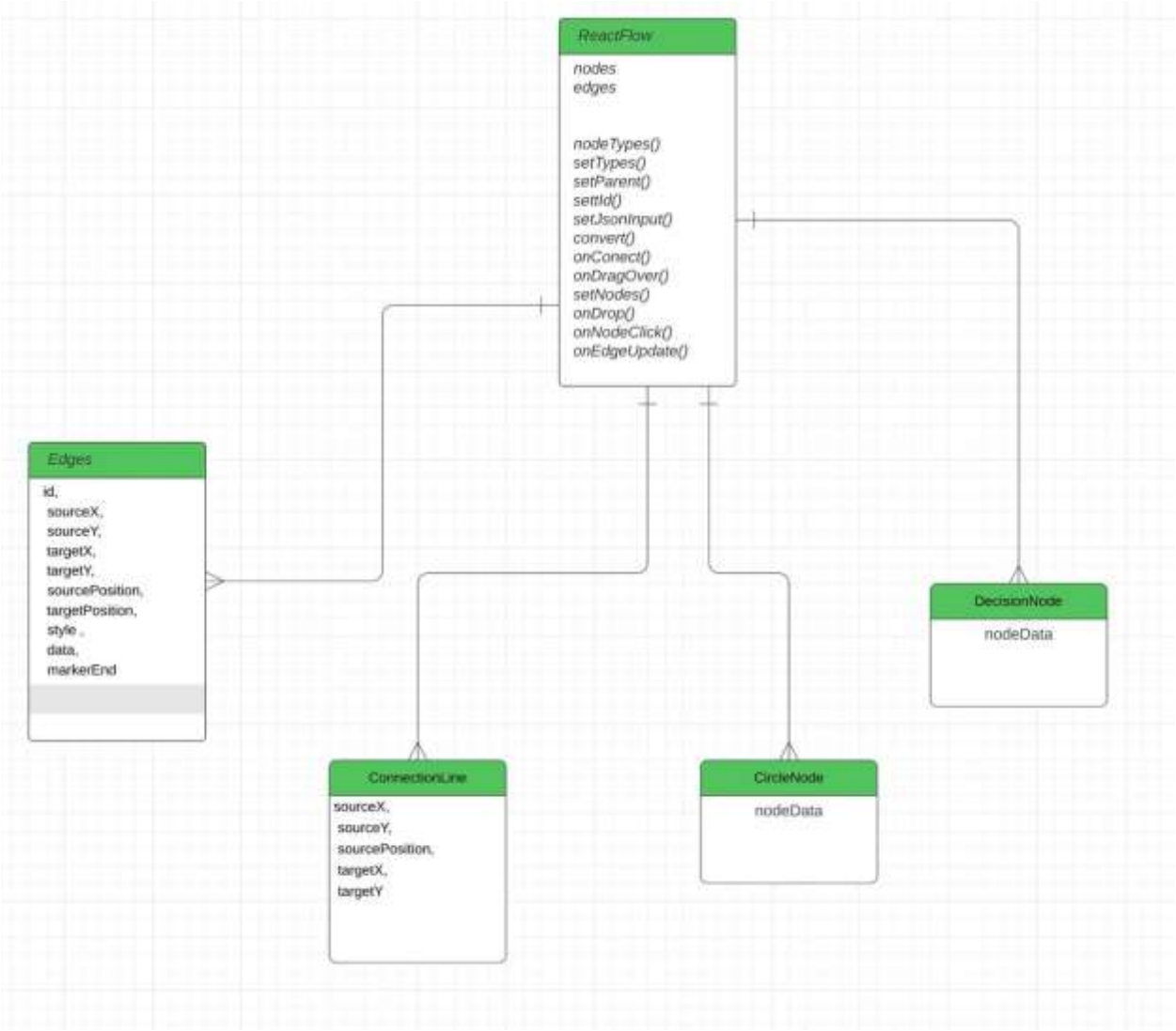

Рисунок 2.1 – Загальна діаграма класів майбутнього програмного забезпечення

Для реалізації функціоналу, що дозволяє імпортувати отриману

візуалізацію використовується Blob - Binary Large Object (можна додоати в словник).

Blob означає «великий бінарний об’єкт» і є представленням фрагмента байтів. Веб-браузери реалізують об’єкт Blob, який відповідає за зберігання даних.

Blob має свій розмір і тип MIME (Multipurpose Internet Mail Extensions Багатоцільові розши́рення для інтерне́т-по́шти), як і файл. Дані Blob зберігаються в пам’яті або файловій системі залежно від браузера та розміру Blob.

2.2 Архітектура застосунку

Маючи на меті розроблення програмного забезпечення, особливу уваги слід приділяти архітектурі застосунку. Правильно побудована архітектура дозволить розробити ефективну, високопродуктивну, гнучку систему, яку з часом можна буде розширювати новим функціоналом з мінімальними зусиллями. Наведемо критерії, яким буде відповідати програмне забезпечення:

– ефективність системи. Насамперед програма повинна вирішувати поставлені завдання та добре виконувати свої функції, причому у різних умовах. Сюди можна зарахувати такі характеристики, як надійність, безпека, продуктивність, здатність справлятися зі збільшенням навантаження (масштабованість);

– гнучкість системи. Будь-який додаток доводиться змінювати з часом – змінюються вимоги, додаються нові. Що швидше і зручніше можна внести зміни до існуючого функціоналу, що менше проблем і помилок це викличе – то гнучкіша і конкурентоспроможніша система;

– розширюваність системи. Можливість додавати до системи нові сутності та функції, не порушуючи її основної структури. На початковому етапі в систему має сенс закладати лише основний та найнеобхідніший

функціонал, але при цьому архітектура повинна дозволяти легко нарощувати додатковий функціонал у міру потреби. Причому так, щоб внесення найімовірніших змін вимагало найменших зусиль.

Розглянемо модульну архітектуру для побудови програмного забезпечення, зважаючи її використання для побудови засобу для моделювання екосистем інженерії програмного забезпечення.

Незважаючи на різноманітність критеріїв, головною при розробленні великих систем вважається завдання зниження складності. Для зниження складності найбільш доцільно використовувати поділ на частини. Іноді це називають принципом «розділяй і владарюй» (divide et impera), але по суті йдеться про ієрархічну декомпозицію. Складна система повинна будуватися з невеликої кількості простіших підсистем, кожна з яких, у свою чергу, будується з частин меншого розміру тощо, доки найменші частини не будуть досить прості для безпосереднього розуміння і створення.

Особливість полягає в тому, що це рішення є не тільки єдиним, але й універсальним. Крім зниження складності, воно одночасно забезпечує гнучкість системи, дає хороші можливості для масштабування, а також дозволяє підвищувати стійкість за рахунок дублювання критично важливих частин.

Відповідно, коли йдеться про побудову архітектури програми, створення її структури, під цим передусім маємо на увазі декомпозицію програми на підсистеми (функціональні модулі, сервіси, шари, підпрограми) та організація їхньої взаємодії один з одним та зовнішнім світом. Причому що незалежніші підсистеми, то безпечніше зосередитися на розробленні кожної з них окремо в конкретний момент часу і при цьому не дбати про всі інші частини. У цьому випадку програма зі «спагетті-коду» перетворюється на конструктор, що складається з набору модулів/підпрограм, що взаємодіють один з одним за добре визначеними та простими правилами, що власне і дозволяє контролювати її складність, а також дає можливість отримати всі ті переваги, які зазвичай співвідносяться з поняттям «хороша

архітектура»:

- масштабованість (Scalability) – можливість розширювати систему та збільшувати її продуктивність, за рахунок додавання нових модулів;
- ремонтопридатність (Maintainability) – зміна одного модуля не потребує зміни інших модулів;
- замінність модулів (Swappability) – модуль легко замінити на інший;
- можливість тестування (Unit Testing) – модуль можна від'єднати від решти і протестувати / полагодити;
- перевикористання (Reusability) – модуль може бути перевикористаний в інших програмах та іншому оточенні;
- супроводжуваність (Maintenance) – розбиту на модулі програму легше розуміти та супроводжувати.

Для побудови архітектури програмного забезпечення необхідно правильно виконати декомпозицію модулів. Можливі варіанти декомпозиції модулів такі:

– ієрархічна. Декомпозицію треба проводити ієрархічно – спочатку систему розбивають на великі функціональні модулі/підсистеми, що описують її роботу в загальному вигляді. Потім отримані модулі аналізують детальніше і ділять на підмодулі чи об'єкти. Виділенню об’єктів передує розділення системи на основні смислові блоки. Для невеликих додатків двох рівнів ієрархії часто виявляється достатньо – система спочатку ділиться на підсистеми / пакети, а пакети діляться на класи. Прикладом цього є добре відомий шаблон MVC (модель відображення контролер). Особливість його полягає у відділенні уявлення від бізнес-логіки, тобто в тому, що будь-який додаток спочатку ділиться на два модулі – один з яких відповідає за реалізацію власне самої бізнес логіки (Модель), а другий – за взаємодію з користувачем (Інтерфейс користувача або Подання) . Для того, аби ці модулі могли розроблятися незалежно, зв'язок між ними послаблюється за допомогою патерна «Спостерігач» і ми фактично отримуємо один із найпотужніших і найпопулярніших «шаблонів», які використовуються в

даний час;

– функціональна. Розподіл на модулі/підсистеми найкраще проводити з тих завдань, які вирішує система. Основне завдання розбивається на складові підзавдання, які можуть вирішуватися/ виконуватися незалежно один від одного. Кожен модуль повинен відповідати за вирішення якоїсь підзадачі та виконувати відповідну їй функцію. Крім функціонального призначення модуль характеризується також набором даних, необхідних для виконання його функції. Причому бажано, аби свою функцію модуль міг виконати самостійно, без допомоги інших модулів лише на основі своїх вхідних даних. Модуль – це не довільний шматок коду, а окрема функціонально осмислена та закінчена програмна одиниця (підпрограма), яка забезпечує вирішення деякого завдання і в ідеалі може працювати самостійно або в іншому оточенні та бути перевикористовуваною. Отже, правильна декомпозиція грунтується, передусім, на аналізі функцій системи та необхідні виконання цих функцій даних;

– декомпозиція на основі високого зчеплення всередині модуля та низької зв’язності модулей між собою (High Cohesion + Low Coupling). Найголовнішим критерієм якості декомпозиції є те, наскільки модулі сфокусовані на вирішення своїх завдань і незалежні. Зазвичай це формулюють так: «Модулі, отримані в результаті декомпозиції, повинні бути максимально пов'язані всередині (high internal cohesion) і мінімально пов'язані один з одним (low external coupling)».

High Cohesion, висока сполученість або «згуртованість» усередині модуля, говорить про те, що модуль сфокусований на вирішенні однієї вузької проблеми, а не займається виконанням різнорідних функцій або непов'язаних між собою обов'язків (спряженість – cohesion, характеризує ступінь, в якій завдання, що виконуються модулем, пов'язані один з одним). Наслідком High Cohesion є принцип єдиної відповідальності (Single Responsibility Principle – перший із п'яти принципів SOLID), згідно з яким

будь-який об'єкт/модуль повинен мати лише один обов'язок і, відповідно, не повинно бути більше однієї причини для його зміни.

Low Coupling, слабка пов'язаність, означає, що модулі, на які розбивається система, повинні бути, по можливості, незалежні або слабко пов'язані один з одним. Вони повинні мати можливість взаємодіяти, але при цьому якнайменше знати один про одного (принцип мінімального знання).
Це означає, що при правильному проєктуванні, при зміні одного модуля не доведеться правити інші або ці зміни будуть мінімальними. Чим слабша пов'язаність, тим легше писати/розуміти/розширювати/лагодити програму.

Якісно спроєктовані модулі матимуть такі властивості:

– функціональна цілісність і завершеність – кожен модуль реалізує одну функцію, але реалізує добре та повністю; модуль самостійно (без допомоги додаткових коштів) виконує повний набір операцій реалізації своєї функції;

– один вхід та один вихід – на вході програмний модуль отримує певний набір вихідних даних, виконує змістовну обробку та повертає один набір результатних даних, тобто. реалізується стандартний принцип IPO - вхід-процес-вихід;

– логічна незалежність – результат роботи програмного модуля залежить від вихідних даних, але з залежить від роботи інших модулів;

– слабкі інформаційні зв'язки з іншими модулями – обмін інформацією між модулями має бути, по можливості, мінімізований.

Коректно зроблена декомпозиція – головна проблема для багатьох програмістів. Якщо виділені модулі виявляються сильно зчеплені один з одним, якщо їх не вдається розробляти незалежно або не зрозуміло, за яку конкретно функцію кожен з них відповідає, то варто задуматися, а чи взагалі виробляється поділ. Має бути зрозуміло, яку роль виконує кожен модуль. Найнадійніший критерій того, що декомпозиція робиться правильно, це якщо модулі виходять самостійними і цінними власними силами підпрограмами, які можуть бути використані у відриві від решти програми (а значить,

можуть бути перевикористовувані).

Насамперед слід, звичайно ж, прагнути того, щоб модулі були гранично автономні – це є ключовим параметром правильної декомпозиції. Тому проводити її потрібно таким чином, щоб модулі спочатку слабко залежали один від одного. Крім того, є ряд спеціальних технік і шаблонів, які дозволяють додатково мінімізувати і послабити зв'язки між підсистемами. Наприклад, у випадку MVC для цієї мети використовувався шаблон «Спостерігач», але можливі інші рішення. Техніки зменшення пов'язаності становлять основний «інструментарій архітектора». Мова йде про всі підсистеми і послаблювати пов'язаність потрібно на всіх рівнях ієрархії, тобто не тільки між класами, а й між модулями на кожному ієрархічному рівні.

Враховуючи це, розглянемо основні методи для послаблення зв’язності між модулями. Найбільш розповсюджені методи виокремлено такі:

а) інтерфейси. Фасад. Головним, що дозволяє зменшувати пов'язаність системи, є звичайно ж інтерфейси (і принцип Інкапсуляція + Абстракція + Поліморфізм):

1) модулі мають бути один для одного «чорними ящиками» (інкапсуляція). Це означає, що один модуль не повинен «лізти» всередину іншого модуля і що-небудь знати про його внутрішню структуру. Об'єкти однієї підсистеми не повинні безпосередньо звертатися до об'єктів іншої підсистеми;

2) модулі/підсистеми повинні взаємодіяти один з одним лише за допомогою інтерфейсів (тобто абстракцій, що не залежать від деталей реалізації) Відповідно, кожен модуль повинен мати чітко визначений інтерфейс або інтерфейси для взаємодії з іншими модулями.

Принцип «чорної скриньки» (інкапсуляція) дозволяє розглядати структуру кожної підсистеми незалежно від інших підсистем. Модуль, що є чорним ящиком, можна відносно вільно змінювати. Проблеми можуть виникнути лише на стику різних модулів (або модуля та оточення). І ось цю

взаємодію потрібно описувати у максимально загальній (абстрактній) формі – у формі інтерфейсу. У цьому випадку код працюватиме однаково з будь-якою реалізацією, що відповідає контракту інтерфейсу. Саме ця можливість працювати з різними реалізаціями (модулями або об'єктами) через уніфікований інтерфейс і називається поліморфізмом. Поліморфізм це не перевизначення методів, як іноді помилково вважають, а насамперед взаємозамінність модулів/об'єктів з однаковим інтерфейсом, або «один інтерфейс, безліч реалізацій»

Завдяки інтерфейсам та поліморфізму якраз і досягається можливість модифікувати та розширювати код, без зміни того, що вже написано (Open-Closed Principle). Доки взаємодія модулів описана виключно у вигляді інтерфейсів, і не зав'язана на конкретні реалізації, ми маємо можливість абсолютно «безболісно» для системи замінити один модуль на будь-який інший, що реалізує той самий інтерфейс, а також додати новий і тим самим розширити функціональність. Це як у конструкторі або «плагінній архітектурі» (plugin architecture) – інтерфейс служить свого роду конектором, куди може бути підключений будь-який модуль із відповідним роз'ємом. Гнучкість конструктора забезпечується тим, що можна просто замінити одні модулі/деталі на інші, з такими ж роз'ємами (з тим же інтерфейсом), а також додати скільки завгодно нових деталей (при цьому вже існуючі деталі ніяк не змінюються і не переробляються) .

Інтерфейси дозволяють будувати систему вищого рівня, розглядаючи кожну підсистему як єдине ціле та ігноруючи її внутрішній пристрій. Вони дають можливість модулям взаємодіяти і при цьому нічого не знати про внутрішню структуру один одного, тим самим повністю реалізуючи принцип мінімального знання, що є основою слабкої зв'язаності. Причому, що у більш загальної/абстрактної формі визначені інтерфейси і що менше обмежень вони накладають взаємодію, то гнучкішою є система. Звідси фактично випливає ще один із принципів SOLID – Принцип поділу інтерфейсу (Interface Segregation Principle). Отже, коли взаємодія та залежності модулів

описуються лише за допомогою інтерфейсів, тобто абстракцій, без використання знань про їхній внутрішній устрій та структуру, то фактично цим реалізується інкапсуляція. З цього випливає, що концепція інтерфейсів включає і в певному сенсі узагальнює майже всі основні принципи ООП - Інкапсуляцію, Абстракцію, Поліморфізм. Однак слід зауважити той факт, що коли проєктування йде не на рівні об’єктів, а на рівні модулів, то реалізацією інтерфейса модуля представляє собою спеціальний об’єкт – Фасад.

Фасад – це об'єкт-інтерфейс, що акумулює в собі високорівневий набір операцій для роботи з деякою підсистемою, що приховує її внутрішню структуру і справжню складність. Забезпечує захист від змін у реалізації підсистеми. Служить єдиною точкою входу;

б) dependency Inversion. Коректне створення та отримання залежностей. Суть цього підходу є доволі простою – всі залежності повинні бути у вигляді інтерфейсів.

При проєктуванні модуля мають бути визначені такі ключові речі:

– що модуль робить, яку функцію виконує;

– що модулю потрібно від його оточення, тобто з якими об'єктами/модулями йому доведеться мати справу;

– як він це отримуватиме.

Вкрай важливо те, як модуль отримує посилання на об'єкти, які він використовує у своїй роботі. І тут можливі такі варіанти:

- модуль сам створює об'єкти, необхідні йому для роботи. Але модуль не може це зробити безпосередньо – для створення необхідно викликати конструктор конкретного типу, і в результаті модуль залежатиме не від інтерфейсу, а від конкретної реалізації. Вирішити проблему в даному випадку дозволяє шаблон Фабричний Метод (Factory Method). Суть полягає в тому, що замість безпосереднього інстанціювання об'єкта, ми надаємо класу-клієнту деякий інтерфейс для створення об'єктів. Оскільки такий інтерфейс при правильному дизайні завжди може бути перевизначений, ми отримуємо певну гнучкість при використанні низькорівневих модулів у модулях

високого рівня". У випадках, коли потрібно створювати групи або сімейства взаємопов'язаних об'єктів, замість Фабричного Методу використовується абстрактна фабрика (Abstract factory);

- модуль бере необхідні об'єкти у того, у кого вони вже є (зазвичай це деякий, відомий усім репозиторій, в якому вже лежить усе, що тільки може знадобитися для роботи програми). Цей підхід реалізується шаблоном Локатор Сервісів (Service Locator), основна ідея якого полягає в тому, що в програмі є об'єкт, який знає, як отримати всі залежності (сервіси), які можуть знадобитися. Головна відмінність від фабрик в тому, що Service Locator не створює об'єкти, а фактично вже містить інстанційовані об'єкти (або знає де/як їх отримати, а якщо і створює, то тільки один раз при першому зверненні). Фабрика при кожному зверненні створює новий об'єкт, який ви отримуєте на повну власність і можете робити з ним що хочете. Локатор же сервісів видає посилання на ті самі, вже існуючі об'єкти. Тому з об'єктами, виданими Service Locator, потрібно бути дуже обережним;

- модуль взагалі не дбатиме про «добування» залежностей. Він лише визначає, що йому потрібно для роботи, а всі необхідні залежності йому поставляються («впорскуються») з кимось іншим. Це і називається – Впровадження Полежностей (Dependency Injection). Зазвичай необхідні залежності передаються або як параметри конструктора (Constructor Injection) або через методи класу (Setter injection). Такий підхід інвертує процес створення залежності – замість самого модуля створення залежностей контролює хтось ззовні. Модуль активного елемента стає пасивним – не він робить, а для нього роблять. Така зміна напряму дії називається Інверсія Контролю (Inversion of Control).

2.3. Бібліотека React.js як інструмент для реалізації клієнтської частини web-застосунку

Оскільки засіб для моделювання екосистем інженерії програмного

забезпечення буде реалізований як web-додаток, доволі важливо обрати зручний інструмент для імплементації програмного забезпечення. Наразі найбільш популярним інструментами для написання клієнтської частини web-сайтів є React.js. React – це одна з найпопулярніших бібліотек для створення зручних інтерфейсів користувача, що допомагає створювати великі вебдодатки, які можуть змінювати дані без перезавантаження сторінки. React зберігає стан програми всередині віртуального DOM та лише повторно відтворює вміст у браузері (маніпулювання DOM), коли стан змінюється. Основна мета – бути швидким, масштабованим і простим. React можна використовувати з комбінацією інших бібліотек або фреймворків JavaScript. Він підходить до створення інтерфейсів користувача, розбиваючи їх на компоненти. Поява React допомогла фронтенд-розробникам створити зручні інтерфейси. Можна сказати, що історія починається в 2010 році зі створення XHP, який надихнув Джордана Волка на створення React.

ReactJS починався як JavaScript-порт XHP, версії PHP, яку Facebook випустив чотири роки тому. XHP головним чином займався мінімізацією атак міжсайтового сценарію (XSS). XSS-атаки спрацьовують, коли зловмисник вводить код, який має на меті завдати шкоди користувачеві цього вмісту. Типовими векторами атак є введення коду за допомогою вбудованого та прихованого JavaScript, а потім використання цього вбудованого JavaScript для викрадення інформації або іншим чином скомпрометувати користувача, який переглядає вміст. Але з XHP була явна проблема: динамічні веб-додатки вимагали багато звернень до сервера, і XHP не вирішив цю проблему. Тому компанією Facebook було вирішено розвинути ідею впровадження XHP у браузер за допомогою JavaScript. Результатом цього є бібліотека React.js.

Однією з основних проблем під час роботи з JavaScript є маніпуляції з DOM-деревом html сторінки. React.js повністю вирішує цю проблему шляхом створення віртуального DOM дерева. ReactJS працює, зберігаючи стан програми всередині та лише повторно відтворюючи вміст у браузері (маніпулювання DOM), коли стан змінюється. Взаємодія з ReactJS,

відбувається за допомогою оповіщення про зміну стану певного компонента.

Головною перевагою веб-додатків React перед статичними веб-сайтами є їх інтерактивність. Інтерфейс програми React реагує на дії користувача змінами в інтерфейсі. Іноді ці зміни відбуваються прямо в компоненті інтерфейсу користувача, з яким взаємодіє користувач. Але іноді вони впливають на кілька компонентів і відображаються в різних частинах програми.

З сотнями дій, які користувач може виконати в інтерфейсі, потрібна хороша система та логіка, щоб керувати динамічним станом програми. Цей процес вимагає передачі даних до кількох компонентів програми, синхронізації між ними та збереження стану програми. Саме для вирішення даної проблеми доцільним буде розглянути Redux, який контролює, коли змінюється певний фрагмент стану програми, звідки надходить певний фрагмент даних і чи є необхідність в певній логіці обробки цієї зміни. Без Redux кожен компонент програми React обробляє дані користувача та керує власним станом. Іноді він ділиться вхідними даними у формі з іншими компонентами, які покладаються на змінені дані. Якщо процес включає лише компоненти, які мають батьківсько-дочірні стосунки з компонентом, який ініціює зміну стану, логіка проста. Однак, якщо стан потрібно надавати спільно з багатьма компонентами, які належать до різних частин структури програми, потік даних стає складнішим. У таких випадках виникає так звана проблема передачі даних ланцюгом від компонента першого рівня до останнього компонента. Даний підхід – це погана практика передачі властивостей через компоненти вниз по ієрархії без їх використання в процесі. З часом це призводить до розширеної кодової бази, де важко відслідковувати ці атрибути. За допомогою Redux кожен компонент надсилає свої дані до глобального сховища та впливає на стан програми, а також прослуховує зміни, ініційовані іншими компонентами. Різниця, яку Redux робить для потоків даних у додатку, проілюстрована на діаграмі нижче.

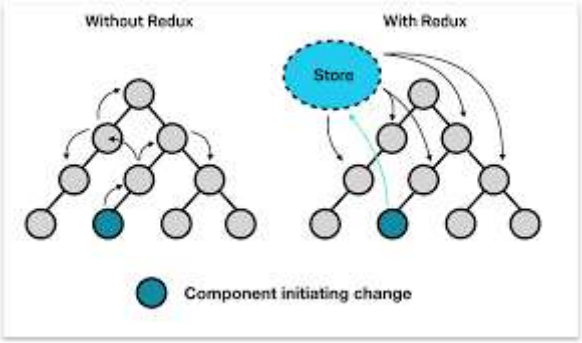

Рисунок 2.2 – Візуалізація передачі даних в компонентах React

Важливим інструментом для написання клієнтської частини вебдодатків є і фреймворк Vue.js. Фреймворк (framework) – це деякий каркас до створення додатків певною мовою програмування. Він включає набір бібліотек, які дозволяють значно спростити і прискорити розроблення додатків. Також мета фреймворку – це надання такого середовища розробки, яке дозволить доопрацьовувати та розширювати функціонал проєкту. Фреймворк використовує певну архітектуру програми для того, щоб проєкт був розділений на логічні частини, модулі. Наприклад, схема поділу Model-View-Controller (MVC) має на увазі поділ на три сегменти: модель, відображення та контролер, кожен з яких можна змінювати незалежно від інших.

Таким чином, спираючись результати аналізу, складено порівняльну таблицю між бібліотекою React.js та фреймворком Vue.js (табл. 2.3).

Таблиця 2.3 – Порівняльна таблиця бібліотеки React.js та фреймворком Vue.js. за окремими параметрами

Параметр	React.js	Vue.js
Тип	бібліотека	фреймвок
мова програмування	JavaScript\TypeScript	JavaScript
Архітектура	MVC	MVC
Робота з DOM	Віртуальний DOM	Віртуальний DOM
Синтаксис	JSX	HTML\JSX
Маштабованість	Більш масштабований	Менш масштабований

Оскільки React.js є бібліотекою, що не накладає жодних обмежень на

написання коду, а також має кращу масштабованість й ширшу екосистему, в роботі було прийнято рішення використовувати саме бібліотеку React.js.

2.4 Висновки до розділу 2

В даному розділі було обрано архітектуру для програмного забезпечення та інструмент для реалізації засобу для моделювання. Ця архітектура допомагає реалізувати ефективну, гнучку, масштабовану систему. Структура програмної системи є декомпозованою на підсистеми (функціональні модулі, сервіси, шари, підпрограми) та організовано їхню взаємодію один з одним і з кінцевим користувачем. Для безпосередньої імплементації засобу для моделювання екосистем інженерії програмного забезпечення було обрано бібліотеку React.js.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАСОБУ ДЛЯ МОДЕЛЮВАННЯ ЕКОСИСТЕМ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Розроблення архітектури засобу

Після проведеного огляду існуючих інструментів для створення клієнтської частини web-застосунків та огляду архітектурних рішень, що використовуються для написання програмного забезпечення, здійснимо безпосередню імплементацію засобу для моделювання екосистем інженерії програмного забезпечення. В рамках даного розділу буде реалізовано програмне забезпечення та наведено приклад його роботи.

Основна задача перед реалізації архітектури будь-якого застосунку – це правильна декомпозиція модулей. Виділимо три основні складові даного застосунку, а саме:

- модуль (Module);
- сховища;
- сервіси (Service).

З цього випливає, що є клієнт, запити якого система повинна обробити, і для обробки цих запитів існують модулі. При цьому модулі можуть використовувати послуги та сховища так, як їм це потрібно. Відповідальність за роботу завжди лежить у модулі. Він моніторить зміну стану сховищ та використовує послуги для виконання того, що йому потрібно. Виходить, що головна одиниця системи – модуль. Якщо заглибитись, то кожен модуль має декілька властивостей:

- він може самостійно ухвалювати рішення (interactor).
- знає у тому, де взяти потрібну йому інформацію (index).
- має певний вигляд (router).

Розберемо кожну складову детальніше.

Модуль (Module). Основний будівельний блок системи – Модуль (Module). Це незалежна одиниця, яка містить деяку інкапсульовану

поведінку. При цьому модуль може бути без візуальної презентації. Це дозволяє розгорнути стандартні залежності програми та зробити так, щоб візуалізація (View) не керувала усім додатком, а перебудовувалася залежно від потрібної поведінки. У нашій архітектурі ми описуємо, що має бути у системі. За умовчанням логіка лежить біля кожного модуля. При цьому не всі модулі містять візуальну презентацію.

і. Ця важлива зміна дозволяє нам розгорнути залежність. Тепер програма залежить не від візуального шару, а від логічного.

Складові модуля (рис. 3.1):

– Базовий модуль і двох файлів: Index, Interactor.

– Якщо у модулі потрібна візуалізація – додаємо Router.

– Якщо модуль складається із складної візуалізації, приміром є кілька A/B тестів, які відрізняються лише візуально, але мають загальну логіку – додаємо View (або декілька, за потребою).

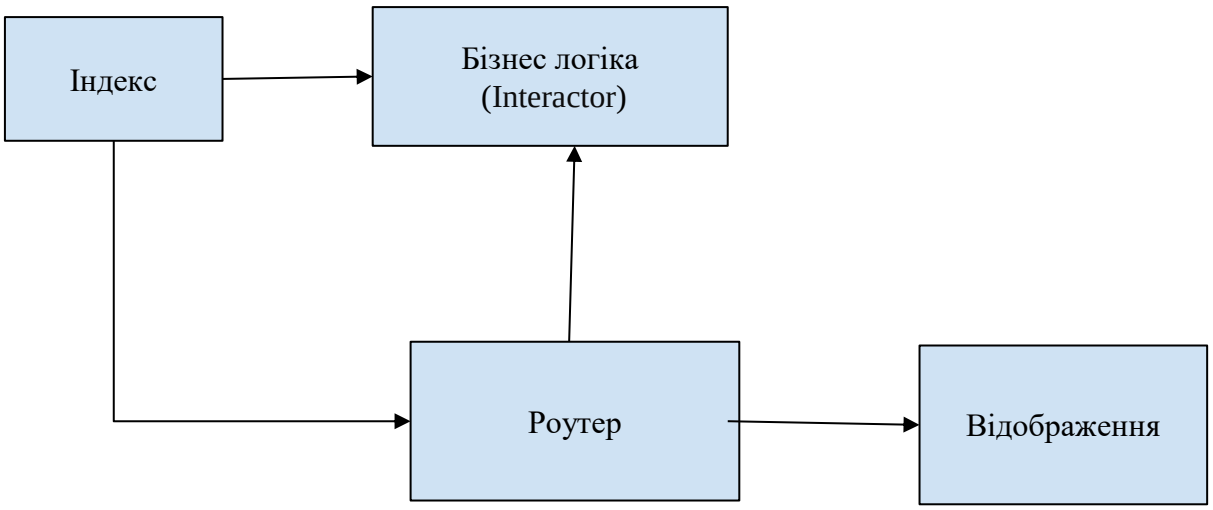

Рисунок 3.1 – Складові модуля

На схемі стрілками позначені залежності між компонентами модуля

Interactor. Цей файл містить бізнес-логіку. Бажано структурувати систему так, щоб інтерактор був ізольований від зовнішнього світу та отримував потрібні залежності через пропси. Також гарною практикою

вважається робити його хуком.

Для більшого розуміння та наочності наведемо приклад коду:

```
type Payload = {
 authenticationService: IAuthenticationService
 authenticationStore: IAuthenticationStore
 router: IRouter
}

interface IUserSignupByEmailInteractor {
 redirectToSignin: () => void
 passwordRecoveryUrl: string
 children: {
        signupByEmail: boolean
        signupByGoogle: boolean
 }
}

const useSignupPageInteractor = ({ authenticationService, router,
authenticationStore }: Payload): IUserSignupPageInteractor => {
 // logic implementation here

 return {
        redirectToSignin: () => router.redirect('/signin'),
   routeName: '/custom-route',
        children: {
        signupByEmail: true,
        signupByGoogle: canUserSignupByGoogle,
        }
 }
}
```

Так ми отримуємо потрібне сховище та сервіс із пропсів. Спочатку позначаємо, які дочірні модулі може рендерувати цей модуль. Потім виконуємо перевірки та повертаємо булеві значення для кожного дочірнього модуля. Також додаємо інші пропси за потреби. Рендер реактових компонентів виконуватиметься вже в роутері.

Роутер. Файл, який пов'язує бізнес-логіку та візуалізацію. Інакше кажучи, прокидає props компоненти, викликає потрібні коллбеки і розставляє дочірні модулі.

Продемонструємо приклад коду.

```
interface IProps {
  signupByEmail: React.ReactNode
  signupByGoogle: React.ReactNode
  interactor: IUserSignupByEmailInteractor
}

const SignupPageRouter: React.FC = ({ signupByEmail, signupByGoogle,
interactor }) => (
  <>


        {interactor.children.signupByEmail && signupByEmail}
        {interactor.children.signupByGoogle && signupByGoogle}
  </>
)
```

Роутер отримує всі необхідні залежності через пропси, і відображає у DOM дерево необхідні модулі та компоненти. Містить лише логіку перевірок if else щоб зрозуміти, чи потрібно відображати той чи інший модуль.

Відображення. Завдання даного файлу – групувати різні «дурні» компоненти (dumb components).

```
interface IProps {
  link: string
}

const ForgotPassword: React.FC = ({ link }) => (
```

Файл Index. Збирає необхідні залежності для інтерактора та роутера. У ньому ми викликаємо всі useContext, завантажуємо потрібні послуги та сховища.

```
{
  const { authenticationService } = useServices()
```

```
  const { router } = useUtils()
  const { authenticationStore } = useStores()

  const interactor = useSignupByEmail({ authenticationService, router,
authenticationStore })

  return (
        }
        signupByGoogle={}
        interactor={interactor}
    />
  )
}
```

Основні файли для побудови модуля – Index та Interactor. Решту додаємо за потребою. Для зручності модулі без UI можна назвати Activity, а ті, що з UI, залишити як і є Module. До тих пір, поки нам не потрібне розширення системи, можемо зберігати стан прямо всередині модулів через хук useState. Для загальних даних можна створити базовий createContext. Для отримання даних із зовнішніх джерел (API та інші сервіси) можна писати невеликі функції у index файлах.

Сервіси (Services). Якщо потрібно використовувати логіку отримання даних із зовнішніх джерел у різних модулях, виносимо її в окремі сервіси. Сервіс – це простий request-response механізм. Він може зберігати внутрішній стан, але тільки той, який йому потрібний для успішного виконання запитів.

Сховища (Stores). Дані потрібно не лише отримати, а й зберігати. У поточній системі кожен модуль може зберігати дані в собі, але якщо потрібно отримати загальний стан, такі дані краще виносити в сховище.

3.2 Реалізація клієнтської частини web-додатку

Спираючись на вимоги, наведені у першому розділі, було побудовано засіб для моделювання екосистеми інженерії програмного забезпечення за

допомогою бібліотеки React.js. Розглянемо розроблений застосунок детальніше.

Точкою входу в даний застосунок, як і у решту застосунків, що розроблено за допомогою React.js, є файл index.jsx, що рендерить в собі базовий клас.

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

Безпосередня логіка, що відповідає за відображення необхідних елементів, обробки тих чи інших подій інкапсульована у компоненті App.

З урахуванням визначених вимог продемонструємо їхню імплементацію. Передусім необхідно надати можливість користувачеві моделювати певну систему за допомогою елементів, наявних в системі. Для поліпшення користувацького інтерфейсу було надано можливість кастомізувати певні елементи в залежності від потреба користувача. Реалізуємо для прикладу базові елементи (Nodes) (рис. 3.2) та продемонструємо код, що відповідає за їх створення.

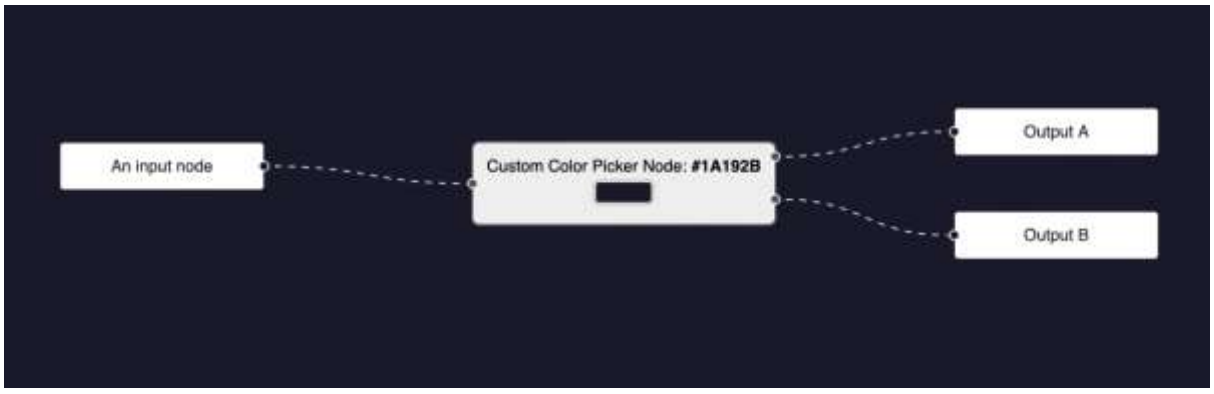

Рисунок 3.2 – Базові елементи (Nodes)

Слід зауважити, що в даному прикладі є дві основних групи. Це безпосередні вузли та їх з’єднання.

Для відображення елементів використано функцію setNodes, яку як пропс передаємо в компонент ReactFlow для її подальшого рендирингу.

Кожен елемент має загальний інтерфейс. Наведемо приклад однієї з таких node

```
{
  id: '2',
  type: 'selectorNode',
  data: { onChange: onChange, color: initBgColor },
  style: { border: '1px solid #777', padding: 10 },
  position: { x: 300, y: 50 },
 },
```

Далі для побудови взаємозв’язків між елементами використовується функція - setEdges, що приймає в собі масив, який описує взаємозв’зки між елементами. Загальний інтерфейс для цього можна відобразити у вигляді такої структури даних:

```
{
  id: 'e1-2',
  source: '1',
  target: '2',
  animated: true,
  style: { stroke: '#fff' },
 }
```

Розглянемо поля даного об’єкту детальніше.

id – необіхдний унікальний ідентифікатор, що допомагає вирізнити той чи інший зв’язок з поміж інших;

– souce – це поле слугує ідентифікатор елементу, який був ініціатор встановлення взаємозв’язку з іншим елементом;

– target – елемент з яким необхідно встановити взаємозв’язок;

– animated – булівська змінна, що слугує показчиком анімованості.

– style – певні CSS стилі, які будуть застосовані для з’єднання елементів.

Далі в процесі моделювання інколи необхідно віднести певні елементи до того чи іншого батьківського класу.

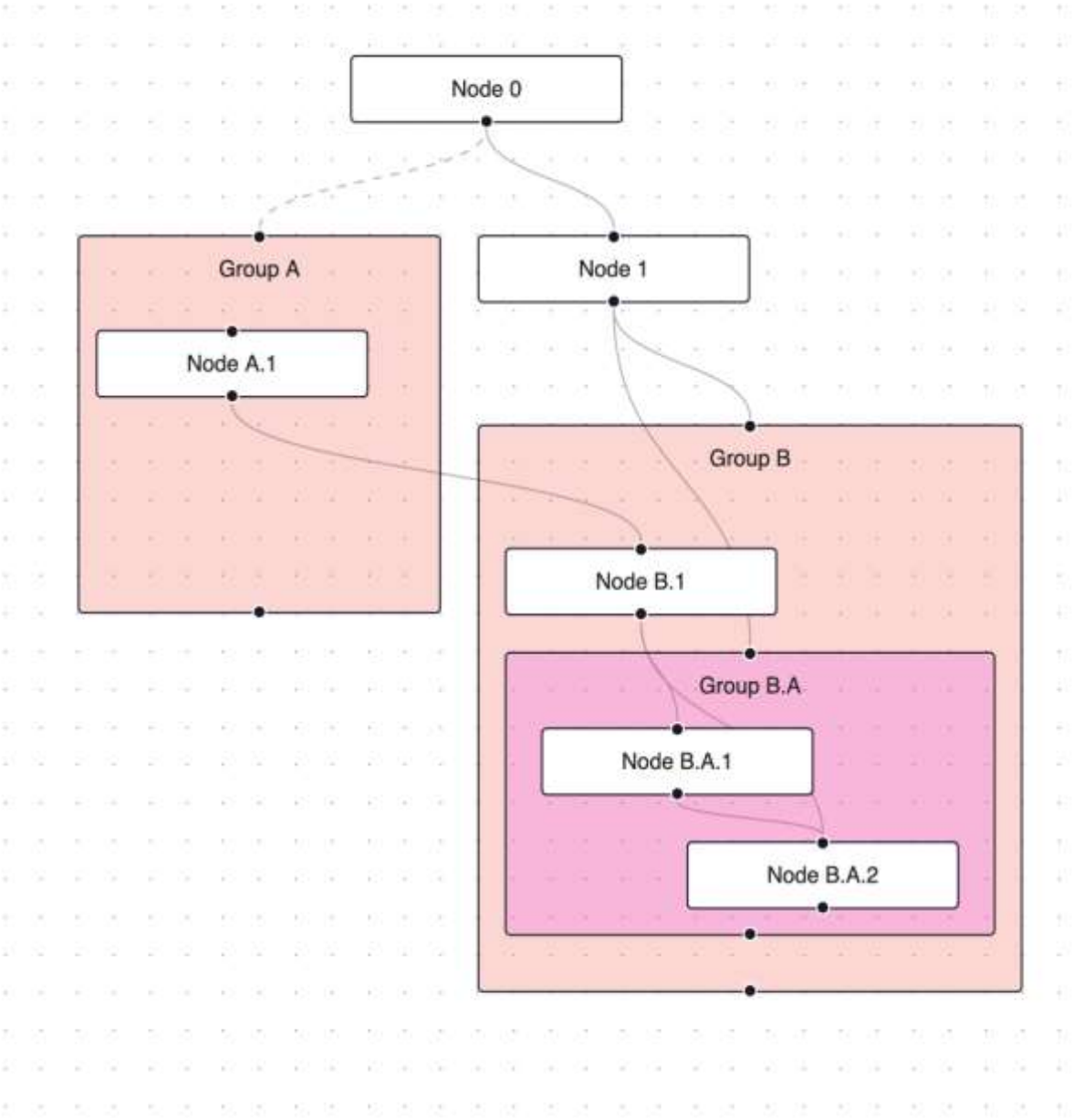

Рисунок 3.3 – Приклад батьківських та дочірніх node

Наприклад, як в групі А чи В.

Для цього нам необхідно розширити базовий інтерфейс елементу node й додати поняття батьківського елементу.

3.3 Обґрунтування ефективності запропонованих рішень

Оскільки бібліотека React.js для мови програмування JavaScript дозволяє пришвидшити роботу з DOM за рахунок роботи з віртуальним DOM, однак необхідно провести профілювання застосунку з метою виявлення місць, що сповільнюють роботу застосунку та тестування безпосереднього програмного забезпечення.

3.3.1 Тестування вебдодатку

Для успішного тестування застосунку наведено табл. 3.1, яка містить в собі інформацію про можливі дії користувача, поведінку системи та фактичний результат

Таблиця 3.1 – Тест кейсів для ПЗ

Дія користувача	Очікуваний результат	Наявний результат
Завантаження вебсторінки	Користувач бачить панель для роботи та певні елементи які наявні в робочій зоні для роботи	Присутність базових нод для моделювання екосистеми, наявність кнопок для збереження зображення та експорту\імпорту даних
Вибір певного елемента для моделювання	Можливість додати елемент в робочу зону для моделювання	При перенесенні певного типу нод в робочу зона поява даного елемента в моделі екосистеми.
Натиск на кнопку:”Save image”	Завантаження картинки на пристрій користувача	Завантаження файлу формату .jpeg
Подвійний клік на елемент моделі екосистеми	Відкриття вікна праворуч на екрані, що дає можливість редагувати той чи інший елемент	Відображення елементу, що дозволяє проводити маніпуляції зі зміни вигляду з елементом екосистеми. Можливість зміни назви елементу, кольору та розміру.
Натиск на кнопку:”Export data”	Завантаження файлу формату .json що містить в собі інформацію про змодельовану екосистему.	Завантаження файлу певного розширення з наявною інформацією про модель екосистеми.

Як бачимо, набір даних дій дозволяють побудувати модель екосистему інженерії програмного забезпечення використовуючи даний вебдодаток.

3.3.2 Продуктивність вебдодатку

Для вимірювання продуктивності запропонованого програмного забезпечення було використано веббраузер Google Chrome оскільки він надає можливість оцінювання будь-якої інтернет сторінки використовуючи консоль розробника.

Як бачимо з рисунку дане програмне забезпечення є високопродуктивним про це вказують такі метрики як швидкість рендирингу, відображення, кількість спожитою пам’яті та загальна швидкість відображення вебсторінки.

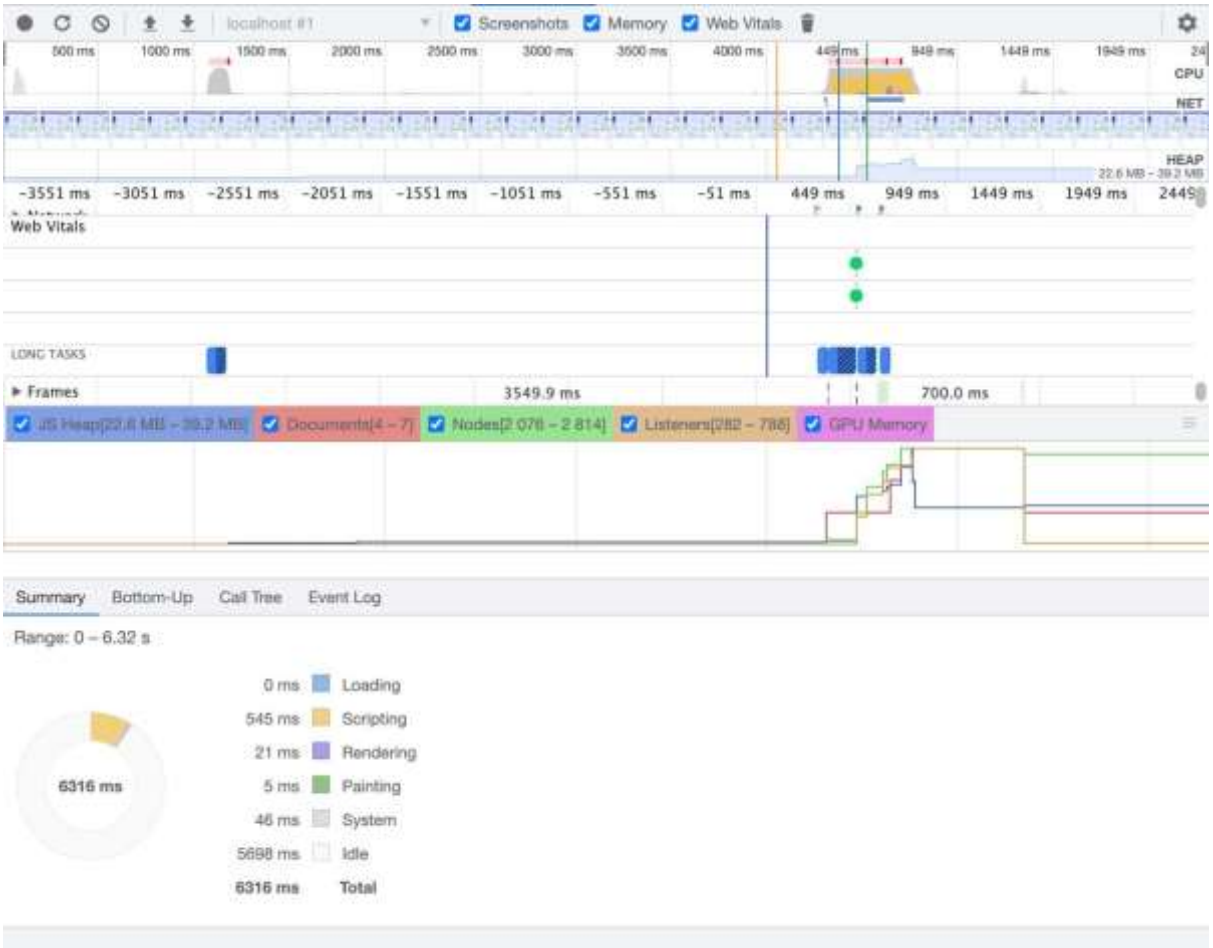

Рисунок 3.4 – Характеристика продуктивності засобу для моделювання екосистеми інженерії програмного забезпечення

Швидкодія даного програмного забезпечення була досягнута завдяки побудованій архітектурі, ефективно написаного коду та вдало підібраній бібліотеці React.js.

3.4 Висновки до розділу 3

В даному розділі, спираючись на обрану архітектуру, було імплементовано засіб для моделювання екосистем інженерії програмного забезпечення за допомогою мови програмування JavaScript та бібліотеки React.js.

Внаслідок декомпозиції модулей було створено три основні складові даного програмного забезпечення – модуль, сховища та сервіси. Основна функція модулей є доволі простою – це просто обробка запитів від користувача системи, для цього модуль вміє правильно працювати зі сховищами та сервісами.

Основному будівельному блоку системи (модулю) притаманна така поведінка, як самостійність під час ухвалення рішень, можливість отримання потрібної інформації для ухвалення тих чи інших рішень. Також слід зауважити, що в модулі може бути зосереджена виключно логіка, без необхідності наявності логіки про візуалізацію. Внаслідок цієї декомпозиції візуалізація не керує всієї бізнес логікою додатка, а відображається залежно від команд модуля. Якщо логіка візуалізації, що зосереджена в модулі, є достатньо складною, внаслідок загального архітектурного підходу є можливість створити декілька файлів, що містять виключно візуалізацію певних компонент.

Для пов’язання модуля, в якому зосереджена основна логіка, та візуалізації використовується Роутер, що передає потрібні пропси в компоненти візуалізації, викликає потрібні функції зворотнього виклику та відповідає за роботу з дочірніми модулями. Логіка, що міститься безпосередньо в самому роутері відповідає лише за перевірку відображення

того чи іншого компоненту, модуля.

Компонент презентації (відображення) даних відповідає за групування всіх необхідних компонентів для подальшого їхнього монтування в DOM дерево.

Для роботи зі сторонніми сервісами варто використовувати сервіси, що представляють собою простий механізм типу запит-відповідь. Також сервіси всередині можуть зберігати певний стан, однак лише той, що потрібен для успішного відправлення запиту.

4 РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ

Під час реалізації програмного забезпечення було використано IDE Visual Studio Code та мову програмування JavaScript з бібліотекою React.js.
Visual Studio Code поєднує в собі простоту редактора з відкритим вихідним кодом та потужними інструментами розробника, такими як IntelliSense. Перш за все, це редактор, який допомагає під час написання коду. Надзвичайно простий цикл редагування-збирання-налагодження означає менші витрати часу на налаштування середовища і більше часу на реалізацію ідей. Також Visual Studio Code доступний на таких операційних системах, як MacOS, Linux та Windows.

JavaScript, часто скорочено JS, є мовою програмування, яка є однією з основних технологій Всесвітньої павутини, поряд з HTML і CSS. Станом на 2022 рік 98% веб-сайтів використовують JavaScript на стороні клієнта для поведінки веб-сторінок, часто включають бібліотеки сторонніх розробників. Усі основні веб-браузери мають спеціальний механізм JavaScript для виконання коду на пристроях користувачів.

JavaScript – це мова високого рівня, яка часто скомпільована точно вчасно і відповідає стандарту ECMAScript. Він має динамічний тип, об’єктну орієнтацію на основі прототипу та першокласні функції. Це мультипарадигма, що підтримує керований подіями, імперативний та функціональний стилі програмування. Він має інтерфейси прикладного програмування (API) для роботи з текстом, регулярними виразами, датами, стандартними структурами даних та об’єктною моделлю документа (DOM).
Стандарт ECMAScript не включає жодних засобів вводу/виводу (I/O), таких як мережеві, сховища чи графічні засоби. На практиці веббраузер або інша система виконання надає JavaScript API для введення-виведення. Механізми JavaScript спочатку використовувалися лише у веббраузерах, але зараз є основними компонентами деяких серверів і різноманітних програм. Найпопулярнішою системою виконання для цього використання є Node.js.

4.1 Суть розробки програмного проєкту за тематикою дослідження

Поняття екосистеми походить від екології. Одне з визначень відображає екосистему як природну одиницю, що складається з усіх рослин, тварин і мікроорганізмів (біотичних факторів) на території, що функціонує разом з усіма неживими фізичними (абіотичними факторами) середовища. Хоча наведене вище визначення є чудовим, воно менш підходить, тому ми виходимо з поняття людських екосистем. Людська екосистема складається з акторів, зв'язків між суб'єктами, діяльності цих акторів і трансакцій уздовж цих зв'язків, що стосуються фізичних або нефізичних факторів. Екосистеми програмного забезпечення (SECO) відносяться до набору підприємств та їхніх взаємозв'язків на спільному ринку програмних продуктів або послуг.. Екосистема програмного забезпечення складається з набору програмних рішень, які дозволяють, підтримують і автоматизують діяльність і транзакції учасників пов'язаної соціальної або бізнес-екосистеми та організацій, які надають ці рішення. Це нова сфера, натхненна концепціями бізнес- та біологічних екосистем.

Добре відомими прикладами спільнот, які можна розглядати як екосистеми програмного забезпечення, є Apple iPhone, Microsoft, Google Android, Symbian, Ruby та Eclipse. Концепція екосистеми може означати широкий діапазон конфігурацій. Проте всі вони включають дві фундаментальні концепції: мережу організацій або акторів і спільний інтерес до розробки та використання центральної технології програмного забезпечення. Індустрія програмного забезпечення постійно розвивається і зараз зазнає швидких змін. Мало того, що продукти та технології швидко розвиваються, багато інноваційних компаній експериментують із новими бізнес-моделями, що час від часу призводить до фундаментальних зрушень у цілих галузевих структурах і взаємозв'язках між компаніями та клієнтами. Останнім часом багато компаній прийняли стратегію використання платформи для залучення масових прихильників розробників програмного

забезпечення, а також кінцевих користувачів, будуючи цілі «екосистеми програмного забезпечення» (SECO) навколо себе, незважаючи на те, що діловий світ і дослідницьке співтовариство все ще не намагаючись краще зрозуміти це явище.

4.2 Необхідність розроблення проєкту

Останні кілька десятиліть ми були свідками різних типів методологій розроблення програмного забезпечення, починаючи від водоспаду, спіралі, компонентів, швидкої розробки додатків, раціонального уніфікованого процесу до гнучких моделей відповідно. Майже всі згадані моделі заохочують розробку програмного продукту повністю на відповідній організації. Поява парадигми розробки програмної екосистеми (SECO) спричинила спільні інновації в результаті участі різних гравців, однак дослідницькі спільноти та практики все ще намагаються зрозуміти цю концепцію. Ця робота має на меті дослідити те, що відомо про програмні екосистеми інженерії програмного забезпечення. Тому для кращого розуміння всіх процесів виникла необхідність у розробленні засобу для моделювання екосистем інженерії програмного забезпечення.

4.3 Аналіз ринкових можливостей запуску стартап-проєкту

Аналіз ринкових можливостей запуску стартап-проєкту є наступним логічним кроком, у рамках якого здійснюється оцінювання кон’юнктури ринку, визначаються актуальні загрози та можливості, а також аналізуються потенційні напрямки розвитку проєкту. Ґрунтовний аналіз даних аспектів дозволяє вже на етапі планування оцінити ризики та зменшити їхній вплив на ефективність впровадження проєкту.

Основні характеристик цільового ринку стартап-проєкту наведені в табл. 4.1.

Таблиця 4.1 – Основні характеристик цільового ринку стартап-проєкту

п/п	Показники стану ринку	Характеристика
1	Кількість гравців, од.	1 (ECOS)
2	Динаміка ринку (експертна оцінка)	Зростаючий попит через ускладнення архітектури ПЗ та складності організаційних процесів
3	Наявність вхідних бар'єрів (вказати характер обмежень)	Відсутні
4	Специфічні вимоги до стандартизації та сертифікації	Відсутні
5	Середня норма рентабельності в галузі (або по ринку), %	65%≥

Аналіз цільового ринку свідчить про наявність потенціалу для розроблення та впровадження стартап-проєкту, а також залучення інвестицій для його масштабування.

При реалізації ідеї стартап-проєкту важливим чинником є розуміння специфіки цільової аудиторії та її потреб, що потенційно забезпечує більшу стабільність проєкту, передбачувані темпи зростання бізнесу та дає розуміння способів розширення бізнесу та диверсифікації його у майбутньому (табл. 4.2)

Таблиця 4.2 – Характеристика потенційної цільової аудиторії стартап-проєкту

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	2	3	4	5	6
1	Outsource / outstaffing компанії	Висока, об'єктивна щоденна потреба ефективного ведення бізнесу Висока	Близький до 100%	Низька конкуренція	Легкий
2	Продуктові IT-компанії				

Кінець таблиці 4.2

1	2	3	4	5	6
3	Приватні особи. Малий бізнес	Середня, ситуативна, залежно від сфери професійних інтересів розробника	Пропозиція актуальна в якості підвищення кваліфікації для спеціалістів, які працюють на outsource	Низька конкуренція, але наявні бюджетні обмеження	Середній
3	Держані підприємства та урядові структури	Низька, потребує додаткових мотиваційних заходів	Низький попит з перспективами зростання за умов подальшого посилення тенденції діждитадлізації державного сектору економіки та системи держаних послуг	Низька конкуренція, але наявні бюджетні обмеження	Важкий, значний організаційно-бюрократичний супротив

Проаналізувавши ключові характеристики потенційних клієнтів бачимо, що ключовими аудиторіями для стартап-проєкту є Outsource / outstaffing компанії та продуктові IT-компанії, як сегменти, що займаються займаються розробкою програмного / продукти і перед якими постійно постає потреба в моделюванні екосистеми програмного забезпечення. Ключовими користувачами на стороні бізнесу будуть: актори (учасники), що залучені до розробки програмного забезпечення (Front-end Developer, Back-end Developer, Full-stack Developer, DevOps2Engineer, QA інженери, Data Scientist, Mobile Developer, GmeDevB тощо).

Для оцінювання ризиків проведемо аналіз факторів загроз та ризиків, які можуть мати вплив на процес впровадження стартап-проєкту (табл. 4.3-4.4).

Таблиця 4.3 – Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція
1	Вихід на ринок конкурентів	Активація компаній з продуктом, що має схожий функціонал	Прискорення запуску продукту, створення бар’єрів входу та підтримка УТП
2	Низька обізнаність ринку про продукт та його технічні можливості	Вповільнений розвиток проєкту через відсутність можливості масштабування та залучення нових клієнтів	Інтенсифікація заходів просування продукту та забезпечення ефективності бізнес-комунікацій та нетворкінгу
3	Загроза безпеці даних	Побоювання з боку потенційних клієнтів щодо можливості витоку даних	Посилення систем безпеки продукту, акцент в при просування на аспекті безпеки ПЗ
4	Технічні обмеження	Потенційна обмеженість технічних рішень через заборони пов’язані з зовнішніми економічними та політичними процесами	Постійний моніторинг зовнішнього середовища на предмет потенційного впливу на стабільність та обмежень і роботі ПЗ

Таблиця 4.4 – Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція
1	2	3	4
1	Зростання попиту на побідні технічні рішення	Розвиток ІТ-індустрії та зростання складності в рамках проєктів вимагають ПЗ, яке б виконувало функції «менеджера»	Відслідковування актуальних технічних запитів ринку та оновлення ПЗ відповідно до їх специфіки
2	Актуальність ПЗ	Враховуючи специфіку процесу розвитку ІТ-індустрії можна прогнозувати відносно довгий проміжок актуальності пропозиції функціоналу ПЗ	Внесення в робочому порядку змін та оновлень для підтримання актуальності ПЗ вимогам ринку

Кінець таблиці 4.4

1	2	3	4
3	Універсальність	УПТ програмного забезпечення є актуальним для практично усіх сфер використання, що зменшує обсяг капіталовкладень для кастомізації	Фокс не на адаптації функціоналу ПЗ до вимог кожного конкретного клієнту, а на забезпечені універсальних аспектів роботи з екосистемами
4	Низький інвестиційний ризик	Універсальність сфери використання робить інвестицій в продукт максимально безпечним	Співпраця з компаніями різними сфер та географії для підтримання стійкості бізнесу в умовах зміни ринкових умов настання форс-мажорних обставин
5	Соціальна привабливість та «апаратна» функція	Можливість використання функціоналу ПЗ в рамках суспільно важливих проєктів та роботи держаних органів	Висвітлення взаємодії / використання ПЗ на рівні державних інституцій з метою формування позитивного іміджу та реноме надійного партнера

За результатами аналізу можливостей та загроз можна говорити про наявність значного потенціалу з боку ринку у випадку впровадження стартап-проєкту в життя.

Наступним етапом аналізу є дослідження особливостей конкурентного середовища (табл. 4.5).

Таблиця 4.5 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1	*2*	*3*
Олігополія	Незначна кількість конкурентів Велика ринкова сила Схожість використовуваних технологій	Інформування учасників ринку щодо появи нової типу ПЗ для управління екосистемами
Глобальний рівень	Робота на міжнародному ринку (велика кількість країн)	Донесення інформації про успіх імплементації в рамках інших країн та транснаціональних компаній

Кінець таблиці 4.5

1	2	3
Галузевий рівень	Загроза появи нових конкурентів Ринкова влада споживачів у товарі	Інформування ринку щодо якості використовуваної новаторської ПЗ, гнучкості оплати та умов співпраці
Внутрішньогалузева конкуренція	Діяльність в одній галузі економіки Надання сервісів одного типу	Розширення каналів збуту, запровадження гнучкої цінової політики
Товарно-видова конкуренція	Конкуренція між різними видами товарів	Удосконалення і оптимізація функціоналу
Цінова	Використання цін для покращення економічних умов збуту	Запровадження гнучкої цінової політики Використання реалізації
Марочна	Пропозиція схожого сервісу Спільна цільова аудиторія	Інформування учасників ринку щодо появи нової типу ПЗ для управління екосистемами

В цілому при прийнятті рішення про реалізацію стартап-проєкту може виникати конкуренція на різни рівнях, але абсолютна більшість з них може піддаватись впливу і коригуватись за рахунок заходів посилення конкурентоспроможності. Після аналізу рівнів конкуренції доцільно оцінити рівень конкуренції в галузі за М.Портером: 5 основних факторів, які мають вплив на привабливість вибору ринку, з огляду на характер конкуренції (табл. 4.6)

Таблиця 4.6 – Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постача-льники	Клієнти	Товари-замінники
	1	*2*	*3*	*4*	*5*
	ECOS	Бар'єри входження високі: для розробки необхідні високо-кваліфіковані спеціалісти та значні інвестиції	Не актуальний для продукту інтелекту-альної власності	Запит на зручність, та повноту функціоналу. Універса-льність інструменту	Обмежений функціонал, який забезпечує окремі фрагментарні запити клієнта

Кінець таблиці 4.6

	1	*2*	*3*	*4*	*5*
	CR4 = 94% Індекс Херфіндаля-Хіршмана (HHI) = 7225 Значення показників	Існує можливість виходу на ринок. Потенціал ємності не вичерпано. Можливий вхід в даний сегмент	Відсутні	Клієнт має високі запити з точки зору якості, функціоналу та	Значних обмежень немає
Висновки	вказує на (монопо-лізацію) даного ринку	міжнародних компаній		забезпечення безпеки	

Відповідно до отриманих результатів аналізу ступенів конкуренції можемо виділити та обґрунтувати фактори конкурентоспроможності і навести порівняльний аналіз сильних та слабких сторін (табл. 4.7-4.8).

Таблиця 4.7 – Порівняльний аналіз сильних та слабких сторін проєкту

№ п/п	Фактор конкурентоспроможності	Обґрунтування (чинники, що роблять фактор для порівняння конкурентних проєктів значущим)
1	Масштабованість	Продукт, що легше масштабується, здатний швидше пристосуватися до зміни попиту
2	Потреби споживачів	Повний функціонал в рамках управляння екосистемою, забезпечення безпеки
3	Актуальність та унікальність	Відповідність запиту ринку та клієнта, забезпечення вирішення актуальної нагальної системної проблеми
4	Підтримка та кастомізація продукту	Можливість внесення персоніфікованих змін під запит конкретного організації-клієнта
5	Підтримка та оновлення	Актуалізація функціоналу та інструментів відповідно до змін ПЗ, змін організаційної структури та архітектори екосистем
6	Ціна	Цінова пропозиція, що забезпечує компенсацію витрат розробнику програми і здатна забезпечити зацікавленість з боку ринку (клієнтів-організацій)

Таблиця 4.8 – Порівняльний аналіз сильних та слабких сторін проєкту

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів порівняно з продуктом-стартапом						
			-3	-2	-1	0	+1	+2	+3
1	Масштабованість	17						+	
2	Потреби споживачів	20							+
3	Актуальність та унікальність	19							+
4	Підтримка та кастомізація продукту	20					+		
5	Підтримання та оновлення	18						+	
6	Ціна	18					+		

Задля конкретизації можливостей та ризиків, а також розуміння сильних та слабких сторін проєкту проведемо SWOT-аналіз (табл. 4.9).

Таблиця 4.9 – SWOT-аналіз стартап-проєкту

Сильні сторони	**Слабкі сторони**
1	2
1. Можливість візуалізувати усі компоненти екосистеми; 2. Деталізація і формалізація процесу розробки ПЗ; 3. Спрощення управління процесом розробки ПЗ; 4. Підвищення ефективності роботи за рахунок більш швидкого входження нового спеціаліста (розробника) в існуючу екосистему; 5. Кросплатформеність (вебінтерфейс), можливість працювати в будь-якому браузері незалежно від типу операційної системи; 6. Універсальність (широкий спектр можливостей і галузей застосування).	1. Необхідність постійного оновлення відповідно до процесів компаній-клієнтів; 2. Значний обсяг робіт на старті проєкту та необхідність залучення висококваліфікованих «вузьких» спеціалістів; 3. Вузькість спеціалізації.

Кінець таблиці 4.10

1	2
Можливості	**Загрози**
1. Низька конкуренція в сегментів і пов'язана з цим перспективність напрямку; 2. Відкриття нових ринків в післявоєнний період; 3. Поява нових технологій і розвиток інженерії; 4. Глобалізація економічних та законодавчих процесів; 5. Покращення світового інвестиційного клімату.	1. Вихід на ринок нових гравців, в т.ч. «гігантів» зі значним капіталом; 2. Скорочення ринку внаслідок війни і світової економічної кризи; 3. Скорочення необхідності технологій управління екосистемами; 4. Посилення економічних та законодавчих бар'єрів у рамках окремих країн; 5. Погіршення світового інвестиційного клімату.

В результаті проведення SWOT-аналізу слід зазначити, що у продукту переважають сильні сторони, що на етапі планування стартап-проєкту є позитивним явищем і може свідчити про високу конкурентоздатність, що здатна забезпечувати економічну доцільність реалізації. Можливості та загрози можуть мати вплив залежно від зміни умов зовнішнього оточення.

Наступним етапом є розроблення альтернатив ринкового впровадження стартапів, а також визначення найкращого час для їхньої ринкової реалізації з урахованням результатів конкурентного аналізу.

Аналіз альтернатив реалізується за допомогою строків та ймовірності отримання ресурсів (табл. 4.11).

Таблиця 4.11 – Альтернативи ринкового впровадження стартапу

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Створення власного підприємства (юридичної особи)	Ймовірне (60%)	9-12 місяців
2	Робота з фізичними особами-розробниками (outsource)	Ймовірне (40%)	11-14 місяців
3	Замовлення робіт компанії-розробнику (outsource / outstaffing)	Ймовірне (50%)	9-12 місяців
Обрана альтернатива: Створення власного підприємства (юридичної особи)			

З трьох альтернативній варіантів обрано створення власного підприємства (юридичної особи). Дана альтернатива забезпечить максимальну зацікавленість в результаті та матиме відносно надійніші гарантії захисту інтелектуальних активів (ідеї стартапу) та дасть можливість в перспективі отримувати більший фінансовий дохід в результаті господарської діяльності.

4.4 Висновки до розділу 4

Враховуючи актуальність розробки програмного забезпечення, що представляє собою засіб для моделювання екосистем інженерії програмного забезпечення, було проведено розробку стартап проєкту, в рамках якого було проаналізовано ринкові можливості запуску проєкту. Було визначено ринкові можливості, що їхможна використати під час ринкового впровадження проєкту, а також ринкові загрози, які можуть перешкодити його реалізації, було проаналізовано стан ринкового середовища, потреби потенційних клієнтів. Здійснено ступеневий аналіз конкуренції на ринку та визначено альтернативи ринкового впровадження стартапу. В результаті аналізу було встановлено, що ідея стартап-проєкту технічно може бути реалізованою.

ВИСНОВКИ

Внаслідок постійного розвитку індустрії програмного забезпечення все більшого розповсюдження набуває стратегія використання платформи для залучення інженерів програмного забезпечення разом з кінцевими споживачами (користувачами), створюючи навколо себе складні системи взаємозв'язків – екосистему програмного забезпечення. Екосистема програмного забезпечення (SECO) – це нове явище, яке стрімко розвивається у сфері розроблення програмного забезпечення. Екосистема програмного забезпечення (SECO) – це набір акторів, таких як інженери програмного забезпечення, вендори, дистриб'ютори, що взаємодіють на спільному ринку програмного забезпечення з метою задоволення потреб споживачів. Це спільний інноваційний підхід розробників, організацій програмного забезпечення та третіх осіб, які поділяють спільний інтерес у розробленні технології програмного забезпечення, за допомогою якого стає можливим вирішення складних відносин між компаніями в індустрії програмного забезпечення. SECO набувають все більшого значення з появою екосистем Google Android, Apple iOS, Microsoft і Salesforce.com. З метою дослідження цього питання в роботі був використаний метод Systematic Mapping Study, що включає в собі обробку інформації для здобуття загального розуміння про предметну область дослідження.

Зроблено висновок, що екосистема програмного забезпечення (SECO) – це: 1) набір учасників, акторів, що взаємодіють на спільній платформі з метою задоволення потреб кінцевого споживача; 2) співтовариство акторів, що беруть участь в платформі програмного забезпечення; 3) набір акторів, які функціонують як єдине ціле, що взаємодіє з розподіленим ринком між програмним забезпеченням і послугами.

Встановлено та розглянуто ключові ролі SECO, такі як Keystone (основа), end-user (Кінцевий користувач), third-party organization (Сторонні організації). Досліджено один зі способів стандартизації моделювання

екосистем програмного забезпечення за допомогою SSN, що використовує такі елементи: постачальник, клієнт, посередник, агрегатор, клієнт клієнта.

З’ясовано, що за допомогою мета моделі для екосистем програмного забезпечення можливо вирішити проблему опису та структуризації SECO. Мета-модель покращує комунікацію щодо досліджень SECO, полегшуючи обмін тематичними дослідженнями або порівнюючи SECO та дослідження про SECO. Визначено основні вимоги яким повинна відповідати мета модель: має бути застосовною до кожного SECO, метамодель має бути простою у використанні та зрозумілою. Виокремлено основні складові мета моделі: актори, продукт та платформа, межі. Розглянуто життєвий спосіб екосистеми програмного забезпечення, що поділяється на три рівні: стратегічий, тактичний та оперативний. Встановлено, що екосистему інженерії програмного забезпечення можна розглядати як екосистему програмного забезпечення в межах певного ландшафту.

З метою надання можливості візуалізації екосистем інженерії програмного забезпечення було розроблено засіб для моделювання екосистем інженерії програмного забезпечення, що дає можливість поліпшити розроблення безпосереднього програмного забезпечення. Запропоновано архітектуру для імплементацію програмного забезпечення на мові програмування JavaScript та бібліотеки React.js. Обраний інструмент має високу масштабованість, швидку роботу з DOM деревом за рахунок наявності віртуально DOM дерева та багату екосистему.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Сидоров Н.А. Стилистика программного обеспечения. *Проблеми програмування*. 2018. №2-3. С. 245–254.

2. Сидоров Н.А. Экология программного обеспечения. *Інженерія програмного забезпечення*, 2010. С. 53–61.

3. Сидоров Н.А., Сидорова Н.Н., Сидоров Е.Н. Дескриптивная модель экосистемы стиля программирования. *Проблеми програмування*. 2020. № 2–3. С. 74-80.

4. Сидоров М. О. Программное обеспечение – экологический подход к исследованиям. *Инженерия программного обеспечения*. 2010. № 1. С. 5-13.

5. Aid to recovery: the economic impact of IT, software, and the Microsoft ecosystem on the global economy. IDC White Paper, 2009, p. 9.

6. Costa, G., Silva, F., Santos, R., Werner, C., and Oliveira, T. (2013). From applications to a software ecosystem platform: *An exploratory study. In Proceedings of the Fifth International Conference on Management of Emergent Digital EcoSystems*, MEDES ’13, pp. 9–16, New York, NY, USA. ACM.

7. Coutinho E., Viana D. and R. Pereira Dos Santos. An Exploratory Study on the Need for Modeling Software Ecosystems: The Case of SOLAR SECO. *2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE)*, Buenos Aires, 2017, pp. 47-53. DOI: 10.1109/MiSE.2017.3

8. Coutinho, E. F., Santos, I., and Bezerra, C. I. M. (2017). A software ecosystem for a virtual learning environment: Solar seco. *In Proceedings of the Joint 5th International Workshop on Software Engineering for Systems-of-Systems and 11th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems*, JSOS ’17, pp. 41–47.

9. David G. Messerschmitt and Clemens Szyperski. Software Ecosystem: Understanding an Indispensable Technology and Industry. Cambridge, MA, USA: MIT Press.

10. Deepak Dhungana, Iris Groher, Elisabeth Schludermann, Stefan Biffl. Software Ecosystems vs. Natural Ecosystems: Learning from the Ingenious Mind of Nature. *ECSA '10: Proceedings of the Fourth European Conference on Software Architecture: Companion Volume. August 2010*. Pp. 96–102 https://publik.tuwien.ac.at/files/PubDat_187570.pdf

11. Eric Yu and Stephanie Deng. Understanding Software Ecosystems: A Strategic Modeling Approach. *Proceedings of the Workshop on Software Ecosystems* 2011, p. 65-76

12. Francisco Victor da S. Pinheiro, Emanuel Ferreira Coutinho, Italo Santos, Carla I. M. Bezerra. A Tool for Supporting the Teaching and Modeling of Software Ecosystems Using SSN Notation. *Journal on Interactive Systems*, 2022, 13:1, DOI: 10.5753/jis.2022.2602.

13. Franco-Bedoya O., Ameller D., Costal D. and Franch X. QuESo a quality model for open source software ecosystems. *9th International Conference on Software Engineering and Applications*. (ICSOFT-EA), Vienna, Austria, 2014 pp. 209-221.

14. Graciano Neto, V. V., Basso, F., Pereira dos Santos, R., Bakar, N. H., Kassab, M., Werner, C., Oliveira, T., and Nakagawa, E. Y. (2019). Model-driven engineering ecosystems. In 2019 IEEE/ACM *7th International Workshop on Software Engineering for Systems-of-Systems (SESoS) and 13th Workshop on Distributed Software Development, Software Ecosystems and Systems-of-Systems* (WDES), pp. 58–61

15. H. Sadi M., Yu E. (2015). Designing Software Ecosystems: How Can Modeling Techniques Help?. In: Gaaloul, K., Schmidt, R., Nurcan, S., Guerreiro, S., Ma, Q. (eds) Enterprise, Business-Process and Information Systems Modeling. BPMDS EMMSAD 2015 2015. Lecture Notes in Business Information Processing, vol 214. Springer, Cham. https://cutt.ly/qBpaF3X

16. Igor R. Alencar, Emanuel F. Coutinho, Leonardo O. Moreira, and Carla I. M. Bezerra. 2020. A Tool for Software Ecosystem Models: An Analysis on their Implications in Education. *In Proceedings of the XXXIV Brazilian Symposium on*

Software Engineering (Natal, Brazil) (SBES 2020). ACM, New York, NY, USA, 10 pages.

17. Jan Bosch. 2009. From Software Product Lines to Software Ecosystems. *In Proceedings of the 13th International Software Product Line Conference* (San Francisco, California, USA) (SPLC '09). Carnegie Mellon University, USA, 111–119.

18. K. Manikas and K. M. Hansen. Software ecosystems - a systematic literature review. *Journal of Systems and Software*, vol. 86, no. 5, pp. 1294–1306, 2013.

19. Lehman M. Software Evolution – Background, Theory, Practice. Integrated Design and Process Technology. Society for Design and Process Science. 2003. 11 p.

20. Mircea F. Lungu. Reverse Engineering Software Ecosystems Doctoral Dissertation – Faculty of Informatics of the Universita della Svizzera Italiana 2009. 186 p.

21. Mircea Lungu; Michele Lanza. The small project observatory: a tool for reverse engineering software ecosystems. *2010 ACM/IEEE 32nd International Conference on Software* Engineering, 2-8 May 2010.

22. S. Jansen, A. Finkelstein, and S. Brinkkemper. A sense of community: A research agenda for software ecosystems. I*n Software Engineering, 2009. Proceedings. International Conference on,* 2009.

23. Santos, R., Werner, C., Alves, C., Jorge Santos Pinto, M., Cukierman, H., Oliveira, F., and Tania Cohen Egler, T. (2013). Ecossistemas de Software: Um Novo Espaço para a Construção de Redes e Territórios envolvendo Governo, Sociedade e a Web, pp. 337–366.

24. Sydorov N. Toward software artifacts ecosystem. *Проблеми програмування*. 2020. № 4. С. 110-120.

25. Sydorov N., Sydorova N., Sydorov E., Cholyshkina O., Batsurovska I. Development of the approach to using a style in software engineering, *Eastern-European Journal of Enterprise Technologies*. 2019. 4/2 (100) Pp. 41–51.

26. Ultra-Large-Scale Systems. The Software Challenge of the Future. SEI. Pittsburgh. June, 2006. 150 p.

27. V. Berk, I. V. Den, S. Jansen, and L. Luinenburg. Software ecosystems: a software ecosystem strategy assessment model. in *Proceedings Of the 4th Software Architecture, European Conference (Ecsa), Companion Volume DBLP*, DBLP, Copenhagen, Denmark, August, 2010.

28. Vasilis Boucharas, Slinger Jansen, and Sjaak Brinkkemper. 2009. Formalizing Software Ecosystem Modeling. In Proceedings of the 1st International Workshop on Open Component Ecosystems (Amsterdam, The Netherlands) (IWOCE '09). Association for Computing Machinery, New York, NY, USA, 41–50.

29. Wouters J., Ritmeester J. R., Carlsen A. W., Jansen S., Wnuk Krzysztof. A SECO meta-model: A common vocabulary of the SECO research domain. Lecture Notes in Business Information Processing, Springer , 2019, Vol. 370, p. 31-45.

30. Yu and Stephanie Deng. Understanding Software Ecosystems: A Strategic Modeling Approach. *Proceedings of the Workshop on Software Ecosystems* 2011, p. 65-76

31. Z. Liao, B. Zhao, S. Liu et al. A prediction model of the project life-span in open source software ecosystem. *Mobile Network Application*, vol. 24, no. 4, pp. 1382–1391, 2019.

32. Zhifang Liao, Xiaofei Qi, Yan Zhang, Xiaoping Fan. How to Evaluate the Productivity of Software Ecosystem: A Case Study in GitHub. Hindawi. August 2020; *Scientific Programming*. 2020(12):1-13.

ДОДАТОК А

```
import React, {
  useState,
  useRef,
  useCallback,
  useEffect,
  useMemo,
} from "react";
import ReactFlow, {
  addEdge,
  useNodesState,
  useEdgesState,
  Controls,
  Background,
  MiniMap,
  updateEdge,
  MarkerType,
} from "react-flow-renderer";

import TextField from "@mui/material/TextField";
import { Layout, Sidebar } from "../components";
import "../assets/Css/updatenode.css";
import "../index.css";
import { Grid } from "@mui/material";
import { v4 as uuidv4 } from "uuid";

import CustomEdge from "../components/FlowComponents/CustomEdge";
import ConnectionLine from "../components/FlowComponents/CustomEdge/ConnectionLine";
import DecisionNode from "../components/FlowComponents/Nodes/DecisionNode";
import CircleNode from "../components/FlowComponents/Nodes/CircleNode";
import RectangleNode from "../components/FlowComponents/Nodes/RectangleNode";
import { LogicalData } from "../assets/LogicalData";



const FlowCanvas = () => {
  const reactFlowWrapper = useRef(null);
  const flowImageDownloadRef = useRef();
  const edgesData = LogicalData.filter((item) => item.type === "edge");
  const nodesData = LogicalData.filter((item) => !(item.type === "edge"));
  const [nodes, setNodes, onNodesChange] = useNodesState(nodesData);
  const [edges, setEdges, onEdgesChange] = useEdgesState(edgesData);
  const [reactFlowInstance, setReactFlowInstance] = useState(null);
  const [openEditor, setOpenEditor] = useState(false);
  const [nodeName, setNodeName] = useState("NULL");
  const [nodeBg, setNodeBg] = useState("NULL");
  const [group, setGroup] = useState("");

  const nodeTypes = useMemo(
    () => ({
      decision: DecisionNode,
      circle: CircleNode,
      rectangle: RectangleNode,
    }),
    []
  );

  const edgeTypes = useMemo(
    () => ({
      custom: CustomEdge,
    }),
    []
  );
  const [sizeX, setSizeX] = useState(0);
  const [sizeY, setSizeY] = useState(0);
  const [type, setType] = useState();
  const [parent, setParent] = useState();
  const [id, setID] = useState();
```

```
const [jsonInput, setJsonInput] = useState("");

const convert = useCallback(
  (event) => {
    const jsonNode = JSON.parse(jsonInput);
    const filteredEdges = jsonNode.filter((item) => item.type === "edge");
    const filteredNodes = jsonNode.filter((item) => item.type !== "edge");
    setNodes((nds) => nds.concat(filteredNodes));
    setEdges((nds) => nds.concat(filteredEdges));
  },
  [jsonInput, setNodes, setEdges]
);

const onConnect = useCallback(
  (params) => {
    //circle to rectangle ---- rectangle to rectangle -=---
    if (!params?.source || !params?.target) {
      return;
    }
    const source = params?.source;
    const target = params.target;
    const newcircleLineEdgeSource = source.split("_").includes("circle");
    // const newcircleLineEdgeTarget = target.split("_").includes("circle");
    const newrectangleEdgeSource = source.split("_").includes("rectangle");
    const newrectangleEdgeTarget = target.split("_").includes("rectangle");
    // const newEdgeId = edgeArrowId(source, target);
    const newSource = new Array(source);
    const newTarget = new Array(target);

    const getAnimatedEdge = () => {
      if (newcircleLineEdgeSource && newrectangleEdgeTarget) {
        return true;
      }
      if (newrectangleEdgeSource && newrectangleEdgeTarget) {
        return true;
      } else {
        return false;
      }
    };
    console.log(getAnimatedEdge());
    setEdges((eds) =>
      addEdge(
        {
          ...params,
          type: "edge",
          animated: getAnimatedEdge(),
          style: { stroke: "black" },
          markerEnd: {
            type: MarkerType.ArrowClosed,
          },
        },
        eds
      )
    );
  },
  [setEdges]
);

const onDragOver = useCallback((event) => {
  event.preventDefault();
  event.dataTransfer.dropEffect = "move";
}, []);

useEffect(() => {
  setNodes((nds) =>
    nds.map((node) => {
      let x = 0;
      let y = 0;
      if (node.id === parent) {
        x = node.position.x;
```

```
        y = node.position.y;
        console.log("parent: " + node.id + " " + parent);
        console.log("parent posx: " + x);
        console.log("parent posy: " + y);
       } else if (node.selected === true && node.type !== "group") {
        node.parentNode = parent;
        node.position.x = x;
        node.position.y = y;
        node.extent = "parent";
       }
       return node;
     })
   );
}, [parent, setNodes]);

useEffect(() => {
  setNodes((nds) =>
    nds.map((node) => {
      if (node.selected === true) {
        node.data = {
          ...node.data,
          label: nodeName,
        };
        node.style = { ...node.style, backgroundColor: nodeBg };
        console.log("size: " + sizeX);

        node.style.width = parseInt(sizeX);
        node.style.height = parseInt(sizeY);
      }

      return node;
    })
  );
}, [nodeName, nodeBg, sizeX, sizeY, setNodes]);

useEffect(() => {
  setNodes((nds) =>
    nds.map((node) => {
      if (node.selected === true) {
        console.log("selected found");
        // when you update a simple type you can just update the value
        node.type = "group";

        setType(node.type);

        setGroup("");
      }
      console.log("not selected");
      return node;
    })
  );
}, [group, setNodes]);

const onDrop = useCallback(
  (event) => {
    event.preventDefault();

    const reactFlowBounds = reactFlowWrapper.current.getBoundingClientRect();
    const type = event.dataTransfer.getData("application/reactflow");
    const label = event.dataTransfer.getData("application/reactflow/label");
    const bgCol = event.dataTransfer.getData("application/reactflow/color");
    if (typeof type === "undefined" || !type) {
      return;
    }
    const position = reactFlowInstance.project({
      x: event.clientX - reactFlowBounds.left,
      y: event.clientY - reactFlowBounds.top,
    });
    let newNode;
    if (type === "rectangle") {
```

```
      newNode = {
        id: `flow_${type}_renderer_${uuidv4()}`,
        type,
        position,
        data: { label: `${label}` },
        style: {
          backgroundColor: bgCol,
          width: 100,
          height: 40,
          borderRadius: 6,
          borderColor: "#1111",
        },
      };
      setNodes((nds) => nds.concat(newNode));
    }
    if (type === "circle") {
      newNode = {
        id: `flow_${type}_renderer_${uuidv4()}`,
        type,
        position,
        data: { label: `${label}` },
        style: {
          backgroundColor: "white",
          width: 80,
          height: 80,
          borderRadius: "50%",
          border: "1px solid gray",
        },
      };
      setNodes((nds) => nds.concat(newNode));
    }
    if (type === "decision") {
      newNode = {
        id: `flow_${type}_renderer_${uuidv4()}`,
        type,
        position,
        data: { label: `${label}` },
        style: {
          backgroundColor: "",
          width: "",
          height: "",
          borderRadius: 6,
          borderColor: "#1111",
        },
      };
      setNodes((nds) => nds.concat(newNode));
    }
  },
  [reactFlowInstance, setNodes]
);
const onNodeClick = (event, node) => {
  setOpenEditor(true);
  event.preventDefault();
  setNodeBg(node.style.backgroundColor);
  setNodeName(node.data.label);
  setSizeX(node.style.width);
  setSizeY(node.style.height);
  setType(node.type);
  setID(node.id);
  if (node.type === "group") {
  }
};

const onPaneClick = (event) => setOpenEditor(false);
const onEdgeUpdate = (oldEdge, newConnection) =>
  setEdges((els) => updateEdge(oldEdge, newConnection, els));
const graphStyles = { width: "100%", height: "650px", Background: "white" };
return (
  <Layout
    jsonInput={jsonInput}
```

```
 setJsonInput={setJsonInput}
 convert={convert}
 flowImageDownloadRef={flowImageDownloadRef}
 downloadJSON={[...edges, ...nodes]}
>
 <div className="bg-indigo-100 py-2">
  <div className>
   <Grid container spacing={2}>
    <Grid item xs={2}>
     <Sidebar />
    </Grid>
    <Grid ref={flowImageDownloadRef} item xs={10}>
     <div
      className=" bg-indigo-100 rounded-md my-1 mx-2 border-2 border-indigo-400"
      ref={reactFlowWrapper}
     >
      <ReactFlow
       nodes={nodes}
       edges={edges}
       onNodesChange={onNodesChange}
       onEdgesChange={onEdgesChange}
       onConnect={onConnect}
       onInit={setReactFlowInstance}
       onDrop={onDrop}
       onDragOver={onDragOver}
       onEdgeUpdate={onEdgeUpdate}
       connectionLineComponent={ConnectionLine}
       connectionLineType="smoothstep"
       onNodeDragStart={(event, node) => {
        event.preventDefault();
        setNodeBg(node.style.backgroundColor);
        setNodeName(node.data.label);
        setSizeX(node.style.width);
        setSizeY(node.style.height);
        setType(node.type);
        setID(node.id);
       }}
       onPaneClick={onPaneClick}
       nodeTypes={nodeTypes}
       edgeTypes={edgeTypes}
       onNodeClick={onNodeClick}
       style={graphStyles}
      >
       <MiniMap />

       {openEditor && (
        <div className="updatenode__controls ">
         <div className="grid grid-cols-1 divide-y divide-black">
          <div>
           <TextField
            value={nodeName}
            label={`Label:`}
            onChange={(evt) => setNodeName(evt.target.value)}
            id="outlined-basic"
            variant="outlined"
           />
           <TextField
            label={`Background:`}
            value={nodeBg}
            disabled={type === "decision" ? true : false}
            onChange={(evt) => setNodeBg(evt.target.value)}
            id="outlined-basic"
            variant="outlined"
           />

           <div className="updatenode__checkboxwrapper">
            <label>
             ----------
             ----------
            </label>
```

```
                    </div>
                    <TextField
                      label={`Width:`}
                      value={sizeX}
                      disabled={type === "decision" ? true : false}
                      onChange={(evt) => setSizeX(evt.target.value)}
                      id="outlined-basic"
                      variant="outlined"
                    />
                    <TextField
                      label={`Height:`}
                      value={sizeY}
                      disabled={type === "decision" ? true : false}
                      onChange={(evt) => setSizeY(evt.target.value)}
                      id="outlined-basic"
                      variant="outlined"
                    />
                    <div>
                      <div className="py-1">Info:</div>
                      <div
                        style={{
                          fontWeight: "600",
                          whiteSpace: "nowrap",
                          width: "180px",
                          overflow: "hidden",
                          textOverflow: "ellipsis",
                        }}
                      >
                        Type: {type}
                      </div>
                      <div
                        style={{
                          fontWeight: "600",
                          whiteSpace: "nowrap",
                          width: "180px",
                          overflow: "hidden",
                          textOverflow: "ellipsis",
                        }}
                      >
                        ID: {id}
                      </div>
                    </div>
                  </div>
                </div>
              )}

              <Controls />
              <Background gap={8} color="black" />
            </ReactFlow>
          </div>
        </Grid>
      </Grid>
    </div>
  </div>
 </Layout>
 );
};

export default FlowCanvas;
```