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

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

«На правах рукопису»
УДК 004.4'22

До захисту допущено:
Завідувач кафедри
_____ Олександр РОЛІК
«___»_____20__ р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Інженерія програмного забезпечення комп'ютерних систем»

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Програмний засіб автоматизації відлагодження вихідного коду високої зв'язаності»

Виконав:
студент VI курсу, групи ІТ-91мн
Покровський Андрій Максимович _____

Керівник:
д-р. фіз.-мат. наук, професор
Дорошенко Анатолій Юхимович _____

Рецензент:
Посада, науковий ступінь, вчене звання,
Прізвище, ім'я, по батькові _____

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

Київ – 2021 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

Рівень вищої освіти – другий (магістерський)

Спеціальність – 121 «Інженерія програмного забезпечення»

Освітньо-наукова програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ Олександр РОЛІК

«___»_____20__ р.

ЗАВДАННЯ

на магістерську дисертацію студенту

Покровському Андрію Максимовичу

1. Тема дисертації: «Програмний засіб автоматизації відлагодження вихідного коду високої зв'язаності», науковий керівник дисертації Дорошенко Анатолій Юхимович, д-р. фіз.-мат. наук, професор, затверджені наказом по університету від «12» березня 2021 р. № 809-с

2. Термін подання студентом дисертації: 11 травня 2021р.

3. Об'єкт дослідження: процес відлагодження та рефакторингу програм.

4. Предмет дослідження: використання графічних візуалізацій з метою автоматизації процесу відлагодження та рефакторингу програм.

5. Перелік завдань, які потрібно розробити: ознайомитися з використанням графічних візуалізацій у процесі відлагодження програм, провести аналіз наявних засобів побудови графічних візуалізацій, визначити задачі, що необхідно вирішити у процесі розробки, сформувати вимоги до програмного засобу, оглянути доступні технології та ахітектурні рішення, реалізувати засіб відповідно до визначених вимог, проаналізувати результати роботи.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: діаграма варіантів використання системи, діаграма компонентів, діаграми класів модулів системи,

діаграма послідновності виконання операцій, приклад застосування запропонованої структури даних.

7. Орієнтовний перелік публікацій: Метод стислого представлення алгоритмів, Засіб вимірювання метрик вихідного коду Fortran за допомогою синтаксичного аналізу.

9. Дата видачі завдання 1 лютого 2021 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Аналіз наявних рішень	02.02.2021-10.02.2021	
2	Формування вимог до системи	11.02.2021-15.02.2021	
3	Визначення сценаріїв використання системи	16.02.2021-22.02.2021	
4	Вибір технологій для розробки засобу	23.02.2021-28.02.2021	
5	Реалізація бізнес-логіки засобу	01.03.2021-28.03.2021	
6	Розробка інтерфейсу користувача	29.03.2021-05.04.2021	
7	Аналіз виконаної роботи	06.04.2021-08.04.2021	
8	Оформлення текстового матеріалу	09.04.2021-16.04.2021	
9	Оформлення графічного матеріалу	17.04.2021-25.04.2021	
10	Подача дисертації на перевірку	26.04.2021-30.04.2021	
11	Передзахист магістерської дисертації	05.05.2021	
12	Доопрацювання пояснювальної записки та підготовка презентації	06.05.2021-11.05.2021	
13	Захист магістерської дисертації	18.05.2021	

Студент _____ Андрій ПОКРОВСЬКИЙ

Науковий керівник _____ Анатолій ДОРОШЕНКО

РЕФЕРАТ

Магістерська дисертація містить 106 сторінок пояснювальної записки, 21 рисунок, 20 таблиць, 10 додатків та 22 посилання на використані літературні джерела.

Об’єктом дослідження є процес відлагодження та рефакторингу програм. Предметом дослідження є використання графічних візуалізацій з метою автоматизації процесу відлагодження та рефакторингу програм.

Метою дисертації є створення інструменту візуалізації вихідного коду за допомогою блок-схем з можливістю багаторазового використання розроблених модулів.

Актуальність даної роботи полягає у вирішенні проблеми проведення рефакторингу та відлагодження програмних систем з сильно зв’язаним вихідним кодом. В роботі реалізовано засіб автоматизації відлагодження що використовує графічнне представлення вихідного коду для полегшення аналізу алгоритмів, на яких побудовано програму. Засіб відрізняється від аналогів використанням модульної архітектури, що значно полегшує процес його інтеграції з популярними середовищами розробки.

Архітектура, розроблена в результаті виконання цієї роботи, може бути розширена та використана при побудові блок-схем з вихідного коду будь-якої мови програмування та в довільному середовищі виконання, оскільки використані технології дозволяють встановити та користуватися засобом на всіх розповсюджених платформах.

Ключові слова: вихідний код, блок-схема, рефакторинг, відлагодження, багаторазове використання, модульна архітектура.

ABSTRACT

The master's thesis contains 106 pages of explanatory note, 21 drawings, 20 tables, 10 appendices and 22 references to used information sources.

The object of research is the process of software debugging and refactoring. The subject of research is the use of graphical visualizations in enhancing software debugging and refactoring experience.

The purpose of this thesis is to design and develop a software tool for source code flowcharting. The work has an added goal of creating reusable infrastructure.

The relevance of research is tied to the severity of challenge debugging and refactoring of highly-coupled source code presents to the developer. As the result of research, a software tool for automatization of debugging and refactoring was developed. The tool utilizes a flowchart representation of the source code to aid the analysis of algorithms that implement the software’s functionality. The main feature of the developed product is its design, which is based on modular architecture and encourages integration with popular IDEs.

Architecture designed during the work on this thesis can be easily extended and used to plot flowcharts based on arbitrary programming language source code. The technologies utilized in development allow the tool to be set up and executed on virtually any operating system and runtime environment.

Keywords: source code, flowchart, refactoring, debugging, reusability, modular architecture.

ЗМІСТ

ПЕРЕЛІК ТЕРМІНІВ ТА СКОРОЧЕНЬ

Індентація - у розробці програмного забезпечення термін, що означає одночасно відступи у вихідному коді програми та правила застосування цих відступів для покращення зрозумілості коду або як частину синтаксису мови програмування

Зв'язаність - характеристика архітектури програмної системи, що означає високий ступінь залежності програмних елементів один від одного. Така риса ускладнює підтримку системи.

Парсинг - процес аналізу текстової інформації, зазвичай у формі природної мови, мови програмування або структури даних, яка має дотримуватися певного набору зазадалегідь визначених синтаксичних правил

Плагін - в програмному забезпеченні модуль розширення, що встановлюється поверх програмної системи, розширюючи її функціональність.

Реверс-інжиніринг (зворотний інжиніринг) - процес переходу від спостереженням за функціональністю об'єкта до розуміння його внутрішньої структури. У програмному забезпеченні означає отримання архітектури програмного продукту через аналіз вихідного коду

Рефакторинг - процес переосмилення реалізації певного функціоналу, що зазвичай визначається як покращення, застосоване методом змін у вихідному коді програми без зміни її поведінки.

Реінжиніринг - поглиблений процес рефакторингу, що покращує підтримуваність та масштабованість системи шляхом видозміни її архітектури

AST (Abstract Syntax Tree) - абстрактне синтаксичне дерево, структура даних що містить інформацію про вкладеністьЮ послідовність та склад вихідного коду програми.

EOT - при передачі даних, спеціальний символ, що означає кінець поточного повідомлення.

FPS (Frames Per Second) - кадрів на секунду, частота оновлення зображення на екрані або у вікні застосунку.

Fixed form - форма запису вихідного коду мови FORTRAN, що враховує горизонтальне положення символа в рядку при перевірці синтаксичного складу програми.

Flowchart - блок-схема - діаграма, що за допомогою елементів, описує алгоритм або комп'ютерну програму як структуру з кроків та логічних переходів.

Free form - форма запису вихідного коду мови FORTRAN, що не накладає обмежень на положення симолів у рядках.

GUI (Graphical User Interface) - графічний інтерфейс користувача, набір візуальних інтерактивних елементів що призначені для вводу та виводу інформації.

IDE (Integrated Development Environment) - інтегроване середовище розробки, набір інструментів для редагування, компіляції, виконання, відлагодження вихідного коду програм.

Open-source - тип програмного забезпечення, що видається під ліцензією, що надає користувачам право нна перегляд, модифікацію, виконання та інші дії над вихідним кодом програми.

UML (Unified Modelling Language) - стандартизована мова моделювання що складається з інтегрованого набору діаграм, за допомогою яких розробники програмного забезпечення та системні архітектори можуть здійснювати моделювання структур та закономірностей, пов'язаних з програмним забезпеченням або описом процесів.

ВСТУП

З кожним роком обсяги програмних проектів стають дедалі більшими. У середовищі де будь-які бізнес-процеси, починаючи від замовлення їжі та закінчуючи системами контролю персоналу вимагають створення певного виду програмних систем та засобів, індустрія програмного забезпечення стає дедалі більш конкурентною, а це означає що швидкість, з якою необхідно здійснювати розробку нових продуктів та нового функціоналу, неперервно зростає.

Одночасно, за останні десять років значно збільшився середній розмір команд, що займаються паралельною розробкою програмного модуля чи системи, а поріг входження у індустрію програмування стає все нижчим з кожним створеним високорівневим фреймворком, технологією чи мовою програмування. Навіть сьогодні цілком звичайною є ситуація, коли в компанії з розробки веб-сайтів одночасну роботу над продуктом ведуть десять-двадцять розробників, половина з яких можуть не мати досвіду програмування поза межами університету.

В описаній системі, де вимоги до кількості коду, написаного за день, зростає швидше, ніж будь-коли, часто виникають ситуації коли швидкість розробки нового функціоналу ставлять вище за якість архітектури та вихідного коду. Після завершення стресового періоду у життєвому циклі продукту зазвичай виявляється, що подальше розширення бази вихідного коду та реалізація нових функцій неможлива без переосмислення наявної структури системи. Такий процес називають рефакторингом вихідного коду і ця задача ставить перед розробником низку нових труднощів, які значно відрізняються від звичайної роботи над проектом.

Виразною рисою процесу рефакторингу є необхідність переосмислення архітектури системи, що зазвичай була реалізована іншим розробником, проконсультуватися з яким частіше за все можливості немає. В цій ситуації зрозумілість такого коду в основному залежить від специфіки мови програмування (наскільки строгі обмеження на структуру вихідного коду та архітектуру системи вона накладає) та від досвідченості та патернів мислення розробника, що писав цей

код (наскільки добре ця архітектура працює з особливостями та обмеженнями мови програмування та використаних підходів).

Рефакторинг - це процес, що концептуально являє собою переклад програми з вихідного коду на абстрактні структури, що вбачалися розробником оригінального алгоритму та на новий набір структур, що за задумом виконавця рефакторингу краще підходять для забезпечення масштабованості та низької зв'язаності, і у фіналі повторну реалізацію цих структур у вихідному коді програми. Він вимагає від розробника високої компетенції у розумінні корисних та шкідливих звичок у програмуванні, а також значного досвіду у розплутуванні вихідного коду та виділенні з нього закономірностей та функцій. Але найбільше, процес рефакторингу вимагає від розробника здібностей до структуризації та візуалізації інформації, якостей, які далеко не всім даються легко.

Для полегшення роботи при проведенні рефакторингу, як і при будь-якому плануванні складної системи, використовуються графічні візуалізації архітектури та алгоритмів. У випадку з алгоритмами, для їх візуалізації ще до початку зародження індустрії програмного забезпечення було запропоновано підхід з представлення алгоритмів у вигляді структур з кроків та переходів між цими кроками. Така структура отримала назву блок-схема та з моменту своєї появи слугує ефективним методом візуалізації роботи програм та алгоритмів.

Зараз, блок-схеми та діаграми в цілому набули глобального застосування як засіб моделювання, скороченого запису та аналізу явищ та систем. Існує багато програмних інструментів, що підтримують побудову, виведення або перетворення діаграм у тому чи іншому вигляді. Такі інструменти зазвичай постачаються як окремі застосунки, відв'язані від програмних екосистем, в яких зазвичай працюють розробники програмного забезпечення.

Інтегровані середовища розробки, що надають функціонал з редагування, компіляції, навігації вихідним кодом, виконання, відлагодження програм, частіше за все містять обмежений набір функцій, що достатній для розробки програмного забезпечення, але не надає засобів автоматизації процесів рефакторингу.

Деякі інструменти з візуалізації вихідного коду вбудовуються розробниками в середовища розробки, але більшість таких інструментів для роботи з вихідним кодом програми користуються власними реалізаціями текстових редакторів та інших функцій, таких як підсвічування синтаксису.

Цей підхід має значні вади, оскільки означає, що хоч і можливо побудувати потужний засіб візуалізації, але сфера застосування таких засобів дуже обмежена. Найяскравішоб ознакою гарної архітектури є можливість повторного використання її компонентів. До цього критерію також входить незалежність від апаратного забезпечення, операційної системи та об’єкту інтеграції. Гарно написаний модуль для такого важливого функціоналу має давати можливість використовувати його для будь-якої системи, середовища розробки та мови програмування.

Вирішення цієї проблеми пропонується здійснити за допомогою розробки засобу автоматизації відлагодження вихідного коду високої зв’язності, що дозволить користувачу отримати інформацію про стуктуру алгоритму та поведінку програми у простому графічному форматі, а також буде давати можливість для подальшого розширення функціоналу та інтеграцій з популярними середовищами розробки.

Серед мов програмування, що більш-менш активно використовуються зараз, яскравим прикладом мови з важливою областю застосування та довгим життєвим циклом продукту, а також синтаксисом, що слабко обмежує «брудні» архітектурні підходи, є мова FORTRAN. Додатковою перевагою використання цієї мови з метою розробки прототипу є порівняна простота синтаксису та невелика кількість мовних конструкцій порівняно з більш сучасними мовами.

Таким чином, наукова новизна даної роботи полягає у розробленій програмній архітектурі, що забезпечує крос-платформеність, заохочує масштабування та інтеграцію засобу з зовнішніми системами. Окрім цього, було розроблено метод стислого представлення алгоритмів, що дозволяє значно зменшити навантаження на систему при збереженні, передачі та обробленні інформації.

Метою роботи є створення інструменту візуалізації вихідного коду за допомогою блок-схем з можливістю багаторазового використання розроблених модулів.

Об’єктом дослідження є процес відлагодження та рефакторингу програм. Предметом дослідження є використання графічних візуалізацій з метою автоматизації процесу відлагодження та рефакторингу програм.

У процесі проведення досліджень використані наукові методи: аналіз та синтез – для виділення з предметної області рефакторингу програм наявних проблем та формування з них задачі з розробки засобу, абстракція та узагальнення – у процесі формування вимог до архітектури системи.

Апробацію результатів досліджень, проведених у рамках дисертації, було здійснено на XXI Всеукраїнській науково-технічній конференції молодих вчених, аспірантів та студентів «Стан, досягнення та перспективи інформаційних систем і технологій» у доповіді «Метод стислого представлення алгоритмів» у 2021 році, м. Одеса.

Публікацію результатів досліджень було здійснено у науковому журналі «Проблеми програмування» №1, 2021, м. Київ, за темою «Засіб вимірювання метрик вихідного коду Fortran за допомогою синтаксичного аналізу».

1 АНАЛІЗ НАЯВНИХ РІШЕНЬ

Візуалізація алгоритмів, структур даних та архітектури системи використовувалися для полегшення розробки та аналізу складних програмних систем з самого виникнення індустрії програмного забезпечення. За час її існування було розроблено велику кількість інструментів для побудови блок-схем алгоритмів та діаграм загалом. Та принцип функціонування та область застосування таких засобів значно відрізняються між собою в залежності від їх призначення. Діаграми є корисним інструментом в набагато більш широкому сенсі, ніж їх застосування для аналізу алгоритмів. Найпоширеніший стандарт відображення діаграм – UML – визначає 14 різних типів діаграм для представлення структури, поведінки та взаємодії компонентів системи. Для роботи з діаграмами за цим стандартом необхідно обрати правильний програмний інструмент. Такі інструменти не тільки містять графічні примітиви UML та стрілки для поєднання їх у діаграми, вони зазвичай платформо-незалежні та можуть здійснювати експорт діаграм у велику кількість форматів. Певні такі інструменти також дозволяють здійснювати імпорт вихідного коду та автоматичну побудову діаграм на його основі або створення вихідного коду з побудованої діаграми.

1.1 Способи роботи з блок-схемами

Блок-схеми у стандарті UML найближче відповідають діаграмам активності (activity diagram), але незалежно від них визначають набір графічних елементів, що входять до їх складу [1]. Засоби для роботи з блок-схемами зазвичай поділяються на типи в залежності від їх головного функціоналу:

- інструменти для побудови діаграм вручну. Такі програмні засоби фокусуються на можливості користувача додавати, видаляти, переміщувати окремі елементи блок-схеми для побудови з них діаграми алгоритму.

- інструменти для генерації діаграм на основі певної мови розмітки. Такі інструменти перетворюють файл з описом елементів діаграми та експортують побудовану структуру, зазвичай у вигляді зображення.

- інструменти для перетворення вихідного коду на набір елементів які можна потім вивести як діаграму. Екпорт структури з таких елементів зазвичай здійнюється в файл певною мовою розмітки.

- інструменти для зворотнього інжинірингу діаграм у набір програмних структур та генерації вихідного коду на їх основі. Виконують роботу, зворотну до наведених вище інструментів.

Найбільш розпосвюдженим представником першої групи інструментів у системі Windows є MS Visio - інструмент для створення діаграм та векторної графіки довільної складності. Він надає інструменти з створення детальних організаційних схем, блок-схем, діаграм UML, планів будівель та багатьох інших типів візуальних представлень. Він поставляється як частина пакету MS Office із вбудованими шаблонами, які допомагають створювати схеми будь-якої складності, такі як фігурні ілюстрації або комплекні малюнки.

Переваги використання Microsoft Visio:

- надає інструменти для простого прототипування архітектури програмних систем або баз даних;
- широкий спектр доступних шаблонів та елементів;
- великий інструментарій ручної побудови діаграм: переміщення, групування, масштабування елементів, зміна кольорів та форм.

Недоліки Visio:

- закритий вихідний код та висока ціна використання
- відсутність можливостей виконання зворотної інженерії баз даних та програм
- працює лише на операційній системі Windows

Друга група інструментів представлена системами, що побудовані навколо мов розмітки – з розповсюджених систем можна навести GraphViz, Mermaid та PlantUML.

GraphViz - це програмне забезпечення для візуалізації графів з відкритим кодом. Візуалізація графів - це спосіб представлення структурної інформації як

діаграм абстрактних структур та мереж. Він має важливе застосування в мережах, біоінформатиці, програмній інженерії, базі даних та веб-дизайні, машинному навчанні та у візуальних інтерфейсах для інших технічних областей. Блок-схеми також являютть собою спрямовані графи.

Програми макета Graphviz беруть описи графіків виконані за допомогою простої текстової мови та складають схеми в зручних форматах, таких як зображення та SVG (векторні рисунки) для веб-сторінок; PDF або Postscript для включення в інші документи. Graphviz має багато корисних функцій для конкретних діаграм, таких як параметри кольорів, шрифтів, макетів табличних вузлів, стилів ліній, гіперпосилань та спеціальних фігур [2].

Mermaid – інструмент створення діаграм та графіків на основі Javascript, який виконує відображення Markdown-подібного тексту для динамічного створення та модифікації діаграм. Цей інструмент інтегровано у сайт системи контролю версій GitHub для можливості групового перегляду та редагування діграм, пов’язаних з розроблюваним проектом [3]. Mermaid підтримує такі типи діаграм:

- блок-схеми
- діаграми послідовності
- діаграми Ганта
- діаграми класів
- графи комітів та гілок Git

PlantUML – бібліотека, що надає функціонал з побудови великої кількості діаграм – як UML, так і інших типів, зокрема:

- діаграми послідовності;
- діаграми варіантів використання;
- діаграми класів;
- діаграми компонентів;
- діаграми станів;
- дані json, yaml, ASciiMath, LaTeXMath;
- діаграми Ганта;
- діаграми структурного поділу роботи (WBS),

та здійснює експорт побудованих структур даних у великій кількості форматів:

- растрові зображення (jpg, png);
- векторні зображення svg;
- LaTeX формат зображень;
- для діаграм послідовності створено експорт у ASCII art.

Для побудови використовується мова PlantUML [4], що подібна до мови DOT у Graphviz.

Третя група інструментів зазвичай не виноситься в окремі модулі, а працює у складі комплексних систем, що будують діаграми на основі вихідного коду різних мов програмування. Проте ця група все ж варта згадки, оскільки інструмент що має характерний для цієї групи функціонал, став основною причиною вибору цієї теми для дослідження.

Fortran 77 Flowcharting Utility – простий інструмент, що був розроблений як допоміжний інструмент для рефакторингу програм мовою FORTRAN. Суть цього інструменту полягає у парсингу вихідного коду FORTRAN версії 77 та представленні його у вигляді файлу розмітки мовою DOT утиліти Graphviz описаної вище [5].

Особливістю цього інструменту є те, що хоч він і є наздвичайно примітивним та простим за своєю структурою, це один з дуже небагатьох доступних інструментів для рефакторингу вихідног коду FORTRAN77. Завдяки використанню цього інструменту робота з визначення функціоналу програм стає набагато легшою, оскільки особливістю синтаксису та характерною рисою програм FORTRAN77 є надмірне використання оператору GOTO, що робить вихідний код дуже заплутаним та складним для розуміння.

Чертверта група інструментів представлена зазвичай окремими елементами функціоналу інтегрованих середовищ розробки, які займаються генерацією вихідного коду за певними шаблонами. Також до цієї групи можна віднести візуальні мови програмування, наприклад популярну у навчанні школярів початкових знань з інформатики системи Scratch.

1.2 Способи взаємодії з середовищами розробки

Робота з блок-схемами в рамках процесу розробки програмного забезпечення може відрізнятися в залежності від використаних для цього інструментів. Всі групи інструментів, описані в минулому підрозділі, існують окремо від середовищ розробки та мають використовуватися разом з іншими технологіями або вручну.

Інструменти для перетворення вихідного коду на діаграми та їх побудови зазвичай дотримуються іншого принципу – вони інтегровані з текстовим редактором або середовищем розробки. Найпростішим прикладом такої поведінки є операція побудови діаграми класів у Visual Studio. Вона вбудована до середовища розробки, використовує його внутрішнє представлення структури проекту та виводить його у вікні на робочій панелі.

Більшість інструментів, що спеціалізуються тільки на побудові блок-схем, мають в основі потужний механізм побудови діаграм, але реалізовані навколо окремо написаного текстового редактора [6, 7, 8]. Такий підхід дозволяє бути незалежним від середовища розробки, але обмежує можливі сценарії використання такої системи та вимагає ручного перенесення вихідного коду між текстовими редакторами. Прикладами таких засобів є code2flow, Visustin v8 та Source Code to Flowchart tool від fatesoft. Всі вони підтримують редагування, підсвічування синтаксису та побудову діаграм для широкого спектру мов програмування, але на цьому їх функціонал закінчується.

Для уникнення проблем двох описаних вище підходів до взаємодії з середовищами розробки можливим аналогом є створення плагінів (розширень) для середовищ програмування. Більшість крупних та розповсюджених IDE та текстових редакторів (Visual Studio, Eclipse, інструменти групи Jetbrains, code:blocks, notepad++ та інші) підтримують підключення таких розширень, що може дозволити включити функціонал побудови діаграм у відповідні системи.

1.3 Підсумки

Проаналізувавши описані вище відмінності та особливості у роботі різних інструментів для побудови та роботи з діаграмами, а також конкретні приклади таких інструментів, їх бізнес-моделі, спрямування та функціонал, можна зробити висновок що підходи до роботи з діаграмами в цілому та конкретно з блок-схемами змушують розробників цих інструментів обмежувати функціонал розроблюваних засобів та йти на компроміси. В цьому полягає актуальність даної роботи. Не зважаючи на велику кількість спроб створення уніфікованих інструментів роботи з блок-схемами, всі вони обмежені залежностями від цільової платформи, способів вводу інформації, систем взаємодії з візуальним представленням готової блок-схеми або деталей реалізації середовища розробки.

Для забезпечення гнучкості розроблюваного продукту необхідно переконатися у тому що кожен його компонент може використовуватися окремо та незалежно від інших. Це сприяє покращенню якості продукта внаслідок полегшення внесення змін до окремих модулів, можливості багаторазового використання одний і тих самих технологій та підходів незалежно від платформи, мови програмування, середовища розробки та виконання. Створення базової модульної архітектури для такої заплутаної системи – єдиний спосіб забезпечити простоту розробки та використання розроблюваного інструменту.

2 ФОРМУВАННЯ ВИМОГ ДО СИСТЕМИ

2.1 Призначення системи

Розроблюваний засіб буде являти собою інструмент, що дозволить користувачеві отримувати інформацію про внутрішню структуру алгоритму та потік керування програми у графічному вигляді. Цю інформацію можна використовувати для відлагодження програм, реінжинірингу, рефакторингу та аналізу архітектури та алгоритмів [9].

Метою створення засобу є автоматизація перетворення вихідного коду програми в структуровану форму, виділення закономірностей та патернів в потоці керування, що має полегшити роботу програміста з проектування архітектури, взаємозв’язків модулів та частин програми.

Задля досягнення цієї мети засіб має виконувати наступні задачі:

- виведення потоку керування програми на екран у вигляді діаграми
- динамічне оновлення діаграми у відповідності до оновлення коду
- переміщення діаграми екраном, зміна масштабу
- можливість перетягувати та міняти місцями елементи діаграми
- перехід від програмної конструкції до віповідного елемента діаграми
- зворотний перехід від елемента діаграми до блоку коду
- здійснення пошуку елементів за внутрішніми даними
- групування елементів діграми (у блоки що відповідають групованим лексемам мови програмування)
- відлагодження програми з допомогою емуляції логічних переходів

2.2 Складники системи

У відповідності до списку поставлених задач та особливостей предметної області, перелічених у минулому підрозділі, для реалізації засобу необхідно визначити три підсистеми:

- модуль побудови діаграми;

- модуль керування вихідним кодом;
- модуль відображення.

Модуль побудови діаграм – набір програмних компонентів, що визначають методи конвертації внутрішнього представлення графу потоку керування у вигляді діаграми. Основними задачами є:

- розпізнавання базових елементів діаграм на основі внутрішнього представлення програми;
- побудова даіграм;
- розміщення вузлів та зв’язків між блоками діаграми;
- можливість динамічного оновлення частини графа;

Модуль керування вихідним кодом – набір програмних компонентів (можливо – у вигляді плагіна-розширення або частини IDE), що призначені для здійснення операцій над текстом програми, а також надання користувацького інтерфейсу на стороні текстового редактора. Основні задачі:

- парсинг програмних структур вихідного коду;
- перетворення набору структур на внутрішнє представлення потоку керування;
- прийом та виконання команд від модуля відображення;
- виконання операцій над вихідним кодом.

Модуль відображення – системний сервіс, що використовує модуль побудови діаграм та керує відображенням інформації, яку отримує від модуля керування вихідним кодом. Основні задачі:

- надання інтерфейсу користувача на стороні діаграм;
- відображення діаграми за допомогою графічних елементів;
- забезпечення передачі інформації від користувача до інших модулів;
- синхронізація вихідного коду та діаграми;
- забезбечення можливостей навігації.

2.3 Вимоги до модуля побудови діаграм

Масштабованість.

Оскільки модуль побудови діаграм відповідає за внутрішнє преставлення алгоритму та на основі його роботи формулюються обмеження накладені на інші модулі, необхідно забезпечити програмний інтерфейс та структури даних які дозволять здійснювати конвертацію між цим внутрішнім представленням, вихідним кодом програми та набором графічних елементів, з яких побудовано діаграму. Одною з основних вимог до представлення потоку керування є можливість відтворення алгоритмів довільного об'єму та складності.

Швидкодія.

При дотриманні вимоги про масштабованість програмний засіб теоретично може бути використаний для роботи з вихідним кодом та діаграмами довільного об'єму. Це означає що критичним є питання швидкодії модуля побудови діаграм, тож необхідно забезпечити належну швидкість конвертації структурного графа потоку керування в діаграму, а також можливість оновлення частини діаграми без глобальної перебудови структури даних.

Пропускна здатність.

Оскільки модуль являє собою програмний інтерфейс, що може бути інтегрований у набір інструментів, передбачається необхідність передачі даних про представлення алгоритмів та діаграм з використанням різних режимів зв'язку – безпосереднього виклику програмного інтерфейсу, локальних сервісів, мережі Інтернет тощо. Це призводить до необхідності забезпечити якомога більш компактне представлення внутрішніх даних як супутньої вимоги до швидкодії підсистеми.

2.4 Вимоги до модуля керування вихідним кодом

Зручний та ергономічний користувацький інтерфейс.

Користувач має мати змогу сповіщати модуль відображення про зміни у вихідному коді, про активний (вибраний в текстовому редакторі) сегмент програми

та блок вихідного коду. При ініціалізації побудови діаграми необхідно надати можливість вибрати цільовий контекст (файл, функцію, блок коду).

Дотримання протоколу комунікації.

Забезпечення вимог до модуля побудови діаграм вимагає створення особливого протоколу представлення структурного графа потоку керування програми. Необхідно забезпечити перетворення вихідного коду з дотриманням цього протоколу, що може ускладнити алгоритм конвертації.

Стійкість до помилок.

Функціонування модуля побудови діаграм напряму залежить від точності даних, що передаються на вхід до алгоритму побудови. Необхідно точно визначати коректність введеного користувачем програмного коду та за неможливості відображення помилкової конструкції виключити відповідну ділянку коду з графа програми.

2.5 Вимоги до модуля відображення

Зручний та ергономічний користувацький інтерфейс.

Необхідно надати користувачу простий та гнучкий набір операцій взаємодії з даграмами та навігації структурним графом програми. В користувача має бути змога змінювати положення вузлів діаграми, здійснювати ручне оновлення діаграми, також необхідно надати можливість повної перебудови. В контексті режиму ручного відлагодження, необхідно надати набір віртуальних інструментів для емуляції відтворення алгоритму.

Інтеграція з зовнішніми системами.

Модуль відображення має містити інтерфейс за допомогою якого можливо здійснювати передачу даних з зовнішнього процесу через систему сервісів операційної системи. Багато IDE мають у своєму складі інструменти для створення розширень та взаємдії з вихідним кодом, за допомогою яких можливе створення для таких систем відповідних модулів керування вихідним кодом та підключення їх до модуля виведення.

Також необхідно передбачити можливість експорту побудованих діаграм у довільні формати даних (малюнки, таблиці, розповсюджені мови розмітки діаграм).

Швидкодія.

Необхідно передбачити моживість подання на вхід модуля діаграм довільної складності, що несе за собою необхідність мінімізації накладних витрат на відображення, взаємодію з діаграмами та передачу даних під час використання режиму ручного відлагодження.

2.6 Підведення підсумків. Функціональні та нефункціональні вимоги

Таблиця 2.1 – Функціональні вимоги

Підсистема	Вимога
Модуль побудови діаграми	Можливість обробки алгоритмів довільного розміру та складності
	Якомога менша затримка при побудові та перебудові діаграм
	Якомога менше навантажена інфраструктура передачі даних між модулями
Модуль управління вихідним кодом	Ергономічні та інтуітивні елементи інтерфейсу для вибору та зміни цільового сегмента вихідного коду
	Надійність та точність алгоритму перетворення вихідного коду на представлення алгоритму
	Стійкість до помилок у вихідному коді
Модуль відображення	Ергономічні та інтуітивні елементи інтерфейсу для взаємодії з графом в режимі реального часу
	Програмний інтерфейс для можливості інтеграцій з зовнішніми системами
	Якомога менша затримка при відображенні та взаємодії з діаграмами

Таблиця 2.2 – Нефункціональні вимоги

Підсистема	Вимога
Модуль побудови діаграми	Визначення основного шляху через алгоритм
	Побудова вузлів діаграми та зв’язків між ними
	Розміщення вузлів діаграми відносно одне одного
	Динамічне додавання, видалення та оновлення вузлів
Модуль управління вихідним кодом	Розбір синтаксичних структур вихідного коду
	Перетворення вихідного коду на внутрішнє представлення алгоритму
	Передача модулю відображення інформації про представлення алгоритму
	Генерація сповіщень про дії користувача (наприклад, вибір активного блоку вихідного коду)
	Генерація сповіщень про зміни в вихідному коді
Модуль відображення	Відображення діаграми за допомогою графічних елементів
	Можливість рухати діаграму та індивідуальні вузли інтерфейсу взаємодії з діаграмою алгоритму
	Прийом та передача інформації про зміни в діаграмі
	Передача сповіщень про вибір вузла
	Пошук елемента за текстовим запитом
	Історія навігації
	Вибір стартового вузла для емуляції відлагодження
	Команди «емулювали наступний крок», «повернутися на попередній крок»
	Вибір результату умовного переходу

3 СЦЕНАРІЇ ВИКОРИСТАННЯ СИСТЕМИ

Інтерфейс засобу відлагодження вихідного коду складається з інтерфейсу керування вихідним кодом, що за задумом вбудовується в текстовий редактор та інтерфейсу взаємодії з побудованою на основі вихідного коду діграмою алгоритму, що являє собою самостійний застосунок. Тому у системі можна виділити два способи взаємодії – взаємодія з вихідним кодом та взаємодія з побудованою діаграмою.

Виходячи з наведених типів взаємодії з системою, передбачається наявність одного актора – користувача, та двох підсистем:

- плагін до редактора вихідного коду
- застосунок взаємодії з діаграмою

3.1 Плагін до редактора вихідного коду

До стандартного функціоналу редактора вихідного коду необхідно реалізувати кілька додаткових функцій інтерфейсу що дозволять ініціювати побудову та оновлення діаграми, навігацію вихідним кодом та повідомлення користувача у випадку виникнення виключних ситуацій.

Початкова ініціалізація модуля відображення та побудова діаграми відбуватиметься при виборі опції «побудувати діаграму». Плагін зробить спробу підключення до модуля відображення та створить новий екземпляр, якщо активного не було знайдено.

При первинній побудові діаграми чи в будь-який момент після цього користувач може здійснити вибір активного сегменту вихідного коду. Таким сегментом можуть бути різні структурні рівні якими представлено програмний код у системі, до якої зроблено плагін:

- весь проект;
- група файлів вихідного коду;
- окремий файл вихідного коду;
- группа підпрограм;

- окрема підпрограма;
- група синтаксичних тверджень;
- окреме синтаксичне твердження.

При здійсненні вибору сегмента модуль надсилає дані про обраний сегмент для внесення змін до області побудови/відображення діаграми.

При виборі певного елемента (блоку, виразу) у вихідному коді, модуль надсилає сповіщення для вибору на діаграмі відповідного елемента.

При виборі опції «оновити діаграму» модуль здійснює аналіз вихідного коду для виявлення змін з моменту останнього оновлення діаграми, та надсилає такі зміни для синхронізації діаграми з вихідним кодом.

В процесі аналізу вихідного коду та перетворення його на внутрішнє представлення алгоритма важливо слідкувати за станом системи. Під час внесення змін у код можливе допущення помилок користувачем, в такій ситуації генерація діаграми або її частин може стати неможливою. Необхідно виконувати відловлення таких ситуацій та повідомляти користувача про неможливість побудови діаграми.

Програми часто містять ділянки «мертвого» коду (коду, який не буде виконано за жодних обставин). Необхідно правильно визначати ці ділянки та виключати такий код з інформації що надсилається на побудову діаграми.

У підсумку, сценарії використання для плагіна керування вихідним кодом будуть такі:

- створити діаграму;
- оновити діаграму;
- вибрати активний сегмент;
- перейти до елемента.

3.2 Застосунок взаємодії з діаграмою

Взаємодія та відображення побудованих діаграм відбувається через інтерфейс користувача в окремому графічному застосунку. Подібно до більшості інструментів роботи з діаграмами користувач може переміщати діаграму по екрану, міняти

масштаб для огляду загальної структури або конкретних елементів та міняти взаємне розташування окремих елементів.

Програмні проекти, особливо такі що написані на мовах програмування середнього рівня, можуть містити сотні тисяч рядків коду що на діаграмі стануть десятками тисяч елементів. Орієнтуватися в такій величезній структурі неможливо без використання інструментів навігації. Для досягнення мети цієї роботи достатньо трьох способів навігації – текстового пошуку, переходу між елементами та відміни/повторного виконання дій. Користувач може захотіти здійснити пошук по діаграмі за вмістом текстових полів елементів. При текстовому пошуку всі елементи що містять зазначений текст стануть виділеними.

Також такий розмір діаграми ускладнює співставлення її з вихідним кодом, на якому вона заснована. Для вирішення цієї проблеми кожен елемент має містити опцію «переходу до джерела». При виборі цієї опції здійснюється перехід до відповідного місця в вихідному коді у текстовому редакторі.

При навігації діаграмою та вихідним кодом часто виникає необхідність повернутися до останнього вибраного елемента. Для цього передбачена опція історії навігації.

Окрім переміщення та навігації користувач може здійснити емуляцію виконання алгоритму. Необхідно вибрати початковий елемент, при його виборі стануть активними інструменти емуляції виконання, що дозволяють перейти до наступного/попереднього елемента та здійснити вибір умовного переходу. У підсумку, сценарії використання для застосунку взаємодії з діаграмою будуть такі:

- перемістити діаграму;
- змінити масштаб;
- вибрати елемент;
- перемістити елемент;
- знайти елемент за текстом;
- перейти до джерела;
- відмінити/застосувати дію;
- відтворити алгоритм.

3.3 Опис прецедентів системи

Таблиця 3.1 – Опис прецеденту побудови діаграми

Назва прецеденту	Побудувати діаграму
Підсистема	Модуль керування вихідним кодом
Унікальний ідентифікатор	П_001
Опис	Користувач ініціює процес первинної побудови діаграми
Актори	Користувач
Бізнес значимість	Базовий функціонал, вимоги користувача
Частота використання	Нижче середнього. Як мінімум кожного разу при запуску IDE чи зміні активного проекту
Тригери	Вибрати опцію контекстного меню «створити діаграму»
Передумови	Застосунок відображення вимкнено
Післяумови	Застосунок відображення увімкнено, виведено діаграму відповідну до вибраного активного сегмента
Виключення	Не вдалося запустити або встановити підключення до модуля відображення, виведено повідомлення про помилку

Таблиця 3.2 – Опис прецеденту вибору активного сегмента

I	II
Назва прецеденту	Вибрати активний сегмент
Підсистема	Модуль керування вихідним кодом
Унікальний ідентифікатор	П_002
Опис	Користувач вибирає сегмент вихідного коду для здійснення операцій над ним
Актори	Користувач
Бізнес значимість	Базовий функціонал, вимоги користувача

Продовження таблиці 3.2

I	II
Частота використання	Вище середнього. Може відбуватися незалежно від змін у вихідному коді.
Тригери	Вибрати опцію контекстного меню «створити діаграму»
Передумови	Відкрито файл вихідного коду програми
Післяумови	Активний сегмент записано у пам’ять та буде використано у інших прецедентах.
Виключення	Відсутні

Таблиця 3.3 – Опис прецеденту оновлення діаграми

Назва прецеденту	Оновити діаграму
Підсистема	Модуль керування вихідним кодом
Унікальний ідентифікатор	П_003
Опис	Користувач ініціює процес оновлення діаграми
Актори	Користувач
Бізнес значимість	Базовий функціонал, вимоги користувача
Частота використання	Середня. Максимум кожного разу після зміни вихідного коду програми.
Тригери	Вибрати опцію контекстного меню «побудувати діаграму»
Передумови	Модуль відображення увімкнено, таблицю побудовано (П_001)
Післяумови	Оновлено діаграму відповідно до вибраного активного сегменту (П_002), масштаб змінено для розміщення сегменту в центрі екрану. Якщо активний сегмент не вибрано, діаграма оновлюється для всього доступного вихідного коду.
Виключення	Не вдалося або встановити підключення до модуля відображення, виведено повідомлення про помилку

Таблиця 3.4 – Опис прецеденту переходу до елемента

Назва прецеденту	Перейти до елемента
Підсистема	Модуль керування вихідним кодом
Унікальний ідентифікатор	П_004
Опис	Користувач здійснює перехід від сегменту коду до його представлення на діаграмі.
Актори	Користувач
Бізнес значимість	Полегшення роботи в системі, вимоги користувача
Частота використання	Вище середнього. Може відбуватися незалежно від змін у вихідному коді.
Тригери	Вибрати опцію контекстного меню «перейдти до елемента...»
Передумови	Вибрано активний сегмент (П_002)
Післяумови	Модуль відображення – масштаб діаграми змінено для розміщення елементу в центрі екрану.
Виключення	Не вдалося або встановити підключення до модуля відображення – виведено повідомлення про помилку. Вибраний елемент не знайдено – оновлення діаграми – повтор спроби запуску.

Таблиця 3.5 – Опис прецеденту переміщення діаграми

I	II
Назва прецеденту	Перемістити діаграму
Підсистема	Застосунок взаємодії з діаграмою
Унікальний ідентифікатор	П_005
Опис	Користувач здійснює переміщення діаграми по екрану для перегляду потрібної області.
Актори	Користувач

Продовження таблиці 3.5

I	II
Бізнес значимість	Базовий функціонал, вимоги користувача
Частота використання	Дуже висока. Основний функціонал.
Тригери	Натиснути ліву клавішу миші поверх поля з діаграмою.
Передумови	Застосунок запущено (П_001)
Післяумови	Діаграму переміщено згідно з діями користувача.
Основний шлях	Натиснути ліву клавішу миші Не відпускаючи, перетягнути мишу Відпустити ліву клавішу миші
Виключення	Відсутні

Таблиця 3.6 – Опис прецеденту зміни масштабу

Назва прецеденту	Змінити масштаб
Підсистема	Застосунок взаємодії з діаграмою
Унікальний ідентифікатор	П_006
Опис	Користувач здійснює зміну масштабу діаграми для перегляду потрібної області.
Актори	Користувач
Бізнес значимість	Базовий функціонал, вимоги користувача
Частота використання	Дуже висока. Основний функціонал.
Тригери	Прогорнути коліщатко миші.
Передумови	Застосунок запущено (П_001)
Післяумови	Змінено масштаб діаграми згідно з діями користувача.
Виключення	Відсутні

Таблиця 3.7 – Опис прецеденту вибору елемента

Назва прецеденту	Вибрати елемент
Підсистема	Застосунок взаємодії з діаграмою
Унікальний ідентифікатор	П_007
Опис	Користувач вибирає елементи діаграми для рооти з ними.
Актори	Користувач
Бізнес значимість	Полегшення роботи в системі, вимоги користувача
Частота використання	Дуже висока. Основний функціонал.
Тригери	Натиснути ліву клавішу миші поверх елемента діаграми.
Передумови	Застосунок запущено (П_001)
Післяумови	Елемент візуально виділено.
Основний шлях	Навести мишу на потрібний елемент Натиснути ліву клавішу миші Відпустити ліву клавішу миші
Виключення	Відсутні

Таблиця 3.8 – Опис прецеденту переміщення елемента

I	II
Назва прецеденту	Перемістити елемент
Підсистема	Застосунок взаємодії з діаграмою
Унікальний ідентифікатор	П_008
Опис	Користувач здійснює переміщення елементу діаграми по екрану для сортування елементів за своїм вподобанням.
Актори	Користувач
Бізнес значимість	Полегшення роботи в системі, вимоги користувача

Продовження таблиці 3.8

I	II
Частота використання	Дуже висока. Основний функціонал.
Тригери	Затиснути ліву клавішу миші поверх елемента діаграми з натиснутою клавішею Ctrl.
Передумови	Застосунок запущено (П_001)
Післяумови	Елемент переміщено згідно з діями користувача.
Основний шлях	Навести мишу на потрібний елемент Натиснути клавішу Ctrl Натиснути ліву клавішу миші Не відпускаючи, перетягнути мишу Відпустити ліву клавішу миші Відпустити Ctrl
Виключення	Відсутні

Таблиця 3.9 – Опис прецеденту пошуку елемента за текстом

I	II
Назва прецеденту	Знайти елемент за текстом
Підсистема	Застосунок взаємодії з діаграмою
Унікальний ідентифікатор	П_009
Опис	Користувач здійснює пошук елемента, текст якого відповідає введеній послідовності.
Актори	Користувач
Бізнес значимість	Полегшення роботи в системі, вимоги користувача
Частота використання	Вище середнього. Частина інфраструктури навігації.
Тригери	Натиснути клавішу «пошук».

Продовження таблиці 3.9

I	II
Передумови	Застосунок запущено (П_001)
Післяумови	Підсвічено наступний елемент, що відповідає введеній послідовності.
Основний шлях	Ввести текст потрібного елемента у поле «пошук» Натиснути кнопку «пошук» Повторити натиск для переходу до наступного результату
Виключення	Елементів не знайдено – вивести сповіщення Пошук закінчено – вивести сповіщення, продовжити пошук

Таблиця 3.10 – Опис прецеденту переходу до джерела

Назва прецеденту	Перейти до джерела
Підсистема	Застосунок взаємодії з діаграмою
Унікальний ідентифікатор	П_010
Опис	Користувач здійснює перехід від елемента діаграми до відповідного йому сегменту коду.
Актори	Користувач
Бізнес значимість	Полегшення роботи в системі, вимоги користувача
Частота використання	Вище середнього. Частина інфраструктури навігації.
Тригери	Натиск на елемент з затиснутим Shift.
Передумови	Застосунок запущено (П_001)
Післяумови	Модуль керування вихідним кодом – курсор встановлено на відповідному сегменті коду.
Виключення	Не вдалося або встановити підключення до модуля керування кодом – виведено повідомлення про помилку.

Таблиця 3.11 – Опис прецеденту відміни/застосування дії

Назва прецеденту	Відмінити/застосувати дію
Підсистема	Застосунок взаємодії з діаграмою
Унікальний ідентифікатор	П_011
Опис	Користувач здійснює перехід до елементів, вибраних в минулому.
Актори	Користувач
Бізнес значимість	Полегшення роботи в системі, вимоги користувача
Частота використання	Вище середнього. Частина інфраструктури навігації.
Тригери	Натиснути комбінацію клавіш вперед/назад
Передумови	Застосунок запущено (П_001)
Післяумови	Виділення змінено для відображення набору блоків, вибраного на певному кроці історії навігації.
Виключення	Історія навігації відсутня – без змін.

Таблиця 3.12 – Опис прецеденту відтворення алгоритму

I	II
Назва прецеденту	Відтворити алгоритм
Підсистема	Застосунок взаємодії з діаграмою
Унікальний ідентифікатор	П_012
Опис	Користувач здійснює емуляцію відтворення алгоритму.
Актори	Користувач
Бізнес значимість	Полегшення роботи в системі, вимоги користувача
Частота використання	Середня. Базовий функціонал.
Тригери	Подвійний натиск на елемент

Продовження таблиці 3.12

I	II
Передумови	Застосунок запущено (П_001)
Післяумови	Вибрано наступний крок з алгоритму
Основний шлях	Натиснути клавішу «стрілка вниз» для переходу на наступний елемент (тільки для послідовності), «стрілка вліво» для переходу на «хибну» гілку умовного переходу, «стрілка вправо» для переходу на «істинну» гілку умовного переходу та «стрілка вгору» для переходу на попередній крок алгоритму.
Виключення	Поточний елемент є останнім – вивести сповіщення, перезапуск алгоритма. Поточний елемент не вибрано – запуск з першого елемента.

Всі описані прецеденти наведені на діаграмі сценаріїв використання наведені в додатку А.

4 ВИБІР ТА ОБҐРУНТУВАННЯ ЕЛЕМЕНТІВ ТА ТЕХНОЛОГІЙ

4.1 Обґрунтування вибору елементів та технологій для модуля керування вихідним кодом

Для розробки цього модуля насамперед необхідно визначити систему в яку буде здійснено вбудування цього модуля та яку мову програмування буде обрано для перетворення на діаграми.

Для обґрунтування такого рішення розглянемо особливості та мету застосування розроблюваного продукту. Системи візуалізації вихідного коду застосовуються здебільшого у випадках, коли програмісту потрібно розібратися у заплутаному коді з високою зв’язністю. Термін «зв’язність» у застосуванні до коду та архітектури програм означає високий рівень залежності елементів програми між собою [10]. Це ускладнює внесення змін в архітектуру програми, оскільки навіть найменша зміна може «потягнути» за собою зміни в інших ділянках коду. В таких умовах графічне відтворення алгоритму майже неминуче, оскільки для великих проектів стає неможливим утримання в голові інформації про структуру програми та функції кожної окремої підсистеми.

Такі радикальні проблеми мають тенденцію поставати у середовищах, де вимоги до стилю програмування змінюються швидше ніж життєвий цикл більшості продуктів. Це характерно для старих мов програмування, більшість програмного забезпечення для яких було написане до формулювання понять «стиль програмування», до популяризації методик та патернів програмування та появи самостійної професії програміста.

Хрестоматійним прприкладом такої екосистеми є мова програмування FORTRAN. Як мова, що з’явилася більше п’ятидесяти років тому, але й досі не втратила своєї актуальності, вихідний код більшості крупних проектів на FORTRAN починав писатися в часи, коли вимог до якості вихідного коду як таких ще не було [11]. Зважаючи на особливості сфер застосування та деколи величезний обсяг програм, написаних на FORTRAN (фізичні симуляції, системи реального часу, математичне моделювання складних процесів), а також на порівняну простоту

синтаксису мови, застосування графічних візуалізацій для полегшення процесів рефакторингу, реінжинірингу та перекладу на інші мови видається явно необхідною задачею та матиме значний позитивний вплив на галузь.

В рамках даної роботи проведено розробку двох незалежних модулів керування вихідним кодом для демонстрації гнучкості підходу. Ці модулі призначені для роботи з двома формами представлення, прийнятими у мові FORTRAN. Ці представлення носять назви «вільна форма» (free form) та «фіксована форма» (fixed form) [12]. Деталі представлення вихідного коду в цих формах описано в розділі 5.

При розробці модуля керування вихідним кодом вигідно реалізувати його як плагін до певної IDE. Таким чином можливо уникнути роботи з власноручного написання надлишкових функцій (текстовий редактор, препроцесор, синтаксичний аналізатор, парсер мови тощо), а також більше розкрити потенціал модульної архітектури, що була вибрана в цій роботі за рахунок інтеграції засобу в зовнішню систему.

Існує велика кількість IDE для мови FORTRAN, більшість з яких не підходять для інтеграції з засобом. Перелік найбільш розповсюджених середовищ розробки наведено в таб. 4.1.

Таблиця 4.1 – Порівняння функціоналу та стану підтримки IDE для FORTRAN

Назва середовища розробки	Стан підтримки	Основний функціонал
Understand	В активній розробці	Інтегроване середовище, що містить функціонал для навігації вихідним кодом та сфокусоване на створенні візуальних звітів по структурі проекту.
SCSE	В активній розробці	Потужний інструмент для навігації вихідним кодом, основним функціоналом є виконання пошукових запитів на великих об’ємах даних.
FPT	Остання активність 07.2020	Технічний аналізатор вихідного коду з фокусом на міграціях програм між

		різними середовищами виконання та операційними системами.
Forcheck	В активній розробці	Технічний аналізатор вихідного коду, що підтримується вже більше 30 років, з фокусом на пошуку неявних помилок у тексті програм.
Fortran Analyzer	Остання активність 09.2019	Стилістичний аналізатор вихідного коду з фокусом на альтернативних метриках, що вказують на «чистоту» коду, таких як глибина вкладених циклів, документованість програмних елементів (файлів, підпрограм, модулів).

Всі перелічені середовища розробки не дають можливості інтеграції програмних модулів, бо є комерційними продуктами або більше не підтримуються розробниками. Найкращим кандидатом для інтеграції є середовище розробки Photran.

Photran - це розвинене крос-платформене інтегроване середовище розробки для Fortran на основі Eclipse. Photran має ряд потужних функцій. Як IDE, він інтегрує редагування, навігацію вихідним кодом, компіляцію та налагодження в синхронізовану систему. Для компіляції програм використовується утиліта make, що дозволяє йому працювати у зв’язці з майже будь-яким компілятором Fortran. Також у до проекту включені так звані парсери помилок, що інтерпретують повідомлення про помилки від популярних компіляторів та співставляють маркери помилок з відповідними рядками коду. Система навігації вихідним кодом дозволяє програмісту Fortran швидко знайти підпрограму або модуль з певною назвою, або знайти всі посилання на певну змінну або підпрограму.

Оскільки обрана система є системою з відкритим кодом (open source) та працює на базі Eclipse, що в свою чергу дозволяє розробку плагінів мовою Java, ця мова буде використана у процесі розробки.

Єдиним мінусом середовища Photran є те що в ньому відсутня підтримка фіксованої форми запису FORTRAN-програм, яка була основним стандартом до версії FORTRAN90 та програми на якій ще більше потребують графічних візуалізацій, оскільки використовують старішу версію мови з більш машинно-орієнтованим синтаксисом.

Для уможливлення підтримки зображення таких діаграм необхідно розробити додатковий модуль, який буде працювати виключно з програмами у фіксованій формі. Для реалізації такого модуля зручно застосувати мову програмування C#, а точніше фреймфворк WPF.

WPF розшифровується як Windows Presentation Foundation. Це потужний фреймворк для побудови додатків Windows. WPF був вперше представлений у версії .NET framework 3.0, а потім стільки інших функцій було додано в наступні версії .NET framework.

У ранніх фреймворках графічного інтерфейсу не було реального розділення між тим, як виглядає програма, і її поведінкою. І графічний інтерфейс, і поведінка були створені однією мовою (у випадку .NET застосунків C # або VB.Net), що вимагало більше зусиль від розробника для реалізації як інтерфейсу користувача, так і поведінки, пов’язаної з ним.

У WPF елементи інтерфейсу розробляються в XAML, тоді як поведінка може бути реалізована на процедурних мовах, таких як C # та VB.Net. В результаті стає дуже легко відокремити поведінку від дизайнерського коду.

За допомогою XAML програмісти можуть працювати паралельно з дизайнерами. Розмежування між графічним інтерфейсом та його поведінкою може дозволити розробнику легко змінити вигляд елемента керування за допомогою стилів та шаблонів.

За допомогою цього інструментарію було розроблено простий текстовий редактор для роботи з фіксованою формою FORTRAN. На його базі можливо реалізувати основний функціонал, описаний в розділі 2.

4.2 Обґрунтування вибору елементів та технологій для модулів побудови діаграм та відображення

Головною перевагою модульної архітектури, що було вибрано для організації системи, є можливість багаторазового використання написаних модулів. В ідеальних умовах модуль відображення має бути цілком незалежним компонентом, який можна завантажити, запустити та підключити до нього модуль керування вихідним кодом за допомогою мережевого протоколу. В таких випадках велику порцію уваги приділяють вибору технології з акцентом на її крос-платформеність, легкість в налаштуванні, перелік доступних бібліотек, простоту роботи з інтерфейсом. Розглянемо найпопулярніші технології що відповідають таким вимогам:

Java. Розробка Java була почата Джеймсом Гослінгом в роботі над проектом під назвою "Oak" у червні 1991 р. Цілями Гослінга було сторення віртуальної машини та мови, яка мала б звичний С-подібний синтаксис, але з більшою одноманітністю та простотою, ніж C / C ++. Проект давав обіцянку дотримуватися принципу "Писати один раз, виконувати всюди" на популярних платформах. Мова та віртуальна машина були досить безпечними та цю безпеку завжди можна було налаштувати, що дозволяло обмежити доступ до мережі та файлів.

Через свої особливості Java присутня скрізь. Виходячи з частки цифрових продуктів на ринку (11%) [14] та постійно зростаючого співтовариства Java, IT-фахівці активно використовують цю технологію. Розробки Java присутні майже у всіх галузях бізнесу.

Проте останні кілька років у світі програмістів нерідкими є обговорення з приводу майбутнього Java. Перша хвиля обговорень стосувалася її життєздатності. Багато програмістів вірили і досі вважають, що Java поступово помирає через свою складність порівняно, наприклад, з Python або іншими мовами зі схожими функціями. Ще однією гарячою темою були особливості володіння Java Oracle, що обмежило її безкоштовне використання в комерційних цілях. Для мети цього проекту Java підтримує крос-платформеність та має фреймворк для створення графічного інтерфейсу Swing.

C#. Мову C# було створено компанією Microsoft, яка й сьогодні здійснює підтримку та розробляє нові стандарти для цієї мови та платформи .Net, яка слугує екосистемою при роботі з C#.

Пакет .NET Framework, на якому ведеться розробка застосунків під Windows має деякі обмеження. Наприклад, він працює лише на платформі Windows. Крім того, для різних пристроїв Windows потрібно використовувати різні API .NET, таких як Windows для PC, Windows Store, Windows Phone та веб-застосунків. На додаток до цього .NET Framework - це загальномашинна система. Будь-які внесені до нього зміни впливають на всі застосунки, що залежать від нього.

Для розробки крос-платформених застосунків можна використовувати пакет .NET Core. .NET Core - це нова версія .NET Framework, яка є безкоштовною платформою загального призначення з відкритим кодом, що підтримується корпорацією Майкрософт [15]. Це крос-платформний фреймворк, який працює на операційних системах Windows, macOS та Linux. Він підтримує мови програмування C #, F # та Visual Basic для розробки програм. Також наявна підтримка підходу до модульної архітектури з використанням пакетів NuGet. За допомогою менеджера можна завантажити різні пакети NuGet для різних функцій, які за необхідності можна додати до проекту .NET Core. Сама бібліотека .NET Core також надається як пакет NuGet. Пакет NuGet для моделі додатка .NET Core за замовчуванням - це Microsoft.NETCore.App.

JavaScript. Node.js - це платформа побудована на середовищі виконання Chrome JavaScript для легкої побудови швидких та масштабованих мережевих додатків. Node.js використовує керовану подіями модель неблокуючого вводу-виводу, що робить її легкою та ефективною, ідеально підходить для додатків у режимі реального часу, що працюють на розподілених пристроях.

Node.js - це середовище виконання з відкритим кодом для розробки серверних та мережевих додатків. Додатки Node.js написані на JavaScript і можуть запускатися в середовищі виконання Node.js в OS X, Microsoft Windows та Linux.

Хоча Node.js відомий як серверна платформа, інтерес до його використання для створення настільних додатків швидко зростає. Існують прив’язки для наборів

інструментів графічного інтерфейсу, таких як GTK +, QT та Cocoa. Однак однією з переваг використання Node.js для веб-розробки є можливість використання однієї мови як на сервері, так і на клієнті.

Можливо використовувати Node.js разом із оболонкою веб-браузера для розробки міжплатформенних настільних додатків та ігор за допомогою WebGL – з використанням HTML, CSS та JavaScript.

Python. Python був розроблений Гідо ван Россумом наприкінці 1980-х років, включаючи все те, що він вважав вдосконаленням мови програмування ABC. Python - це міжплатформна мова: програма Python, написана на комп’ютері Macintosh, працюватиме в системі Linux і навпаки. Програми Python можуть працювати на комп’ютері з Windows, якщо на комп'ютері з Windows встановлений інтерпретатор Python (більшість інших операційних систем постачаються з попередньо встановленим Python). Існує програма під назвою py2exe, яка дозволяє користувачеві «запаковувати» інтерпретатор Python разом з програмою у файл виконання Windows, щоб програма Python, розроблена в іншій системі, могла запускатися на машині Windows без установлення інтерпретатора.

Правила синтаксису Python дозволяють створювати лаконічні програми без написання додаткового коду. У той же час, на відміну від інших мов програмування, Python робить акцент на читабельності коду і дозволяє використовувати ключові слова замість пунктуації. Як і інші сучасні мови програмування, Python також підтримує кілька парадигм програмування. Він повністю підтримує об'єктно-орієнтоване та структуроване програмування. Крім того, його мовні функції підтримують різні концепції функціонального та аспектно-орієнтованого програмування. У той же час Python має систему динамічного типування та автоматичне управління пам'яттю.

Величезний, в порівняннч з іншими мовами, розмір стандартної бібліотеки робить Python навагато більш гнучким. Стандартна бібліотека дозволяє вибирати з широкого спектру модулів відповідно до конкретних потреб розробника. Кожен модуль збільшує кількість функцій програми Python без потреби написання додаткового коду.

Як мова програмування з відкритим кодом, Python допомагає значно зменшити вартість розробки програмного забезпечення. Програміст може використовувати кілька фреймворків Python з відкритим кодом, безплатні бібліотеки та інструменти розробки, щоб скоротити час на створення продукту без збільшення його вартості.

Виходячи з вимог до засобу, що розробляється, найбільш доречним для написання модулю відображення та побудови діаграм використати Python. Ця мова має достатньо багато переваг в порівнянні з іншими саме в нашому випадку, а саме:

- найменша складність прототипування, що дозволить швидко та без написання зайвого коду створити прототип застосунку
- значний обсяг функцій, доступних у стандартній бібліотеці
- дуже простий процес наташтування середовища розробки та готового продукту – не потрібно встановлювати великий перелік програмного запезпечення для запуску коду Python на будь-якій платформі.

гнучка та потужна бібліотека для створення ігор pygame, що дозволяє не витрачати зусиль на реалізацію низькорівневої роботи з графікою та прибирає обмеження, що накладаються фреймворками з розробки візуальних інтерфейсів, такими як Swing чи WPF.

Для написання інтерфейсу користувача буде використано вищеописану бібліотеку pygame, для розгортання його як веб-сервісу – модуль sockets.

Детальніше про ці технології описано в розділі 5.

4.3 Архітектура системи.

Такі інструменти як візуалізатори аглоритму, засоби навігації вихідним кодом, інструменти збору метрик та інші прикладні додатки зазвичай реалізуються простим вбудуванням всього функціоналу в інтегроване середовище розробки. У більшості випадків такий вибір цілком виправданий призначенням компонента, оскільки інструменти навігації, рефакторингу та аналізу коду нерозривно пов’язані з внутрішнім представленням програмного проекту, структурою файлів та реалізацією текстового редактора.

У випадку з візуалізацією вихідного коду ситуація дещо інша. Внаслідок специфіки задачі, процес побудови діаграми при наявності даних про структуру потоку керування програми ідентичне для майже всіх популярний мов програмування. Деякі функції мови (парарельне обчислення, система класів та пакетів) потребують реалізації додаткових методів обробки та виведення діаграм, але навіть такі «бонусні» функції будуть виглядати більш-менш однаково для всіх мов.

Виходячи з вищесказаного можна побачити що реалізація окремого модуля відображення та побудови діаграм це додаткова витрата часу, зусиль програмістів та коштів, особливо враховуючи часом діаметрально протилежні реалізації підключення плагінів до IDE. Має сенс залишити на індивідуальну розробку лише модулі керування вихідним кодом. Застосунок що виконує побудову та відображення діаграми в такому випадку відіграє роль системного сервіса, до якого можливо здійснити підключення та спілкуватися з ним за допомогою наперед визначеного протоколу, що не міняється від середовища до середовища та не залежить від операційної системи. Підхід, коли одна частина програмного засобу являє собою сервіс, а інша здійснює запити та отримує відповіді на них, називається «клієнт-серверна модель» [16].

Опис цього архітектурного підходу корисно почати з прикладу. Продемонструвати роботу за принципом клієнт-серверної архітектури можна подивишись на індустрію розробки програмного забезпечення. Зазвичай коли певному бізнесу потрібно програмне рішення, власники бізнесу звертаються до фірми-розробника програмного забезпечення. Переговори про вимоги та бізнес-цілі програмного продукту власники бізнесу ведуть не безпосередньо з програмістами. Точки зору на продукт як результат та об'єкт розробки у цих двох сторін надто сильно відрізняються щоб дозволити продуктивний діалог. В такій ситуації найчастіше існують посередники (зазвичай навіть більше одного рівня). Замовники продукту ведуть переговори з «власниками продукту» (Product Owners), а ті, в свою чергу, делегують обов’язки з планування самого продукту менеджерам проектів. Така схема дозволяє розробникам співпрацювати з багатьма замовниками, а замовникам – з багатьма розробниками за одноб схемою.

Клієнт-серверна архітектура працює за тим же принципом: вона розділяє завдання між провайдерами послуги, які називаються серверами та користувачами послуг, що називаються клієнтами. Сервер відповідає фірмі-розробнику з прикладу індустрії розробки ПЗ, а клієнт – замовнику.

Подібно до замовника, клієнти зазвичай ініціюють сеанси зв'язку із серверами, які очікують на вхідні запити. Пристрій (ноутбук, планшет, смартфон), або протокол (набір команд та правил обміну інформацією), який клієнт використовує для запиту послуги з сервера, схожі на власників продукту та менеджерів проектів з наведенного прикладу: вони здійснюють посередницьку діяльність між користувачем та постачальником послуг і «знає», як говорити з ними обома.

Стандартна архітектура клієнт-сервер складається з трьох частин:

Front-End: це частина програмного забезпечення, яка взаємодіє з користувачами, навіть якщо вони знаходяться на різних платформах з різними технологіями. Будь-який інтерфейсний модуль в архітектурі клієнт-сервер призначений для взаємодії з усіма існуючими на ринку пристроями. Цей рівень містить екрани входу, меню, екрани даних та звіти, які надають та приймають інформацію до / від користувачів.

У розроблюваному засобі частину front-end займає модуль керування вихідним кодом. Він має бути реалізований для всіх пристроїв, систем та мов програмування, з якими планується здійснити інтеграцію.

Сервер додатків: Це сервер, на якому встановлені програмні модулі системи. Він підключається до бази даних та здійснює взаємодію з front-end частиною. У розроблюваному засобі сервером додатків можна вважати модуль відображення та супутній йому протокол зв'язку.

Сервер бази даних: Цей сервер містить таблиці, індекси та дані, керовані додатком. Тут виконуються операції пошуку та вставки / видалення / оновлення.

Якщо провести аналогію між стандартним додатком з клієнт-серверною архітектурою (інтернет-магазин, хмарна система зберігання файлів тощо) та розроблюваною системою, то аналогом бази даних буде слугувати діаграма. Дійсно,

інформація, що отримується з модуля керування вихідним кодом, впливає в першу чегру на структуру діаграми, дані про яку зберігаються, оновлюються, видаляються.

Переваги використання моделі клієнт-сервер:

Як видно з прикладу, архітектура клієнт-сервер розділяє апаратне забезпечення, програмне забезпечення та функціональність системи. Якщо необхідна адаптація програмного забезпечення для певного середовища (наприклад, внаслідок зміни версії мови програмування), тобто необхідна зміна функціональних можливостей, її можна провести в системі без необхідності переробляти версії для інших мов програмування та середовищ.

Оскільки підхід розділяє зв'язок між апаратним забезпеченням, програмним забезпеченням та функціональністю системи, лише інтерфейс повинен бути адаптований для зв'язку з різними клієнтами.

В підходу також є кілька ключових недоліків:

При одночасному запиті або передачі інформаці з/на сервер багатьма клієнтами, це може призвести до перевантаження сервера та виникнення значних затримок.

Наявність додаткового компоненту системи додає ще один рівень передачі даних, що може бути вразливим до збоїв. Вірогідність пошкодження зв'язку між модулями в монолітному проекті значно менша.

Специфіка задачі цієї роботи на розробку засобу дозволяє значною мірою позбутися вищенаведених мінусів такої архітектури. Оскільки клієнт та сервер працюватимуть на одному пристрої, зв'язок між ними буде набагато менш вразливий до збоїв в мережі, а ситуація з перевантаженням сервера багаторазовими запитами виключається у зв'язку з відсутністю інших клієнтів. Спрощену діаграму компонентів наведено у додатку Б.

4.4 Застосовані патерни проектування

Невід’ємною складовою розробки програмної системи є планування розділеної, масштабованої та незалежної від реалізації архітектури програмних модулів. Це

стосується як високорівневих елементів, наприклад системної архітектури, планування якої описане вище, так і внутрішньої взаємодії модулів.

Під час проектування архітектури програмного модуля чи системи розробник зазвичай вирішує завдання про пов’язування між собою різних частин функціоналу для їх спільної роботи. Перші програмні модулі що мали на меті виконання дуже простих з точки зору сучасного стану індустрії завдань (виконання складних математичних обчислень, чи простих монотонних повторюваних задач) писалися за технікою «в лоб», тобто дуже прямолінійно від вимог до функціональності такого засобу до реалізації його на програмному (а іноді й машинному) коді. З часом, спектр задач, що потрібно реалізувати в програмному продукті ширився, й оскільки таких прямолінійний підхід лишався більш-менш загальновживаним, розроблені програми ставало все більш і більш складно змінювати, оскільки весь функціонал що складає такий модуль, залежить один від одного та вимагає змін (у гіршому випадку) у всіх частинах програми одночасно.

Для розв’язання цієї проблеми розробник починає аналізувати розроблювану систему, виділяти з неї окремі функції та намагатися групувати їх у підсистеми, які потім можуть спілкуватися одна з одною за допомогою певних правил. Таких підхід до архітектурного проектування систем поклад початок багатьох принципів у сучасному програмуванні – починаючи з принципів ООП, а особливо поліморфізму, що становить основу для відділення поведінки та архітектури, продовжуючи принципами SOLID, які слугують зараз практично законами чистої архітектури. Частиною цього розвитку стало формування патернів або шаблонів проектування – архітектурної закономірності або конструкції, що призначена для вирішення певної проблеми проектування [17]. В залежності від призначення використанні цих конструкцій, патерни поділяють на три групи:

- основні (фундаментальні) – патерни проектування, що являють собою найпростіші взаємодії між архітектурними елементами та застосовуються для побудови патернів вищого порядку.

- породжувальні – патерни, що використовуються для узагальнення, делегації або інкаплсуляції інформації при створенні об'єктів або сімейств об'єктів. Використані патерни – фабричний метод.

- структурні – патерни, що використовуються при компонуванні та пообудові ієрархій з об’єктів. Використані патерни – компонувальник, декоратор.

- поведінкові – патерни, що використовують для побудови взаємодії між об’єктами та розділенні обов’язків функціоналу. Використані патерни – спостерігач, відвідувач, делегація.

Делегація (delegation) – фундаментальний патерн проектування, що ще називають принципом проектування, бо ця конструкція лежить в основі багатьох інших патернів. Проблема, що породжує необхідність використання такого патерну полягає в тому, що часто існує необхідність примусити об’єкт виконувати дії над даними або за допомогою алгоритмів, про які об’єкт не має уявлення та які не стосуються безпосередньо його функціональності. Рішенням проблеми стає відділення потрібного функціоналу в окремий клас та застосування композиції для виклику цього функціоналу (делегування) з основного класу.

Фабричний метод (factory method) – породжувальний патерн проектування, метою якого є відділення інтерфейсу створення нових об’єктів від алгоритму їх створення. Проблема, що вирішує патерн – стратегії створення об’єктів, що використовують фреймворки не мають уявлення про конкретні об’єкти, які необхідно створити. Рішення проблеми – делегувати створення об’єктів класам – нащадкам.

Компонувальник (composite) – структурний патерн проектування, що має на меті дозволити створення складної деревоподібної структури подібних одне до одного об’єктів та взаємодіяти з цією структурою як з одним об’єктом. Проблемою, що породжує необхідність застосування патерну є наявність у архітектурі рекурсивних груп елементів (особливо при роботі з графічними інтерфейсами та діаграмами), коли до кожного елемента необхідно застосовувати подібні операції/обмеження. Вирішенням проблеми є створення абстракції що описує одночасно графічні елементи та контейнери, що їх містять. Тоді особлива поведінка,

властива контейнерам, може бути відділена від реалізації взаємодії з конкретними об’єктами та делегована самим класам контейнерів.

Декоратор (decorator) – структурний патерн, застосування якого дає можливість надавати об’єктам додаткової функціональності в ході виконання програми, замінюючи успадкування композицією. Проблемою що вирішує цей патерн є необхідність динамічно створювати об’єкти з різним набором доступних функцій, уникаючи реалізації конкретних класів для кожної можливої комбінації. Рішенням є створення об’єкту-обгортки, що використовує інтерфейс загорнутого елемента, розширюючи чи видозмінюючи його додатковими функціями.

Спостерігач (observer) – поведінковий патерн, що застосовується для заміни прямого виклику функціоналу об’єктів використовуючи інверсію обов’язків, особливо в ситуації коли зміни у стані одного об’єкта мають впливати або ініціювати дії багатьох залежних об’єктів-користувачів. Проблема, що призводить до використання цього патерну полягає у необхідності об’єкта – ініціатора мати доступ до довільної кількості об’єктів-користувачів для повідомлення їх про зміну. Така ситуація є аналогічною до взаємодії видавців газет та підписників (англ. «subscriber/observer», звідки патерн і отримав свою назву. Рішення проблеми полягає в тому, що об’єкт-«видавець» визначає інтерфейс для отримання повідомлень про внутрішні зміни та механізм надсилання запиту на початок або кінець отримання цих повідомлень. Реакція на повідомлення в такому випадку лишається на реалізацію у об’єктах – «підписниках», що дозволяє зменшити «зв’язаність» архітектури.

Відвідувач (visitor) – поведінковий патерн, що дозволяє визначити алгоритм обробки елементів певної структури без потреби змінювати класи цих елементів. Проблема, що призводить до необхідності застосування патерну – при потребі виконання дій або збору інформації з композитної структури з різних типів елементів, алгоритми обробки елементів можуть сильно відрізнятися від типу до типу. В такій ситуації необхідно уникнути написання різних реалізацій обробки всередині класів цих елементів. Рішенням проблеми є виділення алгоритму обробки набору таких елементів у окремий клас. Об’єкт такого класу передається по структурі даних, де

кожен контейнер надає йому доступ до дочірніх елементів, обробка яких виконується незалежно від їх реалізації у тілі об’єкта, що «відвідує» ці елементи.

4.5 Підсумки. Переваги розподіленої архітектури

Вибір архітектури для закладання в основу розроблюваного засобу – відповідалльний етап у життєвому циклі програмного продукту. Від цього вибору залежить те, наскільки довго супровід цього продукту можна буде виконувати без внесення значних змін у його структуру.

Патерни проектування та підходи до проектування систем здебільшого фокусуються на створенні модульної системи з можливістю багаторазового використання її компонентів. Навіть патерни проектування було сформульовано для забезпечення багаторазового використання архітектурних конструкцій та рішень.

Незалежно від обраної архітектури та прийомів проектування, будь-який програмний продукт рано чи пізно потребує рофакторингу. Але використання архітектури, що базується на розділенні обов’язків між класами та побудові взаємодій всередині та між модулями на основі інтерфейсів, застосовуючи патерни програмування в необхідних місцях, дозволяє значно підвищити толерантність продукту до змін у цільовому функціоналі.

5 РЕАЛІЗАЦІЯ БІЗНЕС-ЛОГІКИ СИСТЕМИ

Будь-яке програмне забезпечення, незалежно від сфери застосування, використаних технологій та досвідченості розробників, напряму залежить від структур даних, що покладені в основу бізнес-логіки. Навіть алгоритми, що можуть неймовірно сильно впливати на швидкість роботи застосунку, вимагають наявності тих чи інших закономірностей в структурах даних для розкриття їх потенціалу.

5.1 Внутрішнє представлення алгоритму програми

Програмування – достатньо тривалий процес. Навіть у випадку закінчених продуктів, що не потребують особливої підтримки та подальшого розвитку (наприклад, однокористувацькі комп'ютерні ігри), розробка такого проекту може тривати роками на наліковувати десятки, а то й сотні тисяч строк вихідного коду. Такі проекти, що отримують неперервну підтримку, відповідно й неперервно розширюються, тому складно встановити навіть приблизний верхній ліміт обсягу програмного проекту.

За таких габаритів репозиторіїв вихідного коду, діаграми, що можуть бути створені на їх базі, можуть налічувати тисячі вузлів та займати гігабайти оперативної пам'яті. Також, рендеринг, переміщення та оновлення діаграм такого розміру може займати дуже велику кількість часу. Тому для економії місця та часу виконання програми необхідно розробити якомога більш компактну структуру даних.

Обсяг опублікованих досліджень на цю тему є дуже невеликим. При публікації статей та тез автори переважно цікавляться зворотною технологією – перекладом складених від руки діаграм у вихідний код [18]. Такий підхід до проектування систем також може бути вигідним, якщо на меті не стоїть створення високопродуктивного результату. В деяких роботах [19, 20] досліджуються застосування діаграм послідовності та діаграм класів, згенерованих на основі вихідного коду.

Одне з опублікованих досліджень [21] фокусується у роботі на генерації з вихідного коду PHP діаграм аналізу програм (program analysis diagram, PAD). Окрім

реалізації самої системи, у процесі викладу суті алгоритму автори виділяють примітивні логічні конструкції, з яких здійнюється побудова всіх комплексних алгоритмів. В основі будь-якого алгоритму, згідно цій роботі, лежать три логічні конструкції. Цими конструкціями є послідовність, що означає простий однозначний перехід між двома кроками алгоритму, умовний перехід, що означає застосування певної умови для розгалуження потоку керування та цикл, що використовується для повторюваного виконання послідовності кроків.

І дійсно, якщо звернути увагу на синтаксис більшості мов програмування, ці елементи легко знайти у вигляді ключових слів або символів, зарезервованих для компілятора [22]. Послідовність блоків виконання визначається переносом строки (Python, FORTRAN) або спецсимволом, таким як крапка з комою (C-подібні мови – C#, Java, JavaScript), та обмежується дужками або за допомогою відступів (індентації). Для умовного переходу більшість мов використовує англійське слово if, а для циклів – слово while. У цикла і умовного переходу автори також виділяють по одній додатковій формі – цикл з перед- або постумовою для цикла та оператор вибору з множини для умовного переходу. Загальний список примітивних логічних елементів алгоритму:

- послідновність (sequence, block)
- умовний перехід (condition, conditional statement)
- цикл з передумовою (while-do loop)
- цикл з постумовою (do-while loop)
- оператор вибору з множини варіантів (case statement, case select)

Структуру наведених елементів зображено на рис. 5.1.

Описана структура даних, що складається з п’яти основних елементів добре підходить для ілюстрації принципів побудови алгоритму, але в неї є ряд важливих недоліків, що не дозволяють використати її у високопродуктивній системі для перетворень.

Основною проблемою, з якої утворюють всі інші, для цієї структури є форма запису її в пам’ять програми. При запису послідовності блоків для побудови блок-схеми необхідно мати якомога меншу варіацію у структурі даних цих блоків.

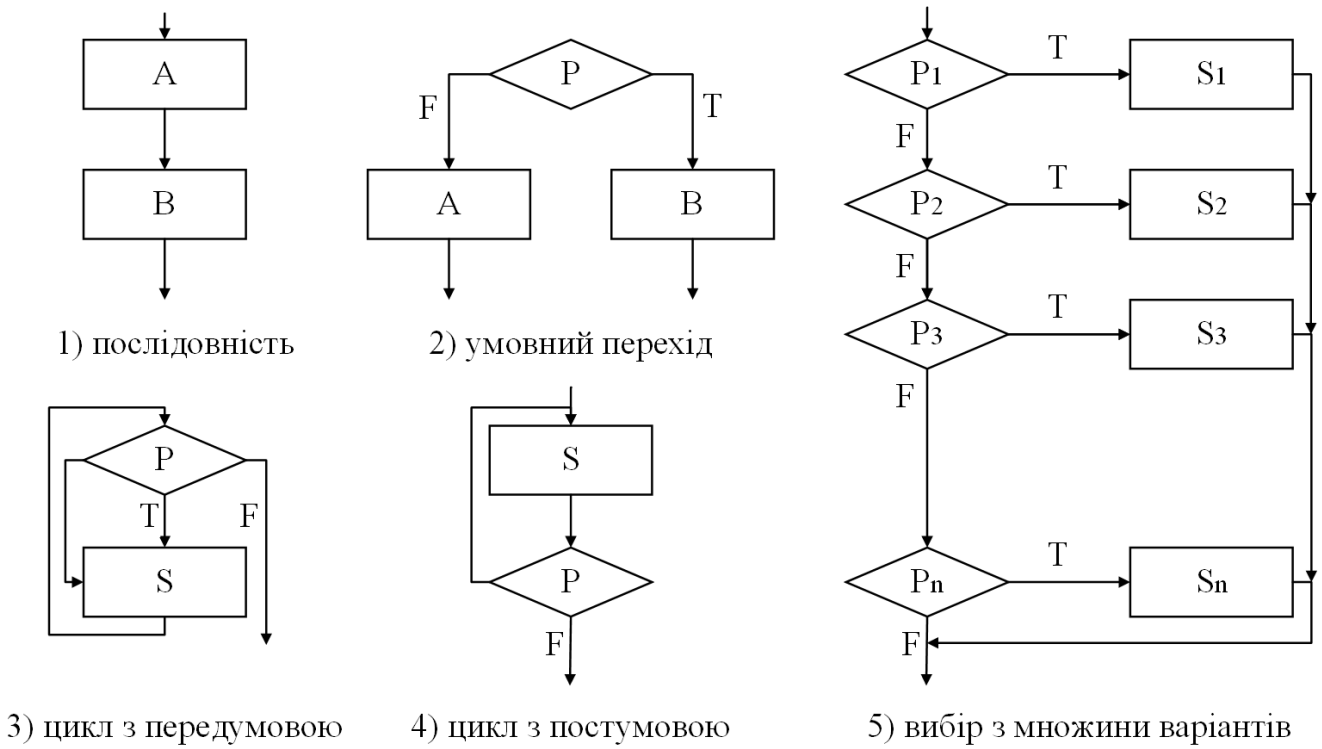

Рисунок 5.1 – Базові структурні складові алгоритму

В описаному переліку з п’яти типів кожен тип володіє уннікальними характеристиками та даними (умова для блоку condition, розміщення послідовностей коду у циклах з перед- та постумовами, перелік умов та відповідних переходів для блоку case select.

Структура даних, що стоїть у основі розроблюваного модуля, позбавляється від цих обмежень шляхом використання одного формату запису для будь-якої одиниці потоку керування алгоритму. Для простоти опису структури даних тут і далі таку логічну одиницю називатимемо вузлом (за аналогією до терміну «вузол графа»).

Вузлом в розроблюваній моделі називається об’єкт, що має асоційований з ним набір метаданих (наприклад, сегмент вихідного коду) та посилання на два наступні вузли, що для полегшення опису називатимемо «істинним» та «хибним» нащадками. Для реалізації механізму посилань присвоїмо кожному вузлу графа діаграми унікальний чисельний ідентифікатор (номер). Таким чином для запису описаної структури даних у базу, масив чи конвертації цієї структури у потік байтів для передачі мережею (серіалізації) достатньо для кожного вузла мати набір з чотирьох полів:

- номер вузла (число);
- метадані (довільний формат);
- номер вузла, що йде наступним при виконанні умови переходу. Тут і далі його названо «істинним» нащадком;
- номер вузла що йде наступним при невиконанні умови переходу. Тут і далі його названо «хибним» нащадком.

Така структура даних, безперечно, займає набагато менше місця ніж структура з п’яти елементів, описаних вище. Проте постає питання вираження цих елементів з набору таких вузлів. Для парсингу конструкцій умовного переходу та послідовності кроків, а також можливості визначення шляху через алгоритм, необхідно накласти на цю структуру даних наступні обмеження:

- обмежимо можливий діапазон номерів вузлів невід’ємними числами для можливості використання беззнакових типів та збільшення кількості допустимих номерів;
- вузол буде вважатися конструкцією послідовності за умови рівності номерів істинного та хибного нащадків або конструкцією умовного переходу в протилежному випадку;
- при відсутності в списку переданих вузлів номера, що вказаний як нащадок вузла, відповідний перехід вважатиметься термінальним (таким, після якого роботу алгоритму буде завершено);
- обов’язковою є передача вузла з номером «0» для позначення першого кроку в алгоритмі;
- посилання на вузол «0» в якості нащадка заборонене, такі переходи також вважатимуться термінальними;
- основний шлях через алгоритм є «хибним шляхом» (проходить тільки через хибні нащадки).

Для виведення трьох інших типів конструкцій необхідно виконати парсинг описаної структури даних. Спочатку визначимо у графі, відображеному такою структурою, цикл. Для цього можна здійснити послідовний обхід графа, визначити місце, де потік керування «зациклюється», тобто момент коли номер вузла

трапляється другий раз за обхід. Останній вузол, чий істинний нащадок містить цей цикл, буде вважатися його умовою. Якщо цей вузол умови дорівнює вузлу «зациклення», це цикл з передумовою, в іншому випадку – з постумовою. Алгоритм визначення циклів з перед- та постумовою наведено на рис. 5.2.

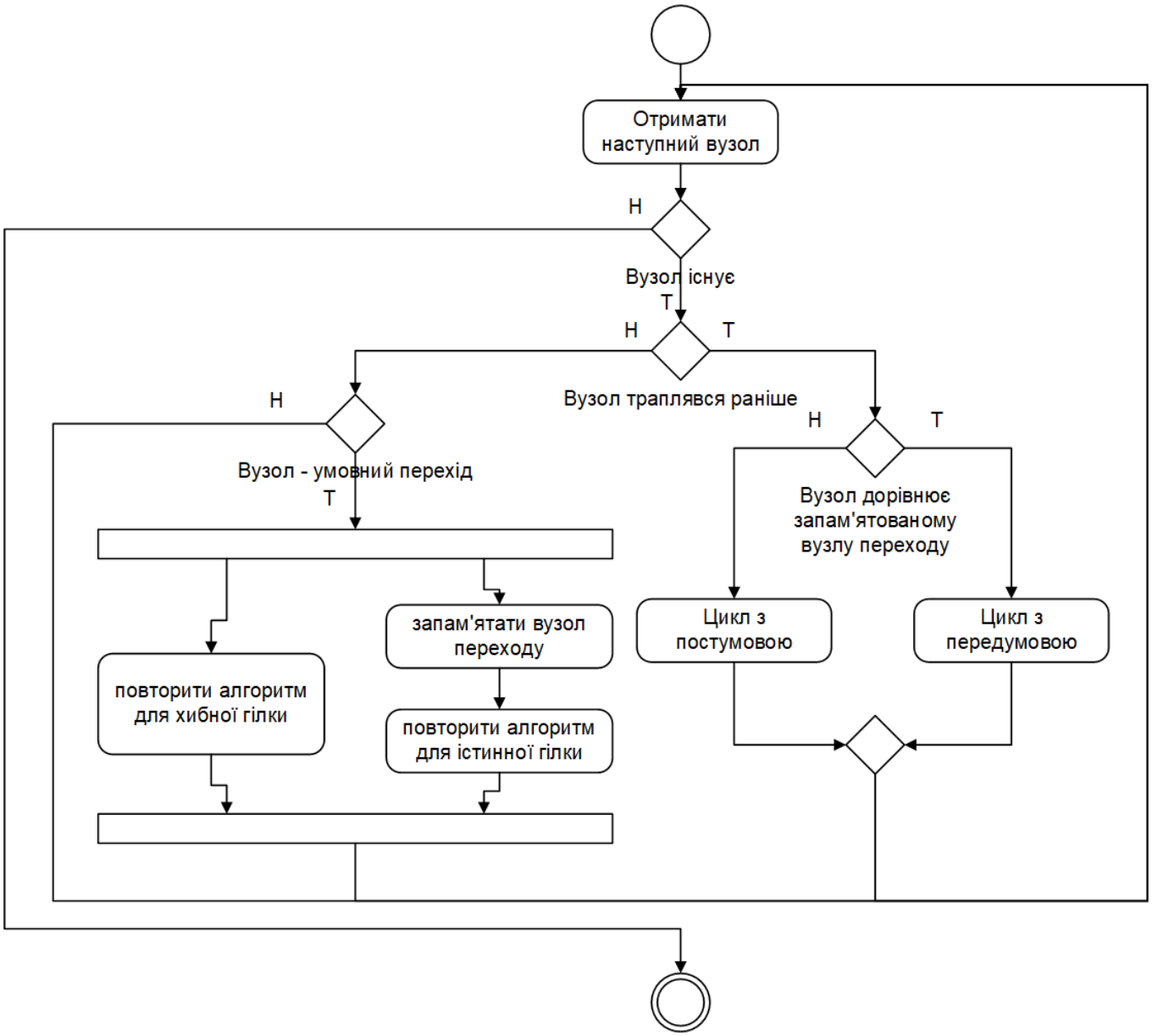

Рисунок 5.2 – Алгоритм визначення циклів з перед- та постумовою

Для визначення множинного вибору достатньо знайти у графі послідовності з кількох умовних переходів, результати яких по «хибних» шляхах сходяться до одного вузла.

Запропонована структура даних є компактною та лаконічною формою запису, що підтримує представлення довільного алгоритму, а також дуже просто може бути перетворення на текстове представлення, використовуючи популярні синтаксиси запису діаграм. Для демонстрації реалізовано переклад такого представлення на мову DOT утиліти GraphViz, описаної в розділі 1. В додатку В наведено лістинг вихідного коду перетворення послідовності вузлів на текстове представлення діаграми (на основі реалізації модуля побудови діаграми) та тестову FORTRAN-програму, в додатку Г– результат перетворення та виведення діаграми утилітою GraphViz.

5.2 Модуль побудови діаграми

Першим з трьох модулів варто описати розробку модуля побудови діаграми, оскільки він визначає програмні інтерфейси, яких повинні дотримуватися обидва інші модулі, обмеження, що накладаються на швидкодію та пропускну здатність програми, а також основні структури даних та методи перетворень, що будуть використані в роботі засобу є частиною цього модуля.

Цей модуль містить у собі програмну реалізацію структури даних, описаної вище, її парсингу, виділення з неї п’яти базових елементів та побудову з них представлення діаграми. Також цей модуль надає функціонал з порівняння двох версій діаграми для визначення та перебудови змінених частин без впливу на збережені вузли.

Реалізація описаної вище структури даних відбувається за допомогою класу Graph. Цей клас відповідає за зберігання, оновлення та отримання інформації про граф потоку виконання.

Клас містить наступні методи:

- get_node(index), що вертає вузол за його індексом, або None, якщо такого вузла немає;
- true_next(node) та false_next(node), що вертають «істинного» та «хибного» нащадка заданого вузла відповідно;

- false_path(node), що вертають перелік всіх вузлів на «хибному шляху», починаючи з заданого.

Ініціалізація класу відбувається за допомогою методу load, що приймає на вхід набір кортежів, кожен з яких містить дані про один вузол. Кожен кортеж перетворюється на об’єкт класу Graph.Node, що являє собою просту структуру даних, призначену лише для зберігання стану вузла. При перетворенні здійснюється перевірка на від’ємні номери, у випадку їх наявності створюється виключення ValueError. Після ініціалізації знаходиться термінальний вузол, що визначений як останній вузол на основному хибному шляху. Вихідний код класу Graph представлено на рис. 5.3.

```
class Graph:
    _nodes: dict[int, 'Graph.Node']
    _end_node: 'Graph.Node'

    def load(self, source):
        self._end_node = None
        self._nodes = {}
        for index, false_id, true_id, text in source:
            if index < 0:
                raise ValueError
            self._nodes[index] = Graph.Node(index, false_id, true_id, text)
        *_, self._end_node = self.false_path(self.get_start_node())

    def get_nodes(self):
        yield from self._nodes.values()

    def get_start_node(self):
        return self._nodes.get(0, self._end_node)

    def _get_node(self, index: int):
        return self._nodes.get(index, self._end_node) if index else self._end_node

    def get_false_next(self, node: 'Graph.Node'):
        return None if node is self._end_node else self._get_node(node.false_next)

    def get_true_next(self, node: 'Graph.Node'):
        return None if node is self._end_node else self._get_node(node.true_next)

    def false_path(self, node: 'Graph.Node'):
        while node:
            yield node
            node = self.get_false_next(node)
```

Рисунок 5.3 – Вихідний код класу Graph

За процес парсингу базових структур та основу взаєморозміщення блоків діаграми відповідає клас Plotter. Основу цього класу складає метод plot, що приймає на вхід частину графа, яку необхідно «розмістити» та вертає список блоків діаграми, згенеровану на основі вхідного набору вузлів. Оскільки структура даних містить два типи переиходів, необхідно розробити механізм відображення для кожного з них.

Для переходу типу «послідовність» механізм відображення дуже простий та прямолінійний – два вузли, що складають цей перехід можна відобразити вертикально один під одним. Для цього необхідно створити об’єкт класу BlockStack, що об’єднуватиме в собі послідовність з блоків, що відображається в одну лінію.

Для переходу типу «умова» механізм відображення дещо складніший. Оскільки такий перехід має двох нащадків, для початку необхідно визначити, чи «хибні шляхи» гілок обидвох нащадків сходяться в одному вузлі. Якщо такий вузол знайдено, довжина гілок обмежується цим вузлом. Далі метод plot викликається рекрсивно для кожної гілки нащадка, послідовності блоків, отримані в результаті, можна об’єднати в два BlockStack-и. Розміщення двох таких гілок логічно організувати горизонтально одна поруч з іншою. Для цього необхідно модифікувати клас BlockStack, щоб він дозволяв вибирати між горизонтальним та вертикальним розміщенням, та помістити обидві гілки в нього.

Дані про блок діаграми описані в абстрактному класі BlockBase. Для відображення структур на екрані використовується рекурсивна структура «дерева» елементів, що складається з трьох операцій: «виміряти» (measure), «розмістити» (arrange), «відобразити» (display). Ідея використання такої структури полягає в тому, що абстракцію роботи алгоритму розміщення блоків можна відділити від реалізації описаних вище операцій. Така структура є реалізацією патерну «компонувальник», описаного вище. Користувач такого фреймворка може успадкувати клас BlockBase для реалізації цих процедур. Клас DiagramBlock, що реалізує ці операції для роботи розроблюваного засобу, описано у розробці модуля відображення.

Ці ж операції необхідно реалізувати і для класу BlockStack, описаного вище.

Операція measure має обчислити та повернути розмір у пікселях, який цей блок хоче займати на екрані. Для вертикального стека висота блоку буде дорівнювати сумі

висот дочірніх блоків, а ширина – максимуму з їх ширин. Для горизонтального стека ці виміри необхіднно відзеркалити. Визначення розмірів дочірніх блоків відбувається за допомогою того ж методу measure, викликаного для всіх блоків послідовно.

Операція arrange приймає на вхід позицію на екрані у пікселях та має обчислити координати, які буде займати її початкова точка (для pygame такою точкою є верхній лівий кут блока). BlockStack для розміщення дочірніх блоків визначає центр (горизонтальний або вертикальний в залежності від орієнтації стека) та розміщує блоки один біля одного вздовж лінії цього центра. Для цього викликається відповідний метод arrange для дочірніх блоків, куди передаються розраховані координати дочірнього блока.

Операція display просто викликає відповідний метод для всіх дочірніх блоків. Реалізація такого методу залежатиме від системи, що використовується для виведення діаграми.

Вихідний код класу BlockStack наведений на рис. 5.4.

З опису цих операцій видно, що при побудові діаграми буде здійснено три послідовних обходи дерервовидної структури, в яку організовано блоки. Після завершення кожного з цих обходів дерево стає готовим для проведення наступного, а останній з них – відображення – буде викликатися кожного разу, коли необхідно оновити зображення на екрані.

Спрощену діаграму класів модуля наведено в додатку Д.

5.3 Модуль керування вихідним кодом вільної форми

Цей модуль реалізовано як плагін до середовища розробки Photran. Він містить у собі операції що здійснюють перетворення вихідного коду мовою FORTRAN free form на структуру даних з вузлів, що описана вище. Для парсингу вихідного коду та перетворення його на дерево з синтаксичних структур FOTRTAN застосовуються засоби, що надані Photran.

Структури даних, що Photran використовує для представлення вихідного коду та внесення в нього змін, називаються абстрактним синтаксичним деревом (abstract

```
class BlockStack(BlockBase):
    children: list[BlockBase]

    def __init__(self, children, orientation: Orientation = Orientation.VERTICAL):
        BlockBase.__init__(self)
        self.flip = orientation == Orientation.HORIZONTAL
        self.children = list(children)

    def _measure(self):
        for c in self.children:
            c.measure()

        if self.children:
            measure_max = (v.size.x for v in self.children)
            measure_sum = (v.size.y for v in self.children)
            if self.flip:
                (measure_max, measure_sum) = measure_sum, measure_max
            width = max(measure_max)
            height = sum(measure_sum)
        else:
            width = height = 0
        if self.flip:
            (width, height) = height, width
        return Vectora(width, height)

    def _arrange(self, pos: Vectora):
        pos -= self.margin.y_comp() if self.flip else self.margin.x_comp()

        for c in self.children:
            # center in parent
            offset = (self.size - c.size) / 2
            offset = offset.y_comp() if self.flip else offset.x_comp()
            c.arrange(pos + offset)

            # advance position to next child
            pos += c.size.x_comp() if self.flip else c.size.y_comp()

    def _draw(self, camera):
        for c in self.children:
            c.draw(camera)
```

Рисунок 5.4 – Вихідний код класу BlockStack

syntax tree, AST) та віртуальним графом програми (virtual program graph, VPG). Структура AST – це набір синтаксичних одиниць мови програмування, організований у дерево. Гілками цього дерева є композитні мовні конструкції, а листками – оператори, літерали, ідентифікатори, що визначені в тексті програми. Наприклад, конструкція if представлена у графі вузлом, значенням якого є об’єкт класу ASTIfStmtNode, конструкція циклу DO-END DO представлена об’єктом ASTProperLoopConstructNode тощо. Приклад структури AST для одної з демонстраційних програм, використаних для тестування алгоритму (див. додаток В), наведено в додатку Е. Структура VPG являє собою програмний інтерфейс, за

допомогою якого програміст може отримати доступ до дерева AST певного файлу програми, здійснювати навігацію кодом, перевірку відповідності посилань на ідентифікатори, вивільняти ресурси парсера після закінчення роботи.

Обхід синтаксичного дерева для перевторення вихідного коду на послідовність вузлів графа потоку виконання здійснено за допомогою патерна «відвідувач», описаного раніше. Для цього у програмному інтерфейсі Photran існує інтерфейс IASTVisitor, що необхідно реалізувати. Цей інтерфейс описує набір методів, в тілі яких має здійснюватися обробка «відвіданих» синтаксичних конструкцій. Реалізація кожного структурного вузла містить метод accept(IASTVisitor), що слугує реалізацією патерну та надає відвідувачу доступ до внутрішніх даних конструкції та переліку дочірніх вузлів. Згаданий вище клас «ASTIfConstructNode» в своїй реалізації методу accept дозволяє об’єкту відвідувача обійти тіло конструкції (внутрішній код у гілці умовного переходу) та конструкції альтернативних умовних переходів (if-else та else).

Тривіальним прикладом структур дерева та відвідувача буде програма, що містить визначення певних даних (змінної, масиву тощо.) та виклик підпрограми. Підпрограма, в свою чергу, міститиме конструкцію умовного переходу (if-elseif). Клас-відвідувач для обходу такого дерева повинен реалізувати ряд методів, по одному для кожного типу структури.

Парсинг абстрактного синтаксичного дерева програми здійснюється за допомогою класу ASTConverter, що реалізує описаний вище інтерфейс. При парсингу дерева важливою структурою даних буде логічний блок коду, поведінку якого реалізовано за допомогою класу Scope. Цей клас описує послідовність вузлів, що обмежені певним стартовим та кінцевим вузлами. Для збереження списку вузлів використано стандартний клас ArrayList.

Кожен вузол на діаграмі представлений класом Node, що містить у собі ті ж поля, що і аналогічний клас у модулі побудови діаграми. Додатково цей клас містить методи setNext(int), що встановлює обидва нащадки в одне значення, для скорочення запису створення послідовності блоків, та метод append(String), що додає нову строчку до текстових метаданих вузла. Це потрібно для групування кількох послідовних виразів мови програмування в один вузол.

Для додавання програмних структур до об’єкту класу Scope реалізовано два методи – addNode(IASTNode) та appendNode(IASTNode). Перший метод використовується тоді, коли додана конструкція має стати самостійним вузлом у графі. Цей метод також пов’язує доданий вузол з попереднім у переліку. Другий метод використовується коли необхідно додати текст до вже створеного вузла (при групуванні блоків). В такому випадку він просто додає текст структурного елемента до останнього доступного вузла. Під час додавання вузлів клас Scope слідкує за збереженням унікальності їх номерів за допомогою внутрішньої змінної класу IdGenerator, що інкрементує номер вузла під час кожного додавання.

Оскільки об’єкти класу Scope використовуються для зберігання інформації як про структурні елементи, що будуть перетворені на самостійні діаграми (програми, підпрограми, функції), так і для менших елементів (як-от тіла циклів, умовних переходів), то важливою частиною функціоналу буде об’єднання менших блоків коду в більші, що може відбуватися рекурсивно багато разів залежно від рівня вкладеності програмного коду в певній точці.

Запис блоків виконується за допомогою такого об’єкту Scope, що доступний всім методам класу ASTConverter. Проте цей об’єкт має міняти свою поведінку в залежності від того, який рівень вкладеності обробляється на даний момент. Для цього клас Scope містить два додактові методи. Метод open(IASTNode, IASTNode, Link) призначений для того щоб створити блок коду, вкладений в поточний. При цьому новий блок коду отримує той же IdGenerator, що й поточний блок. Це гарантує відсутність колізій номерів вузлів.

Другий метод close() «закриває» поточний блок та записує всі вузли в батьківський блок між двома вузлами, переданими при виклику методу open(), використовуючи задане відношення між блоками. Link – це об’єкт типу «перечислення», що може приймати три значення – FALSE, що позначає хибний нащадок, TRUE що позначає істинний та NEXT, що означає просту послідовність. Відповідно до цього значення, перший елемент вкладеного блоку записується в істинний, хибний або обидва шляхи стартового вузла. Вихідний код класу Scope наведено на рис. 5.5.

```
public class Scope
{
    private final Scope superScope;
    private final List<Node> graph = new ArrayList<Node>();
    private final Node startNode;
    private final Node endNode;
    private Node last;
    private final EnumSet<Link> link;
    private final IdGenerator generator;

    public Scope open(Node startNode, Node endNode, EnumSet<Link> link)
    {
        if (startNode != endNode) startNode.defaultTo(endNode.id);
        return new Scope(generator, this, startNode, endNode, link);
    }

    public Scope close()
    {
        if (superScope != null)
        {
            if (last != null)
            {
                last.defaultTo(endNode.id);
                Node firstNode = graph.get(0);
                if (link.contains(Link.T)) startNode.true_next = firstNode.id;
                if (link.contains(Link.F)) startNode.false_next = firstNode.id;
            }
            superScope.graph.addAll(graph);
        }
        return superScope;
    }

    public Node addNode(IASTNode node)
    {
        if (last == null || last.hasText()) newNode();
        if (node != null) appendInternal(node, true);
        return last;
    }

    public void append(IASTNode node) { appendInternal(node, false); }

    private void appendInternal(IASTNode node, boolean close)
    {
        if (last == null || !last.isWritable()) newNode();
        last.append(ScopeUtil.nodeToString(node), close);
    }

    private void newNode()
    {
        Node result = new Node(generator.GetId());
        if (last != null) last.defaultTo(result.id);
        graph.add(result);
        last = result;
    }
}
```

Рисунок 5.5 – Вихідний код класу Scope

Різні програмні структури потребують різних дій при їх виявленні в програмному коді. З них можна виділити чотири групи конструкцій, дії при виявленні яких суттєво відрізняються між групами, але подібні між собою всередині групи. Такими групами є:

- підпрограми;
- умовні переходи;
- цикли;
- твердження.

Підпрограми є групою, що відповідає за оператори subroutine, program та function мови FORTRAN та у AST представлена трьома видами вузлів, що наведені в таб. 5.1.

Таблиця 5.1 – Мовні конструкції групи підпрограм

Назва вузла	Опис мовної конструкції	Відповідний клас AST
Програма	Позначає об’єкт, що містить головну програму модуля, що викликається при компіляції цього модуля та виконанні отриманого файлу.	ASTMainProgramNode
Процедура	Позначає набір програмних інструкцій, що об’єднані в певну операцію. Головним призначенням є зміна переданих даних.	ASTSubroutineSubprogramNode
Функція	Позначає набір програмних інструкцій, що об’єднані у аналог математичної функції. Основним призначенням є розрахунок та повернення результату.	ASTFunctionSubprogramNode

Ці структури після виконання перетворені будуть передані в модуль побудови як окремі алгоритми. Тобто для кожної підпрограми буде створена своя блок-схема. При виявленні такої структури необхідно створити новий об’єкт Scope, записати в нього заголовок підпрограми, всі вузли, прочитані з тіла підпрограми та індикатор кінця підпрограми. Вихідний код для парсингу підпрограм наведено на рис. 5.6.

Умовні переходи – група, що відповідає за оператори умовного переходу IF, ELSEIF та ELSE та представлена у AST чотирма видами вузлів, що наведені у таб. 5.2.

```
private void collectScope(ScopingNode scope)
{
    IASTNode body = scope.getBody();
    if (body != null) body.accept(this);
}

private void collectFunction(ScopingNode scope, IASTNode start, IASTNode end)
{
    target = new Scope();
    Node startNode = target.addNode(start);
    Node endNode = target.addNode(end);
    target = target.open(startNode, endNode, Link.NEXT);
    collectScope(scope); // $NON-NLS-1$
    target = target.close();
    result.add(target);
    target = null;
}

@Override
public void visitASTMainProgramNode(ASTMainProgramNode node)
{
    collectFunction(node, node.getProgramStmt(), node.getEndProgramStmt());

    IASTNode subPrograms = node.getInternalSubprograms();
    if (subPrograms != null) subPrograms.accept(this);
}

@Override
public void visitASTSubroutineSubprogramNode(ASTSubroutineSubprogramNode node)
{
    collectFunction(node, node.getSubroutineStmt(), node.getEndSubroutineStmt());
}

@Override
public void visitASTFunctionSubprogramNode(ASTFunctionSubprogramNode node)
{
    collectFunction(node, node.getFunctionStmt(), node.getEndFunctionStmt());
}
```

Рисунок 5.6 – Вихідний код для парсингу підпрограм

Таблиця 5.2 – Мовні конструкції групи умовних переходів

Назва вузла	Опис мовної конструкції	Відповідний клас AST
Твердження умовного переходу	Позначає однорядковий умовний перехід	ASTIf StmtNode
Конструкція умовного переходу	Позначає багаторядковий умовний перехід.	ASTIf ConstructNode
Конструкція альтернативного переходу	Позначає альтернативний шлях, що виконується після перевірки умови.	ASTElse ConstructNode
Конструкція альтернативної умови	Позначає додаткову умову, що перевіряється та виконується перед виконанням альтернативного переходу.	ASTElseIf ConstructNode

Для парсингу умовних переходів застосовано опцію Link методу Scope.open(). За її допомогою можливо відкрити «підблок» який запишеться у вказану гілку. Різні умовні переходи вимагають різних дій для їх обробітку.

Твердження умовного переходу не може містити альтернативних шляхів та завжди містить один вираз присвоєння у тілі. Тому для його обробітку потрібно додати вузол заголовку умовного переходу та пустий вузол кінця умовного переходу. Після цього потрібно відкрити підблок Scope на хибній гілці між цими двома вузлами, записати в нього вираз присвоєння, та закрити його.

Конструкція умовного переходу обробляється значно складніше. Блок ASTIfConstructNode може містити блоки, що відповідають за переходи ELSEIF та ELSE. Блоки elseif також можуть містити інші блоки elseif та else. Всі три типи блоків – IF, ELSEIF та ELSE можуть містити блок END IF. В такому випадку зручно скористатися тим, що піблоки Scope можна відкривати рекурсивно. Для конструкції if так само як для твердження необхідно відкрити істинну гілку блоку, але подальші дії суттєво відрізняються:

1. створити вузол заголовку умовного переходу та пустий вузол кінця умовного переходу;
2. відкрити істинну гілку цього блоку;
3. записати тіло умовного переход;
4. закрити істинну гілк;
5. перевірити чи блок містить конструкції ELSEIF або ELSE;
6. якщо так, відкрити хибну гілку для поточного блоку;
7. для конструкції ELSEIF рекурсивно запустити алгоритм з кроку 1;
8. для конструкції ELSE записати у хибну гілку тіло альтернативного шляху;
9. якщо на кроці 6 було відкрито хибну гілку, закрити її;
10. Вернути на попередній рівень рекурсії поточну конструкцію END IF;
11. Якщо конструкції не знайдено, спробувати вернути конструкцію вкладених рівнів;
12. Після завершення цього алгоритму на найвищому рівні, отриману в результаті конструкцію end if записати в кінець блоку.

За допомогою цього рекурсивного алгоритму можна обробити будь-яку структуру з конструкцій умовного переходу з довільним рівнем вкладеності та кількістю альтернативних шляхів. Вихідний код операцій парсингу умовних переходів наведено на рис. 5.7.

```
private ASTEndIfStmtNode convertIfElse(Node start, Node end, IASTNode body,
    ASTElseIfConstructNode elifNode, ASTElseConstructNode elseNode, ASTEndIfStmtNode endNode)
{
    if (body != null) body.accept(this);
    target = target.close();

    ASTEndIfStmtNode result = null;
    boolean falseBranch = elifNode != null || elseNode != null;
    if (falseBranch)
        // false branch for enclosing scope
        target = target.open(start, end, Link.FALSE);

    if (elifNode != null)
    {
        start = target.addNode(elifNode.getElseIfStmt());

        // true branch for else if, closed internally
        target = target.open(start, end, Link.TRUE);
        result = convertIfElse(start, end, elifNode.getConditionalBody(),
            elifNode.getElseIfConstruct(), elifNode.getElseConstruct(),
            elifNode.getEndIfStmt());
    }

    if (elseNode != null)
    {
        body = elseNode.getConditionalBody();
        if (body != null) body.accept(this);
        result = elseNode.getEndIfStmt();
    }

    if (falseBranch)
        // closing false branch
        target = target.close();

    if (result == null) result = endNode;
    return result;
}

@Override
public void visitASTIfConstructNode(ASTIfConstructNode node)
{
    Node start = target.addNode(node.getIfThenStmt());
    Node end = target.addNode(null);
    // open true branch
    target = target.open(start, end, Link.TRUE);

    IASTNode result = convertIfElse(start, end, node.getConditionalBody(),
        node.getElseIfConstruct(), node.getElseConstruct(), node.getEndIfStmt());

    if (result != null) target.addNode(result);
}
```

Рисунок 5.7 – Вихідний код для парсингу умовних переходів

Цикли – група, що описує цикли DO, DO WHILE у їх варіантах з мітками та з виразом кінця циклу. Їх всі у AST представлено одним типом вузла – ASTProperLoopConstructNode, їх значення наведено в таб. 5.3.

Для парсингу циклів використовується той же механізм підблоків, що й для парсингу умовних переходів. Для цього створюється вузол з заголовком цикла та істинна гілка підблоку відкривається використовуючи цей вузол як початок та кінець одночасно. Тіло циклу записується в цей підблок, після чого він закривається. До поточного блоку дописується вузол з твердженням закриття циклу.

Таблиця 5.3 – Мовні конструкції групи циклів

Назва вузла	Опис мовної конструкції	Приклад синтаксису
Цикл DO з міткою	Позначає вираз циклу з ітератором, кінець якого позначений міткою.	DO 100 i=1,10 … 100 CONTINUE
Цикл DO з виразом кінця циклу	Позначає вираз циклу з ітератором, кінець якого позначений твердженням.	DO i=1,10 … END DO
Цикл DO WHILE з міткою	Позначає вираз циклу з передумовою, кінець якого позначений міткою.	DO 100 WHILE (I.NE.0) … 100 CONTINUE
Цикл DO WHILE з виразом кінця циклу	Позначає вираз циклу з передумовою, кінець якого позначений твердженням.	DO WHILE (I.NE.0) … END DO

Всі групи синтаксичних конструкцій, описані вище, що можуть містити тіло з переліком дочірніх виразів у фреймворку Photran AST наслідують клас ScopingNode. Це дає змогу здійснювати парсинг таких вузлів однаковим чином, оскільки кожен з них потребує перевірки на пустоту цього списку для уникнення виключних ситуацій. Вихідний код операції парсингу циклів та блоків з тілом наведено на рис. 5.8.

Більшість інших тверджень, як-от твердження визначення структур даних, твердження присвоєння, виведення на екран, пусті інструкції CONTINUE тощо не потребують особливого ставлення та можуть бути записані в останній відкритий блок. Блок вважається відкритим, якщо є пустим або містить подібні твердження. Для цього використовується метод Scope.append().

```
private void collectScope(ScopingNode scope)
{
    IASTNode body = scope.getBody();
    if (body != null) body.accept(this);
}

@Override
public void visitASTProperLoopConstructNode(ASTProperLoopConstructNode node)
{
    Node loopNode = target.addNode(node.getLoopHeader());
    target = target.open(loopNode, loopNode, Link.TRUE);
    collectScope(node);
    target = target.close();
    target.addNode(node.getEndDoStmt());
}
```

Рисунок 5.8 – Вихідний код парсингу циклів та блоків

Виключення для цього становлять рядки вихідного коду що містять мітки. Оскільки мітки можуть бути використані конструкціями DO, WRITE та GOTO для модифікації їх поведінки, то ці блоки необхідно виділяти окремо та записувати звичайним методом Scope.add().

Спрощену діаграму класів модуля наведено в додатку Ж.

5.4 Модуль відображення

Призначенням цього модуля є відображення діаграми та взаємодія з користувачем. Python містить багато можливих шляхів створення користувацького інтерфейсу. Більшість з них орієнтовані на створення статичних вікон з низькою частотою зміни інформації на вікні. Це прекрасно підходить для створення віконних застосунків по роботі з списками, текстом, базами даних тощо. Прикладом такого фреймворку є Tkinter.

Tkinter - це найбільш часто використовувана бібліотека для розробки графічного інтерфейсу користувача (GUI) в Python. Це стандартний інтерфейс Python до набору інструментів Tk GUI, що постачаються разом з Python. Оскільки Tk і Tkinter

доступні на більшості платформ Unix, а також у системі Windows, розробка графічних додатків за допомогою Tkinter є одним з найшвидших та найпростіших способів.

Усі віджети Tkinter мають доступ до певних методів управління геометрією, які використовуються для організації віджетів у візуальній області батьківського елемента. Tkinter пропонує наступні методи менеджерів геометрії: pack(), grid() та place().

Метод pack() - цей менеджер геометрії організовує віджети по блоках, перш ніж розміщувати їх у батьківському віджеті.

Метод grid() - цей менеджер геометрії організовує віджети у структурі, подібній до таблиці, у батьківському віджеті.

Метод place() - цей менеджер геометрії організовує віджети, розміщуючи їх у певному положенні батьківського віджета.

З цих трьох менеджерів геометрії найбільш підходящим для створення інтерфейсу модуля відображення є place(). Але хоч він і дає можливість розміщувати елементи в довільних місцях на екрані, керування розміром вікна та положенням елементів саме у цьому вікні є проблематичним. А додаткові функції, як-от приближення та віддалення діаграм, переміщення їх екранном можуть зайняти велику кількість часу на розробку та виявитися недостатньо швидкими.

Альтернативою для менеджерів віконних інтерфейсів є бібліотеки для розробки комп’ютерних ігор. У зв’язку з специфікою цього напряму, такі бібліотеки часто надають користувачу додаткові та розширені можливості керування зображенням у вікні. Серед необхідних для модуля відображення можна виділити:

- можливість ручного оновлення зображення на екрані;
- прямолінійний доступ до розміру вікна;
- можливість розміщувати зображення поза межами вікна;
- можливість змінювати масштаб зображень;
- простий програмний інтерфейс для малювання графічних примітивів;
- прямий доступ до програмних переривань (натисків клавіш, роботи з мишею).

Найпопулярнішою бібліотекою, що містить всі вищенаведені функції, є pygame.

Pygame це набір модулів python призначених для написання комп'ютерних ігор. Він слугує обгорткою потужної C-бібліотеки SDL. Це означає що pygame дозволяє програмісту створювати повнофункціональні ігри та мультимедійні програми мовою python. Перевагами pygame для створення високопродуктивних інтерактивних систем візуального інтерфейсу є:

- надзвичайна крос-платформеність – бібліотека працює майже на всіх операційних системах та платформах;
- висока популярність, що означає наявність широкого переліку навчальних матеріалів та розвивену спільноту;
- бібліотека постачається за ліцензією LGPL, що дає можливість розробляти як open-source, так і комерційні продукти;
- нативна та проста підтримка багатоядерних процесорів;
- використовує високопродуктивні функції написані на C та assembler. Це дає приріст продуктивності у 10-100 раз більше за код на пітоні;
- pygame не є фреймворком, він надає повний контроль над ігровим циклом.
- лаконічність написаного коду. Pygame не вимагає писати сотні рядків для реалізації простої поведінки.
- модульність. Підмодулі pygame можна ініціалізувати та використовувати окремо, програміст не потрапляє в залежність від вибраних бібліотек.

Реалізацію основного функціоналу системи необхідно почати зі створення вікна, керування основним циклом програми, та контролю виводу інформації на екран. Перед початком роботи з бібліотекою необхідно здійснити її ініціалізацію. Для цього викликається функція pygame.init(). Ця функція є обгорткою над функцією бібліотеки SDL SDL_Init() та виконує налаштування графічної системи, реєстрацію ресурсів та іншу роботу з ініціалізації.

Керування вікном застосунку у модулі відображення інкапсульовано у класі Window. Цей клас призначений для всіх основних функцій вікна – відображення вмісту графічного буфера, оновлення зображення, контролю частоти кадрів (fps).

При ініціалізації вікна в конструктор передається розмір по ширині та висоті, який необхідно встановити. Саме вікно відкривається за допомогою методу show().

Метод show виконує кілька операцій з ініціалізації, першою з яких є створення вікна застосунку. Для створення та керування вікном у pygame існує пакет display. Він керує єдиним доступним екземпляром вікна. Для отримання об’єкту Surface (поверхня), яким представлений екран вікна, необхідно викликати метод set_mode(size). Аргументом size є кортеж з ширини та висоти, яку необхідно встановити. Потім метод show() встановлює назву вікна за допомогою методу set_caption(str), ініціює внутрішній таймер головного циклу, що представлений класом Clock та встановлює флаг що контролює повторення основного циклу у положення true. Після цього у циклі, що триває весь час роботи застосунку, здійснюється основне тіло обробки даних.

Головний цикл застосунку починається з обробки вводу користувача. Процес обробки вводу полягає в переборі списку з об'єктів класу Event. Для обробки кожного з цих об’єктів у класі Window створено об’єкт словника (dict), що співставляє кожен тип події користувацького вводу (натиски кнопок, рухи мишею тощо) з переліком функцій, що повинні отримувати цю подію. Така функціональність описана патерном «спостерігач», який дозволяє зовнішнім класам «підписуватися» на події. Клас Window при отриманні події викликає всі функції «підписників», що зареєстровані в структурі даних, тим самим повідомляючи їх про надходження події.

Після обробки вводу екран очищається (заповнюється білим фоновим кольором), та викликається метод відображення, куди передається об’єкт екрану. Метод відображення є публічною змінною, яку можуть змінювати всі зовнішні класи. Значення цієї змінної можна встановити ззовні для того щоб вікно на кожному кадрі виконувало виведення потрібної користувачу класу інформації. Така взаємодія з вікном є реалізацією патерну «делегування», описаного в підрозділі 4.4. Це дає можливість відділити поведінку вікна від поведінки відображення інформації (наприклад, малювання діаграм).

Після виклику делегату малювання, необхідно забезпечити коректну частоту кадрів застосунку, для чого викликається метод внутрішнього таймера tick(fps). Цей

метод переконується, що на виконання одного проходу циклу витрачено щонайменше 1/fps секунд шляхом додавання додаткової затримки потрібного розміру. Після виконання всіх цих операцій, дисплей оновлюється за допомогою методу flip(), та цикл починається спочатку.

Для демонстрації прогресу роботи над алгоритмом виведення діаграми використано коротку програму:

```
program x
integer :: n = 0
if (n < 0) then
  n = 0
elseif (n>10)
  n = 10
else
  n = -n
end if
end program x
```

Для відображення діаграми необхідно використати структуру класів, описану у підрозділі 5.2. Для цього класу Plotter необхідно передати «алгоритм» створення вузлів, що будуть коректно визначати своє положення та розмір на екрані. Така поведінка підпадає під патерн «фабричний метод» та дозволяє реалізувати стратегії взаємодії об’єктів незалежно від реалізації їх створення. Таким чином можна перевизначити клас BlockBase, створивши новий клас DiagramBlock, що буде виконувати операції з вимірювання та відображення блоку діаграми на екрані. У випадку pygame для цього використовуються об’єкти Surface (такі ж як для представлення екрану вікна). При створенні блоку необхідно передати в нього відповідний вузол графа потоку виконання. При ініціалізації, блок здійснює рендеринг тексту та додаткових елементів (рамки, стрілок) на об’єкт поверхні, та записує цю поверхню в приватну змінну. Операція вимірювання блоку тоді полягатиме в отриманні розмірів цієї поверхні за допомогою методу getSize(). Відображення об’єкта буде виконуватися знову ж таки за допомогою патерна

делегації, де кожному вузлу передається функція camera, що представляє собою абстрактну операцію малювання. Вузол може викликати цю функцію, передавши в неї об’єкт Surface та координати, а реалізація цієї функції забезпечить відображення тексту та іншої графіки на екрані за допомогою методу Surface.blit(Surface, size, position). Цей метод реалізовано у класі вікна, і таким чином патерн «делегація» застосовується рекурсивно. Метод Window.draw_surface() передається у метод Plotter.display(), що сам був переданий у клас Window. Таким чином реалізації класів Window та Plotter нічим не залежать один від одного та можуть бути вільно змінені. Результат розміщення набору вузлів наведено на рис. 5.9. Кожен вузол замінено прямокутником для демонстрації результатів роботи алгоритму на цьому етапі. Навколо відображень об'єктів BlockStack додано рамки для наглядності.

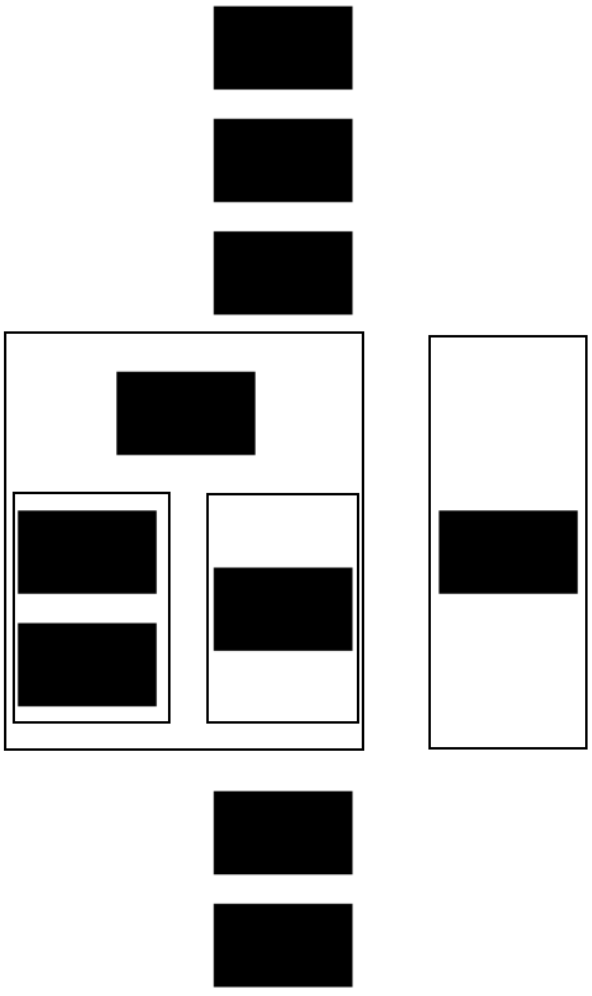

Рисунок 5.9 – результат розміщення блоків отриманих з тестової програми

Якщо з виведенням діаграми на екран завдяки правильно підібраній архітектурі все виявляється просто, то реалізація переміщення діаграми по екрану та наближення/віддалення «сцени» вимагає видіозміни методу draw_surface.

Для реалізації переміщення діаграми екраном додамо до всіх вхідних координат певний зсув. Для реалізації роботи з координатами у модуль побудови діаграм було додатково включено клас Vector.

Клас вектор містить декартові координати у вигляді точок з плаваючою комою, та підтримує основні операції над векторами – додавання, віднімання, множення на число, отримання x- та y-компонентних векторів. Також для можливості передачі об’єктів Vector методам, що приймають на вхід списки чи кортежі з координат реалізовано доступ до елементів за індексом та ітератор. Весь цей функціонал вимагає перевантаження операторів python та дозволяє спрощену схему маніпуляції даними.

При отриманні координат виведення певного об’єкту Surface до них додається координати зсуву, обидва значення класу Vector. Маніпуляція зсувом виконується за допомогою миші. При натисканні лівої клавіші миші перемикається флаг, та об’єкт Window починає слідкувати за переміщеннями миші, оновлюючи розмір зсуву в залежності від того, наскільки та в яку сторону переміщено мишу. При відпусканні лівої клавіші рухи миші перестають відслідковуватися.

Для реалізації зуму потрібно слідкувати за коліщатком миші. Pygame сприймає прокручування коліщатка вгору як клавішу миші 4, а вниз – клавішу 5. Це прокручування збільшує або зменшує змінну zoom_factor, яка відповідає за коефіцієнт збільшення сцени.

При зміні цього коефіцієнта також необхідно врахувати поточне положення миші (zoom_center), щоб збільшення відбувалось «навколо» нього. Для цього при кожному збільшенні перераховується значення зсуву за наступним алгоритмом:

1. вираховується поточний зсув відносно положення миші (result=offset-zoom_center)
2. результат ділиться на поточне значення zoom_factor;
3. результат кроку 2 множиться на нове значення zoom_factor;
4. результат кроку 3 додається до положення миші (offset=zoom_center+result).

Таким чином зсув відносно положення миші лишається незмінним (з урахуванням коефіцієнту наближення). Для застосування наближення необхідно передані у метод відображення координати та розміри зображень домножити на значення zoom_factor.

Для відображення тексту та рамок навколо блоків до фабричного методу блоків необхідно модифікувати табличний метод. Рендеринг тексту в pygame виконується за допомогою методу render класу Font. В результаті отримується об’єкт класу Surface який потім можна скопіювати на інші подібні об’єкти. В залежності від виду вузла, з якого створюється блок, малюватимуться наступні фігури:

Вузол-послідовність – текст вписується в прямокутних з відповідним співвідношенням сторін та певним проміжком між текстом та рамкою. Для цього використовується функція pygame.draw.lines() що приймає поверхнню на якій необхідно малювати, колір лінії а також перелік координат точок, через які необхідно провести лінію. В цьому випадку такими точками будуть кути з координатами (0,0), (0,1), (1,1), (1,0). Координати задані відносно розмірів поверхні, тому перед передачею їх в метод необхідно привести їх до абсолютного вигляду, домноживши на розмір поверхні та зробивши поправку на товщину лінії.

Вузол-умовний перехід – текст вписується у ромб зі співвідношенням діагоналей 2:1. Малювання ромба здійснюється за допомогою того ж методу, що й для квадрату, але відносні координнати точок позначають середини відповідних сторін: (0,0.5), (0.5,0), (1,0.5), (0.5,1). Оскільки ширина та висота поверхні для відображення дорівнюють більшій та меншій діагоналям ромба, висота поверхні буде дорівнювати половині її ширини. Для розрахунку ж ширини поверхні використовується формула:

$$w_s = w_t + 2 \times h_t, \tag{5.1}$$

де w_t – ширина тексту,

h_t – висота тексту.

Останній етап побудови діаграми – відображення стрілок, що позначають переходи між кроками алгоритму. Для поєднання блоків стрілками додамо до кожного об’єкту класу BlockBase поле scope, що буде вказувати на поточний умовний перехід циклу. При виведенні блоку передаватимемо в функцію камери номер вузла, що відповідає блоку. В реалізації камери необхідно отримати блок-нащадок та з’єднати їх стрілками. Форма стрілки залежатиме від типу зв’язку. Для послідовності це пряма лінія донизу. Для умовного переходу це лінії в сторони та донизу. Для завершень циклів скористаємося тим що хибний шлях розміщено справа від умовного переходу та виведемо стрілку через ліву сторону.

Результат відображення тексту та рамок наведено на рис. 5.10, спрощену діаграму класів модуля – в додатку И.

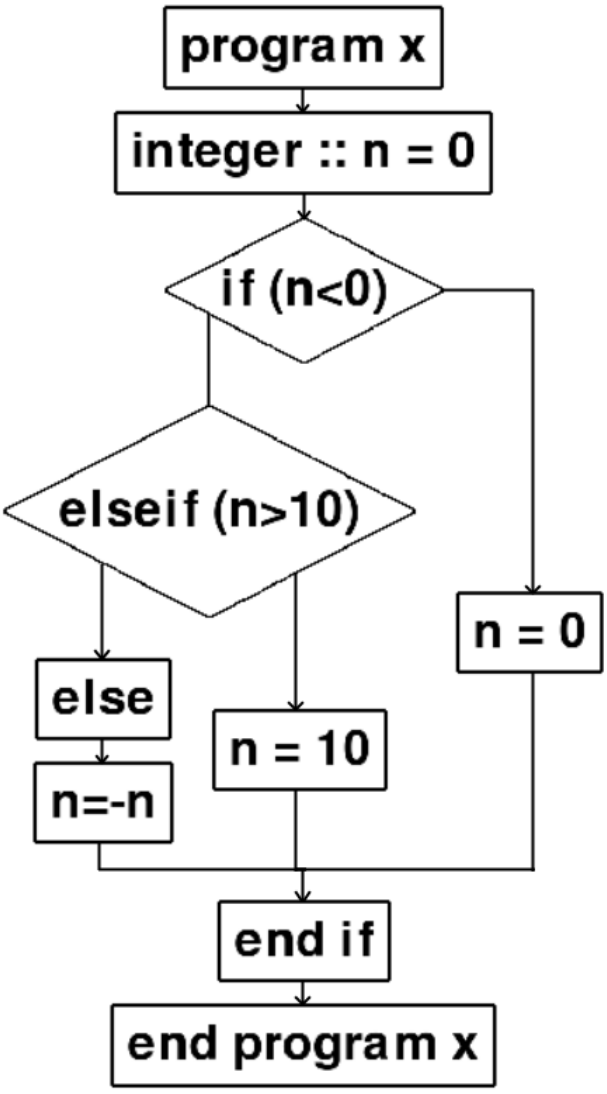

Рисунок 5.10 – Результат виведення тексту та рамок

5.5 Взаємодія модулів

Після завершення роботи над реалізацією модулів відображення та перетворення коду необхідно здійснити реалізацію архітектури ваємоді модулів, що полягає у створенні екземпляра застосунку з модулем відображення, підключенні до нього з модуля керування вихідним кодом та надсилання інформації про діаграми які необхідно побудувати.

Для отримання інформації на сервері (в модулі відображення) використаємо серверний TCP-сокет. TCP (протокол керування передачею – один з двох основний протоколів стеку TCP/IP транспортного рівня. Причиною використання цього протоколу є:

- надійність: пакети, скинуті в мережі, виявляються та повторно передаються відправником;
- впорядкована передача даних: дані зчитуються приймачем повідомлення у тому порядку, в якому їх було надіслано.

На відміну від цього, сокети UDP (протокол датаграм користувача) не є надійними, і дані, зчитувані одержувачем, можуть бути непослідовними відносно записів відправника.

Оскільки в даному застосунку постійна потокова передача даних непотрібнаЮ, а фокус має робитися саме на передачі фіксованих «пакетів», використання TCP позбавляє від необхідності турбуватися про втрату пакетів, надходження неповних даних та багато інших речей, які незмінно трапляються під час спілкування через мережу.

Сокети – абстрактний програмни інтерфейс, через який програма може здійснювати взаємодію з парою «мережева адреса/мережевий порт». Він визначається як одна кінцева точка (endpoint) двостороннього зв’язку між двома програмами, що працюють у мережі. Сокет прив’язаний до номера порту, щоб рівень протоколу TCP міг правильно визначити програму, якій призначено надіслані дані.

Зазвичай сервер працює на певному комп'ютері та має сокет, прив'язаний до певного номера порту. Серверний застосунок просто чекає, прослуховуючи сокет, доки клієнт зробить запит на підключення.

Клієнт має інформацію про ім’я хосту (або мережеву адресу) машини, на якій запущений сервер, та номер порту, на якому розміщено сервер. Щоб зробити запит на підключення, клієнт намагається надіслати дані на машину та порт сервера. Клієнту також потрібно ідентифікувати себе на сервері, щоб прив'язатися до номера локального порту, який він буде використовувати під час цього з'єднання. Зазвичай цей порт призначається системою.

Якщо запит клієнта вдається, сервер приймає підключення. Після прийняття сервер отримує новий сокет, прив'язаний до того ж локального порту, а також віддалену кінцеву точку встановлюну за адресою та портом клієнта. Новий сокет необхідно отримувати щоб сервер міг продовжувати прослуховувати оригінальний «вхідний» сокет для запитів на підключення, одночасно здійснюючи комунікацію з підключеним клієнтом.

У python серверний TCP-сокет можна отримати, викликавши метод socket(family, type), що приймає ідентифікатор сімейства адрес (у випадку розроблюваного засобу – AF_INET, що позначає IPv4-адреси інтернет протоколу) та тип сокета (загалом, SOCK_STREAM стосується TCP а SOCK_DGRAM – UDP протоколу). У результаті отримується об’єкт сокета, що можна налаштувати як на роботу в якості сервера, так і в якості клієнта. Оскільки модуль відображення буде працювати в ролі сервера, налаштуємо сокет на прослуховування. Для цього прив’яжемо сокет до локальної адреси машини та вільного порта за допомогою команди socket.bind((‘localhost’, 60060)). Після цього сокету необхідно дати вказівку прослуховувати зв’язаний порт за допомогою методу listen(1). Параметр «1» повідомляє сокету що необхідно встановити ліміт на одне підключення одночасно.

Після налаштування серверного сокета необхідно прослухати його на вхідне підключення. Це буде робитися з допомогою методу socket.accept(), що блокує потік доки не визначить вхіднного підключення та вертає клієнтський сокет, за допомогою якого можна отримувати та передавати інформацію до підключеної концевої точки,

яка в нашому випадку буде модулем керування вихідним кодом. Після підключення до одного клієнта серверний сокет стає непотрібним на час роботи з ним, тому його можна закрити, використовуючи метод socket.close(). Цей метод зупиняє прослуховування нна порту машини, вивільнює зайняті ресурси, зокрема робить прив’язаний порт знову вільним для використання іншими програмами.

Для надсилання та отримання потоку байтів за допомогою сокета будуть використовуватися методи send() та recv() відповідно. Особливістю роботи з протоколом TCP є те, що відправлення та запис бінарних даних працюють на мережевих буферах. Методи send() та recv() не обов’язково обробляють усі байти, ої їм було передано (або очікуються від них), оскільки їх основна увага приділяється обробці мережевих буферів та дотриманню фіксованих розмірів даних у них. Як правило, вони повертають результат, коли пов'язані з ними мережеві буфери заповнені (надіслані) або спорожнені (отримані). При поверненні вони повідомляють програмісту скільки байт вони обробили. Програміст в свою чергу несе відповідальність викликати їх знову і знову, поки потрібне повідомлення не буде повністю оброблене.

Коли метод recv() повертає ноль байт, це означає, що інша сторона розірвала (або знаходиться в процесі розриву) з'єднання. З цього моменту за цим сокетом даних надходити більше не буде.

Такі протоколи, як HTTP, використовують TCP-сокет лише для однієї передачі. Клієнт надсилає запит, потім читає відповідь. Після цього, сокет закривається та вивільнюється. Це означає, що при користуванні цим протоколом клієнт може виявити кінець відповіді, отримавши 0 байт.

Але якщо за задумом необхідно повторно використовувати сокет для подальших передач, необхідно розуміти що на сокетах відсутній сигнал про припинення передачі (EOT). Якщо зв’язок не було розірвано, програма може очікувати на перезапис невизначену кількість часу, оскільки сокет не повідомлятиме, що інформації для читання більше нема. Тому при роботі з сокетами необхідно в загальному випадку дотримуватися одного з чотирьох підходів:

- фіксована довжина повідомлень;

- завершення повідомлення символом-розмежувачем;
- попередня передача даних про довжину повідомлення;
- розрив з’єднання після закінчення передачі.

Вибір, який підхід використовувати, повністю лишається за програмістом, але деякі шляхи будуть ефективнішими за інші в залежності від мети з’єднання.

З цих підходів найчистішим буде використання третього варіанту, який передбачає попередню передачу розміру повідомлення. Дані з модуля керування вихідним кодом можуть містити кілька діаграм, в залежності від вибраного сегменту. Структура одного повідомлення наведена в таб. 5.4.

Таблиця 5.4 – Структура запису повідомлення для передачі через TCP

<table>
<tr><td>Назва поля</td><td>Заголовок повідомлення</td><td colspan="2">Тіло повідомлення (діаграма 1 … діаграма n)</td></tr>
<tr><td rowspan="2">Внутрішні поля</td><td></td><td>Заголовок діаграми</td><td>Тіло діаграми (вузол 1 ... вузол m)</td></tr>
<tr><td></td><td></td><td>Тіло вузла</td></tr>
<tr><td>Вміст поля</td><td>Кількість діаграм (n)</td><td>Кількість вузлів (m)</td><td>Кортеж (номер вузла, номер хибного нащадка, номер інстинного нащадка, метадані вузла).</td></tr>
</table>

Довжина цього повідомлення в байтах буде передана перед самим повідомленням, після цього буде зчитано необхідний розмір повідомлення та записано його в буфер. Для перетворення потоку байтів з буфера на набір змінних, які можна записати до графу, використаємо python-бібліотеку struct. Ця бібліотека призначена для зчитування/запису даних у двійковому форматі між буфером та C-подібними структурами (об’єктами з впорядкованими полями).

Довжину повідомлення в байтах, кількість діаграм, вузлів та номери вузлів у діаграмі , а також довжину строки метаданих кодуватимемо за допомогою 4-байтних цілих чисел, самі байти строки зчитуватимемо безпесередньо з буфера та декодуматимемо у символьне представлення. Результат цієї операції можна передати у конструктор класу Graph у вигляді переліку кортежів, а отримані діаграми

прив’язати до виводу на екран за допомогою класу Plotter, описаного вище. Вихідний код для декодування повідомлення наведено на рис. 5.11.

```
import struct

from graph.graph import Graph

_list_format = '!i'
_node_format = '!iiii'
_string_format = '!{}s'


class GraphReader:

    def __init__(self, buffer: bytes):
        self._buffer = buffer
        self._offset = 0

    def read_message(self):
        self._read_list(self._read_graph())

    def _read_graph(self):
        result = Graph()
        result.load(self._read_list(self._read_node()))

    def _read_node(self):
        index, false_next, true_next, text_size = self.read_tuple(_node_format)
        text = self.read_tuple(_string_format.format(text_size))
        return index, false_next, true_next, text

    def _read_list(self, item_reader):
        size = self.read_tuple(_list_format)[0]
        while size > 0:
            yield item_reader()
            size -= 1

    def _read_tuple(self, __format: str):
        size = struct.calcsize(__format)
        result = struct.unpack_from(__format, self._buffer, self._offset)
        self._offset += size
        return result
```

Рисунок 5.11 – Вихідний код класу декодування повідомлень

Зі сторони модуля керування вихідним кодом необхідно здійснити зворотну операцію – підключення до сервера, пакування (серіалізацію) даних у двійковий формат та передачу на сервер. За відсутності підключення до сервера необхідно запустити сервера та зробити повторну спробу. За використання вищеописаної архітектури, коли сервер використовує порт 60060 лише під час підключень, цей порт

має більш-менш постійно бути доступним для зайняття новими екземплярами серверів модулю відображення. Це дозволяє користуватися довільною (обмеженою тільки кількістю доступних портів на машині) кількістю пар «модуль керування вихідним кодом-модуль відображення».

Реалізація програмного інтерфейсу TCP-сокетів у Java не сильно відрізняється від Python. Ключовою відмінністю є те що у Java зчитування та запис інформації з/у сокети здійснюється за тим же принципом, що й взаємодія з файлами. Для обміну інформацією неохідно отримати з сокета потоки вводу та виводу та записувати/отримувати дані за їх допомогою. Для полегшення роботи з цими потоками необхідно загорнути їх у програмні обгортки, що атвоматизують процеси буферизації та прочищення (flush) відповідних потоків. Така взаємодія є реалізацією патерну «декоратор», описаного у підрозділі 4.4.

Для створення клієнтського сокета та здійснення спроби підключення використаємо об’єкт класу java.net.Socket. Конструктор цього об’єкту для клієнтського сокета приймає мережеву адресу та номер порта, до якого здійснюється підключення. В даному випадку це буде пара (127.0.0.1, 60060), які визначають серверний порт модуля відображення на локальній машині.

У випадку коли сервер недоступний, що означає що відсутні вільні модулі відображення, конструктор цього класу створює виключення ValueError. В такому випадку необхідно запустити новий екземпляр модуля відображення.

Запуск процесів у Java найпростіше реалізувати за допомогою ProcessBuilder API. Це набір класів, що дозволяють викликати команди операційної системи з коду Java. Основним параметром, що приймає конструктор класу ProcessBuilder, є строка що представляє собою команду терміналу операційної системи. При встановленому на комп’ютер інтерпретаторі Python цією командою буде строка «python». Додатково конструктор приймає параметр для команди, що в нашому випадку буде місцезнаходженням скрипта на python, наприклад «GraphPlotter.py». Для запуску зазначеного процесу необхідно викликкати метод ProcessBuilder.start(), що дає операційній системі команду на виконання зазначеного процесу та вертає об’єкт, що використовує унікальний ідентифікатор процесу в системі для взаємодії з ним

(завершення, отримання коду результату тощо). Якщо робота з процесом полягає лише в його запуску, об’єкт процесу що отримується в методі start() можна проігнорувати. Логічно передбачити що закриття модуля керування вихідним кодом не означає відсутності потреби перестати працювати з діаграмою, а отже користувач сам зможе закрити застосунок коли він буде йому непотрібний.

Для запису даних у сокет отримаємо об’єкт потоку виводу за допомогою методу Socket.getOutputStream(). Для можливості запису буферизованих даних загорнемо цей об’єкт у об’єкт класу PrintStream, що дозволяє записувати у потік двійкову репрезентацію базових типів Java. Порядок запису аналогічний порядку зчитування даних, описаному вище.

Для спрощення визначення розміру повідомлення, що надсилається, використано той самий патерн «декоратор». Написано додатковий клас-обгортку GraphWriter для PrintStream, що містить методи для запису усіх необхідних примітивів, а також для запису списків та діаграм. Виклики методів цього класу записують повідомлення в тимчасовий буфер, використовуючи PrintStream та об’єкт класу ByteArrayOutputStream, що здійснює тимчасове зберігання записаних даних у динамічному масиві байтів. Після завершення запису повідомлення, метод flush() записує довжину та тіло повідомлення у вихідний буфер сокета.

6 РОЗРОБКА ІНТЕРФЕЙСУ КОРИСТУВАЧА

Окрім основного функціоналу виведення діаграми, переміщення її екраном та зміни масштабу, необхідно також реалізувати частини користувацького інтерфейсу, що відповідають за інший функціонал застосунку. Вони включають засоби для роботи з діаграмою а також синхронізацію вихідного коду з діаграмою, що побудована на його основі. Цей розділ покриває всі додаткові шляхи взаємодії користувача з засобом, реалізованим в процесі виконання цієї роботи.

6.1 Вибір активного сегмента

Першим елементом користувацького інтерфейсу, з яким буде взаємодіяти користувач при використанні модуля перевторення вихідного коду це можливість вибору активного сегмента для відображення діаграми.

Для плагіну, розробленого у роботі, вибір активного сегмента інтегровано з системою рефакторингу, наявною у середовищі Photran. Застосування операцій рефакторингу – основна функціональна мета створення Photran, і для цього у середовищі програмісту надаються прості засоби, за допомогою яких можна розширити існуючий та додати новий функціонал до цих операцій.

Photran поділяє операції рефакторингу на дві категорії: рефакторинг на основі тестового редактора, який вимагає від користувача вибрати частину вихідного коду програми FORTRAN у текстовому редакторі, над яким буде виконуватися операція, та рефакторинг ресурсів, які застосовуються до цілих файлів. Щоб створити новий рефакторинг, необхідно вирішити, чи буде ця операція рефакторингом на основі текстового редактора або рефакторингом для ресурсів. Photran пропонує різні суперкласи для кожного. Для реалізації операції рефакторингу розробник створює конкретний підклас і додає рядок XML до конфігураційного файлу, щоб зареєструвати новий рефакторинг у фреймворку Photran. Цей конкретний клас має визначити ряд операцій, перша з яких дає назву новому рефакторингу. Ця назва стає

текстовим полем у меню рефакторингу. Також ця назва використовується для опису операції у меню відміни останніх дій.

Другим кроком для створення рефакторинга є перевірка початкових умов. Ці перевірки мають на меті переконатися, що користувачем обрано коректний сегмент файлу, у вихідному коді відсутні синтаксичні помилки, вибраний файл не є заблокованим для запису тощо.

Третім кроком є отримання вводу користувача. Наприкла, рефакторинг перейменування змінної вимагає вводу нового імені, операція додавання параметра до виклику функції має отримати назву та тип параметра тощо.

Після цього, необхідно провести перевірку фінальних передумов. Цей етап необхідний для того щоб валідувати ввід користувача, переконатися у тому що поведінкка програми не буде змінена внаслідок виконання цієї операції та вихідний код можна буде скомпілювати.

Фінальним кроком буде застосувати потрібні зміни. Після перевірки всіх передумов, у цій операції описані зміни, що будуть виконані над вихідним кодом.

Завдяки використанню можливостей Eclipse та функціональних особливостей Java, такий рефакторинг може бути автоматично вбудований у візуальний інтерфейс Photran. Рефакторинги що вносять зміни також надають вікно підтвердження виконання операції та вікно попереднього перегляду, що дає користувачу можливість побачити зміни, що будуть внесені у вихідний код в результаті виконання операції.

Процес реєстрації нової операції продемонстровано на прикладі операції побудови діаграми. Ця операція має приймати довільний сегмент вихідного коду програми, тому за її допомогою можна розглянути обидва варіанти роботи з вихідним кодом. Спершу цю операцію реалізовано для сукупності файлів.

Для реалізації операції з взаємодією з ресурсами, створимо клас GraphPlotterOperation, успадкований від FortranResourceRefactoring.

Потім необхідно відредагувати файл plugin.xml для внесення рядків, що зареєструють нову операцію у фреймворку Photran. Далі наведено приклад сегменту коду цього файлу, необхідного для реєстрації цієї операції:

<group>

```
<submenu name="Flowchart"><!—Operations to visualize source code. -->
  <resourceRefactoring class="org.eclipse.photran.internal.core.refactoring.
GraphPlotterOperation"/>
</submenu>
</group>
```

Для реалізації процедури виводу діаграми необхідно визначити у створеному класі перелік методів, що будуть викликатися фреймворком рефакторингів при виборі в меню зареєстрованої опції Flowchart (рис. 6.1):

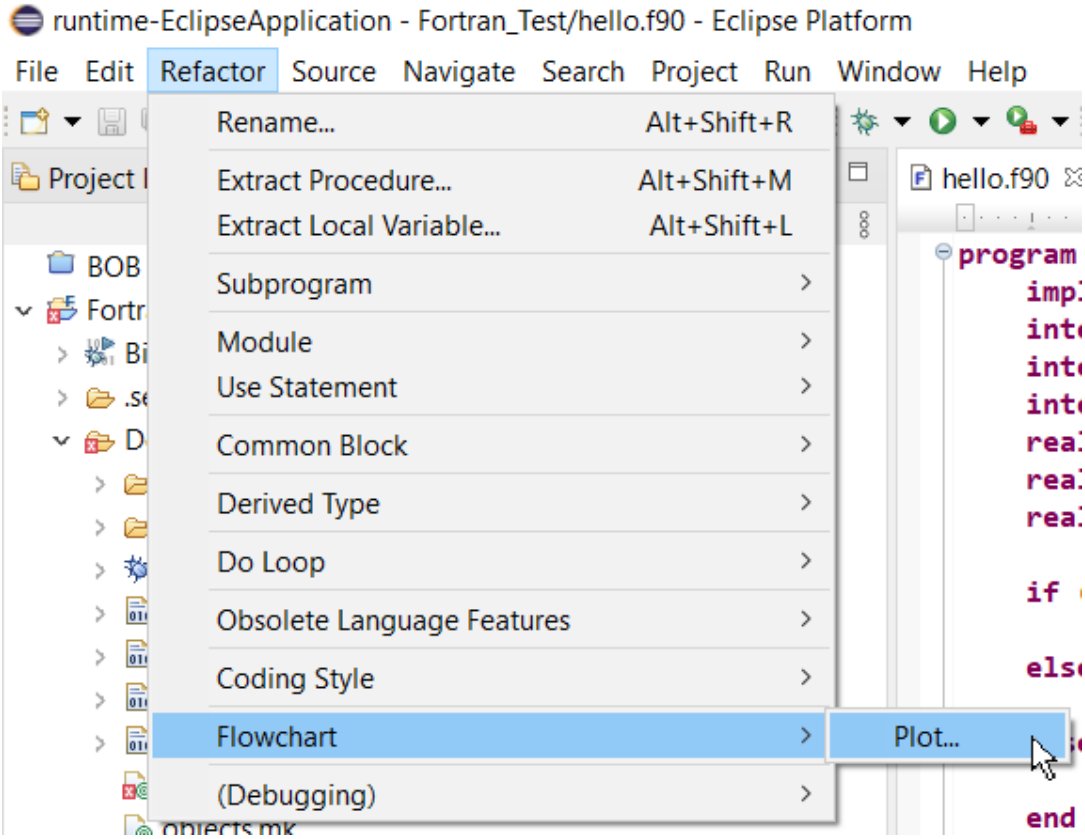

Рисунок 6.1 – Відображення операції у меню Photran

Метод getName() вертає об’єкт строки, що буде записано у відповідний пункт меню при підключенні цієї операції до фреймворку. Назвемо цю конкретну операцію Plot (побудувати).

Метод doCheckInitialConditions() проводить перевірку початкових умов. В даному випадку необхідно виконати стандартні для операцій рефакторингу перевірки:

- перевірити чи проект має увімкнену опцію рефакторингу;
- видалити з набору оброблюваних файлів файли вихідного коду фіксованої форми;
- видалити з оброблюваних файлів файли з препроцесингом.

Ці перевірки необхідні для забезпечення коректної роботи парсера синтаксису вихідного коду оскільки вони здійснюють видалення файлів, з якими парсер працювати не може.

Оскільки користувацький ввід непотрібен, метод doCreateChange() лишаємо пустим, а парсинг та побудову діаграми будемо виконувати у тілі методу doCheckFinalConditions(). Спершу у тілі методу необхідно виконати перевірку на можливість компіляції коду, тобто на відсутність у ньому синтаксичних помилок. Цю перевірку можна зробити викликавши метод logVPGErrors() для всіх вибраних користувачем файлів. Цей список файлів передається як аргумент до цього метода. Після цього необхідно обійти цей список файлів та для кожного з них викликати створений метод parseFlowcharts(). Цей метод приймає на вхід об'єкт FortranAST, що являє собою вузол-корінь синтаксичного дерева файлу та надає доступ до всіх структур вихідного коду. У тілі цього методу цей об’єкт буде передано класу ASTConverter, описаному у підрозділі 5.3. Після збору інформації про діаграми, присутні у вихідному коді файлу, їх буде додано до загального списку діаграм, що необхідно створити.

Об’єкт ASTConverter для парсингу вказаного файлу потрібно отримати та перевірити на помилки. Для цього використаємо об’єкт поточного файлу IFile. Метод this.vpg.acquirePermanentAST(IFile). Якщо цей метод вертає null, це означає що файл неможливо прочитати (наприклад, це не файл FORTRAN). В такому випадку потрібно повідомити про це фреймворк Photran, створивши повідомлення про помилку IntroImplicitNoneRefactoring_SelectedFileCannotBeParsed. При виникненні виключень під час обробки файлу або всього набору файлів, необхідно переконатися що всі об’єкти FortranAST буде вивільнено. Для цього використано конструкцію try…finally, яка виконує код у блоці-фіналізаторі за будь-яких обставин (навіть якщо виконання функції завершилося до передачі керування цьому блокові).

Для передачі отриманих діаграм модулю відображення виконаємо набір операцій, що описані у підрозділі 5.5:

- перетворити набір діаграм на бінарний буфер за допомогою методу GraphWriter.write();

- зробити спробу надіслати цей буфер на сервер. Якщо спроба невдала, створити новий екземпляр застосунку модуля відображення;

- повторити спробу підключення.

Для забезпечення стабільності функціонування застосунку виконаємо три спроби підключення до модуля відображення, у випадку відмови повідомимо користувача про невдачу за допомогою змінної refactoring status (рис. 6.2).

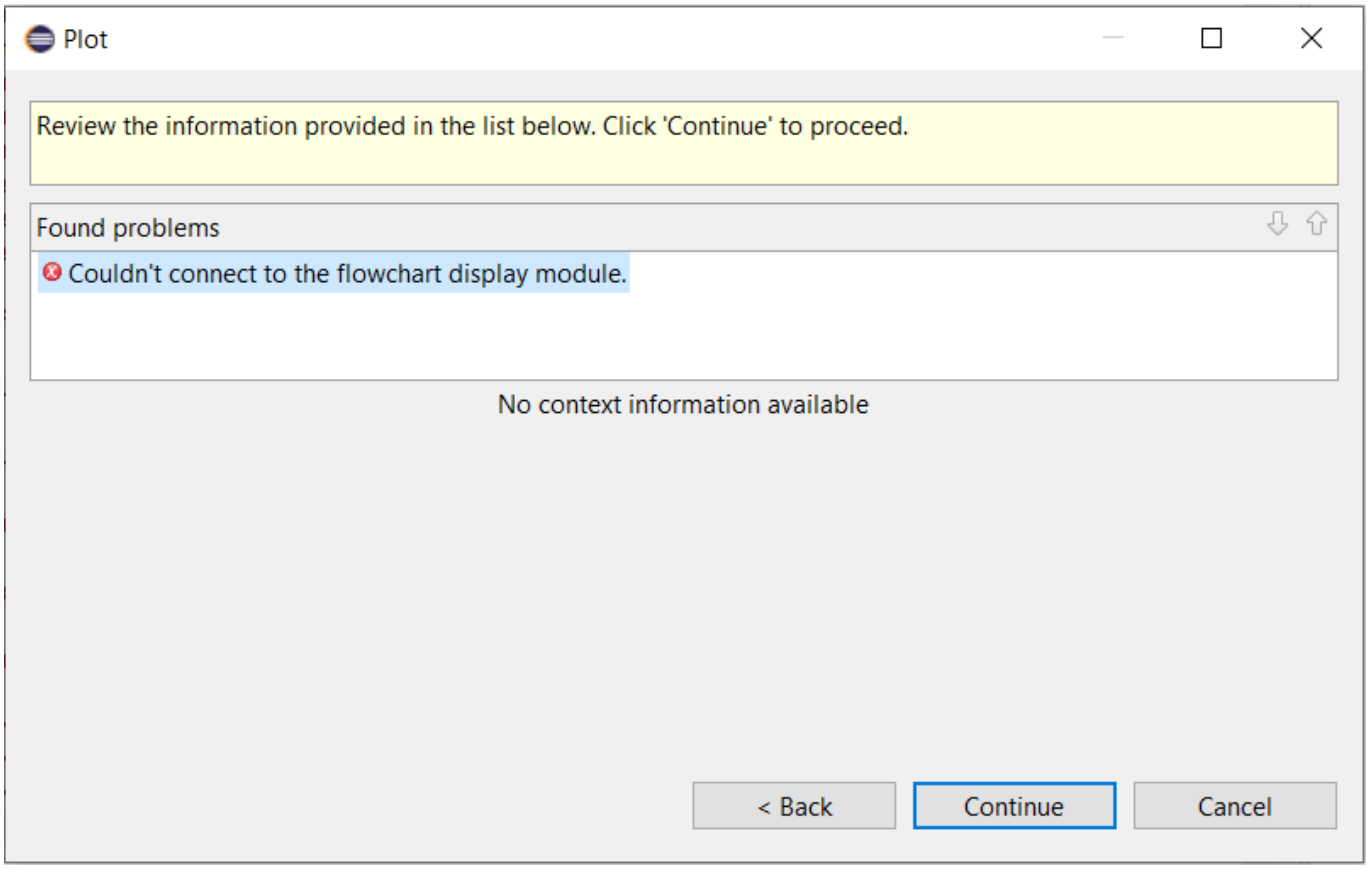

Рисунок 6.2 – Повідомлення про помилку підключення до сервісу

Переданий буфер даних буде розпаковано модулем відображення, який за допомогою модуля побудови діаграми покаже її користувачу на екрані. Послідовність

виконання дій для операції GraphPlotterOperation проілюстровано на діаграмі послідовності, наведену в додатку К.

Реалізація операції побудови діаграми для виділеного тексту у файлі відрізняється процесом реєстрації операції та отримання об’єкту IASTNode для парсингу вузлів. Для реєстрації операції над виділенням використовується такий же запис XML, як для ресурсної, але з значенням тегу “editorRefactoring” замість “resourceRefactoring”. Окрім цього, необхідно успадкувати клас FortranEditorRefactoring. Цей клас надає доступ до нової змінної – selectedRegionInEditor, що реалізовує клас ITextSelection, та містить посилання на сегмент коду, вибраний у файлі. За допомогою цієї змінної, а також змінної astOfFileInEditor, що містить об’єкт FortranASTNode для поточного файлу, можна отримати об’єкт IASTNode, що включає в себе все виділення. Далі його можна передати на парсинг за тою ж процедурою, що використовується для операції над файлами. Також, необхідно прибрати виклики методів перевірки коректності вибраних файлів, натомість виконати перевірку на наявність тексту в об’єкті ITextSelection і у випадку його відсутності вивести повідомлення для користувача.

При передачі інформації для побудови діаграми автоматично здійснюється перевірка вихідного коду на додані або видалені вузли. Такі вузли надсилаються до модуля побудови діаграми окремо, де додаються до наявної структури даних для економії часу виконання операції та даних.

6.2 Перехід до елемента

Операція переходу до елемента полягає в тому, що після вибору певного сегменту коду в текстовому редакторі та виборі опції «перехід до елемента», в модулі відображення стануть виділеними ті елементи, що відповідають виділенню тексту в редакторі. Для реалізації цієї функції зареєстровано нову операцію рефакторинга виділення – SelectNodeOperation.

При запуску цієї операції необхідно отримати перелік вузлів, що відповідають виділеному сегменту. Для початку отримаємо стартовий та кінцевий номери строк,

які відповідають виділеному сегменту коду. Для цього інтерфейс ITextSelection містить методи getStartLine() та getEndLine(). Після цього пройдемо у циклі по переліку вибраних строк та спробуємо отримати вираз (Token), що містить ця строка. Для пустих строк або строк з коментарями, цей вираз матиме значення null. Для кожного отриманого таким чином виразу необхідно визначити відовідний вузол на діаграмі. Для цього додамо у кожен об’єкт Scope(), що визначає діаграму, словник HashMap<IASTNode, Node>, в який записуватимемо пари вузлів, що співвідносяться. При перевірці за дуже невеликий час можливо провести пошук по словнику та визначити, чи відповідає даному виразу якийсь вузол на діаграмі.

Далі отриманий перелік вузлів необхідно надіслати. Оскільки поточна реалізація протоколу комунікації підтримує тільки операцію побудови діаграми, необхідно внести нові поля до даних та структури повідомлення. Додамо в початок повідомлення заголовок, що буде ідентифікувати тип повідомлення (унікальний ідентифікатор операції). Позначимо операцію побудови діаграми кодом 0, операцію переходу до елемента – 1. Також для реалізації цієї операції необхідно додати унікальні ідентифікатори діаграм, щоб оновлення вузла проходило на потрібній діаграмі. Додамо цей ідентифікатор перед переліком вузлів при побудові діаграм. Структура даних, що буде передана для виконання переходу до елемента, наведена в таб. 6.1.

Таблиця 6.1 – Оновлена структура повідомлення для передачі вибраних вузлів

Назва поля	Заголовок повідомлення	Тіло повідомлення	
Внутрішні поля		Заголовок списку	Тіло списку (вузол 1 ... вузол n)
			Тіло вузла
Вміст поля	Код операції (1)	Кількість вибраних вузлів (n)	Кортеж (номер діаграми, номер вузла).

На стороні модуля відображення необхідно внести зміни у функцію десеріалізації даних для обробки оновленої структури повідомлення. В залежності від коду повідомлення, необхідно викликати або операцію побудови діаграми, описану

вище, або операцію виділення елементів. Виділення елементів відбувається за подібним принципом до того, як було реалізовано пошук вузлів у модулі керування вихідним кодом.

Змінимо клас DiagramBlock для зберігання всередині нього ідентифікатора вузла та стану «вибрано/не вибрано» (is_selected). Реалізуємо патерн «відвідувач», описаний у підрозділі 4.4, для обходу дерева, та внесемо відповідні зміни у класи BlockBase та StackBlock. Абстракцією об’єкта відвідувач буде клас BlockVisitor, який містить метод visit() для обробки довільного об’єкту BlockBase. Класи, успадковані від BlockBase міститимуть метод invite(), що прийматиме об’єкт-відвідувач та надаватиме йому доступ до дочірніх елементів. Наприклад, для реалізації патерну відвідувач для класу BlockStack достатньо передати йому всі дочірні елементи з внутрішнього списку (рис. 6.3).

```
def accept(self, visitor: BlockVisitor):
    for c in self.children:
        visitor.visit(c)
```

Рисунок 6.3 – Реалізація патерну «відвідувач» для класу BlockStack

Для класу DiagramBlock необхідно створити особливий функціонал в об’єкті відвідувача. Щоб відділити реалізацію алгоритму обходу об'єкта від відвідувача, дамо можливість користувачу класу BlockVisitor самостійно визначати алгоритм обробки для заданого типу блоку. Для цього створимо в BlockVisitor словник, що співставлятиме клас блоку з функцією обробки блоку. Реалізуємо оператор індексації (квадратні дужки) для доступу до значення цього словника (рис. 6.4).

Користувач матиме можливість за допомогою цього індексатора додавати власні методи обробки вузлів до відвідувача «на ходу». Це дозволяє використовувати один об’єкт-відвідувач для виконання великої кількості операцій без необхідності наслідувати його при кожному використанні.

Для завершення реалізації вибору елемента створимо об’єкт відвідувача, передамо йому локально створену функцію, що шукає номер вузла у переліку

визначених та оновлює змінну стану блоку в залежності від наявнності його у цьому переліку. Для відображення стану блоку виведемо поверх блоку кольорову рамку.

```
from graph.plotter.block import BlockBase


class BlockVisitor:
    _doors = dict()

    def visit(self, block: BlockBase):
        clazz = type(block)

        if clazz in self._doors:
            self._doors[clazz](block)
        else:
            block.accept(self)

    def __getitem__(self, item):
        return self._doors[item]

    def __setitem__(self, key, value):
        self._doors[key] = value

    def __delitem__(self, key):
        del self._doors[key]
```

Рисунок 6.4. Вихіднний код класу відвідувача блоків

Кольором за замовчуванням є чорний, тому можемо встановити колір виділеного блоку як синій. Приклад процесу переходу до елемента наведено на рис. 6.5.

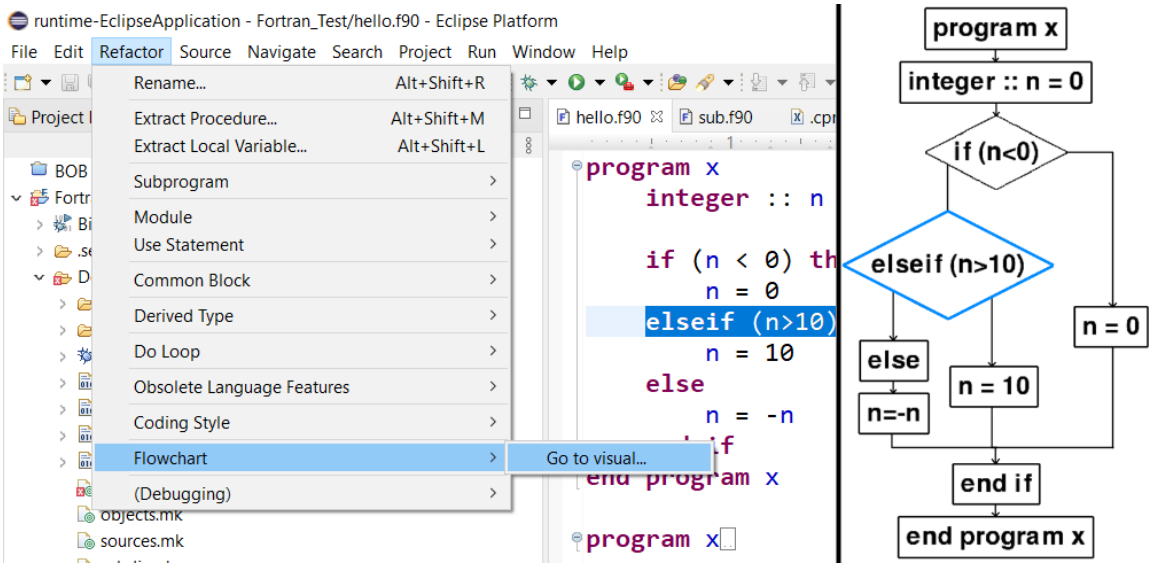

Рисунок 6.5 – Процес переходу до елемента

6.3 Переміщення елемента

Можливість змінити розташування елемента на екрані може виявитися для користувача дуже важливою якщо з якихось причин стандартне розміщення блоків при побудові діаграми його не задовільнить. Для зручності роботи з набором елементів, поведінка перетягування працюватиме так:

- при натиску лівою клавішею миші на елемент він стає виділеним
- при натиску правою клавішею по виділеному елементу він перестає бути виділеним
- при натиску по пустій області правою клавішею виділення знімається з усіх елементів
- перетягувати все виділення можна, затиснувши клавішу Ctrl.

Для реалізації цієї функції використаємо клас-відвідувач, створений вище. При отриманні події натиску лівої клавіші миші необхідно отримати координати миші аналогічно до реалізації переміщення сцени. Позицію курсора треба тепер перетворити на координатну сітку діаграми. Для цього використаємо алгоритм, зворотний до методу draw_surface, описаного в підрозділі 5.4.

Отримавши координати курсора миші відносно діаграм, запусктимо об’єкт-відвідувач обійти дерево. За допомогою метода pygame.rect.collidepoint() можна перевірити чи курсор миші перебуває над даним елементом. Визначивши найбільш «вкладений» блок, встановимо флаг is_selected в позицію True. Аналогічним чином обходитимемо дерево при отриманні натиску правої клавіші. Якщо в результаті обходу жодного елемента не змінено, виконаємо другий обхід для очищення всього виділення.

Для реалізації самого перетягування блоку додамо до нього змінну offset. За допомогою відвідувача для виділених об’єктів цю змінну можна оновлювати в залежності від переміщень миші, доки флаг Window.mouse_pressed встановлений в позицію True. Додавши до цієї поведінки обробку натиснення клавіші Ctrl, буде реалізовано весь необхідний для цього сценарію використання функціонал.

6.4 Перехід до джерела

Для реалізації операції переходу до джерела надамо користувачу можливість відкрити відповідний елемент в коді, натиснувши лівою клавішею миші на виділений елемент з затиснутим Shift. Виявлення натиску відбувається за принципом, описаним вище. Після отримання номеру натисненого вузла, необхідно передати його модулю керування вихідним кодом. Для цього скористаємося тим самим ідентифікатором операції переходу до елемента (1) та форматом повідомлення.

При отриманні повідомлення в модулі керування вихідним кодом необхідно отримати з номера вузла відповідний об’єкт IASTNode. Оскільки ці об’єкти зберігаються всередині об’єктів Node, цього можна досягти перебором списку діаграм з пошуком потрібної за номером та перебором списку вузлів діаграми з пошуком номеру. Після знаходження об’єкту IASTNode можна отримати з нього значення зсуву та довжини у файлі. Для отримання значення зсуву скористаємося методом getFirstToken(), що видасть об’єкт класу Token з потрібним нам методом getFileOffset(). Для визначення розміру виділення подібним чином отримаємо останній токен за допомогою методу getLastToken(). З нього можливо отримати зсув останнього символу як getFileOffset() + getLength(). Віднявши від позиції останнього символу позицію першого, отримаємо довжину.

Photran не надає вбудованих методів взаємодії з текстовим редактором, тому доведеться скористатися API Eclipse. Для відкриття файлу необхідно отримати об’єкт Page, що являє собою робочу область Eclipse, та об’єкт IFile. Перший можливо отримати виконавши запит PlatformUI.getWorkbench().getWorkbenchWindows()[0] .getPages()[0]. Для отримання останнього можна скористатися методом Token.getLogicalFile. Для відкриття файлу у Eclipse скористаємося функцією IDE.openEditor(page, file). В результаті отримаємо об’єкт текстового редактора. Для того, щоб виставити в ньому виділення, необхілно викликати метод ITextEditor.selectAndReveal(offset, length), куди передано отримані раніше значення. Але викликати цей метод у потоці операції не можна, необхідно використати поток користувацького інтерфейсу. Для цього створено об’єкт класу Runnable, в методі run()

якого виконано цей код, а цей об’єкт викликано через диспетчер потоку користувацького інтерфейсу (рис. 6.6).

```
final int start_offset = firstToken.getFileOffset();
final int end_offset = lastToken.getFileOffset()+lastToken.getLength();

Display.getDefault().syncExec(new Runnable() {
    public void run() {
        try
        {
            ((ITextEditor)IDE.openEditor(
                PlatformUI.getWorkbench().getWorkbenchWindows()[0].getPages()[0], fileInEditor))
                    .selectAndReveal(start_offset, end_offset - start_offset);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
});
```

Рисунок 6.6 – Відкриття редактора та виділення тексту

Внаслідок застосування вищеописаних змін користувач має змогу натиснути на виділений елемент з затисненим Shift та побачити відповідний йому сегмент вихідного коду в редакторі. Процес виконання операції «перехід до джерела» наведено на рис. 6.7.

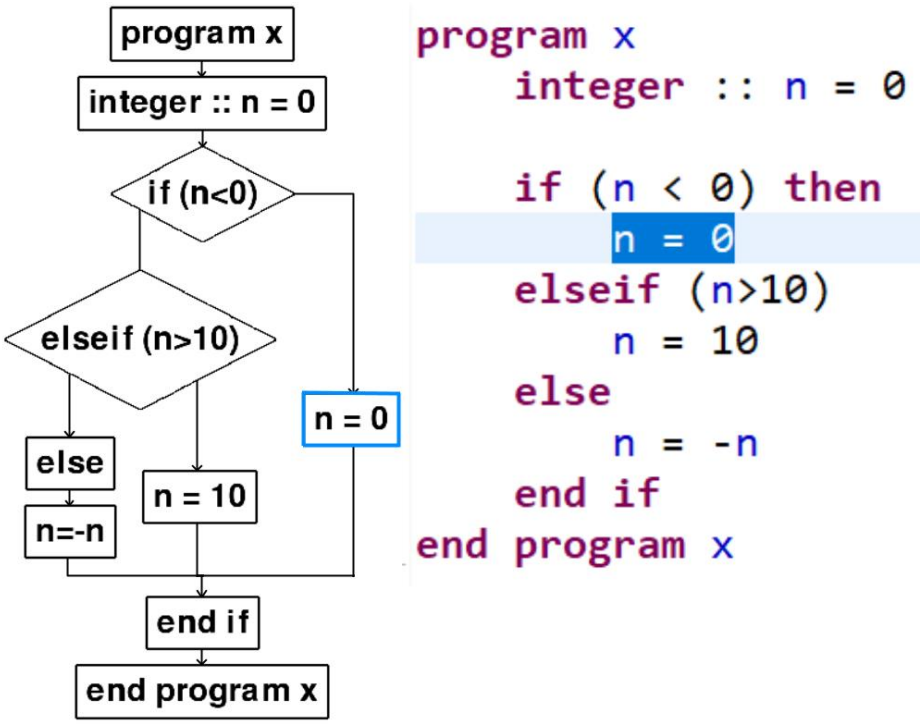

Рисунок 6.7 – Перехід до джерела

6.5 Текстовий пошук

Оскільки деякі діаграми алгоритмів можуть мати занадто великий розмір для пошуку елементів вручну (за допомогою перетягування діаграми та зміни масштабу), необхідно надати користувачу можливість здійснювати текстовий пошук. Для цього необхідно реалізувати текстове поле.

Створено клас TextBox, який містить у собі положення та розмір текстового поля у форматі pygame.surface.Surface. Виведення цього елементу на екран реалізовано за допомогою того ж механізму, що й для блоків діаграми. Метод відображення цього об’єкту на екрані додається до списку операцій, що треба виконати під час Window.main_cycle(). Клас TextBox містить змінну text, що зберігатиме введене значення. Виведення цього тексту відбувається за допомогою методу pygame.Font.Render(), так само як у DiagramBlock. Навколо тексту малюється рамка та задній фон заливається фоновим кольором для можливості відрізнити текстове поле від фону діаграми та виведення поверх її елементів (рис. 6.8).

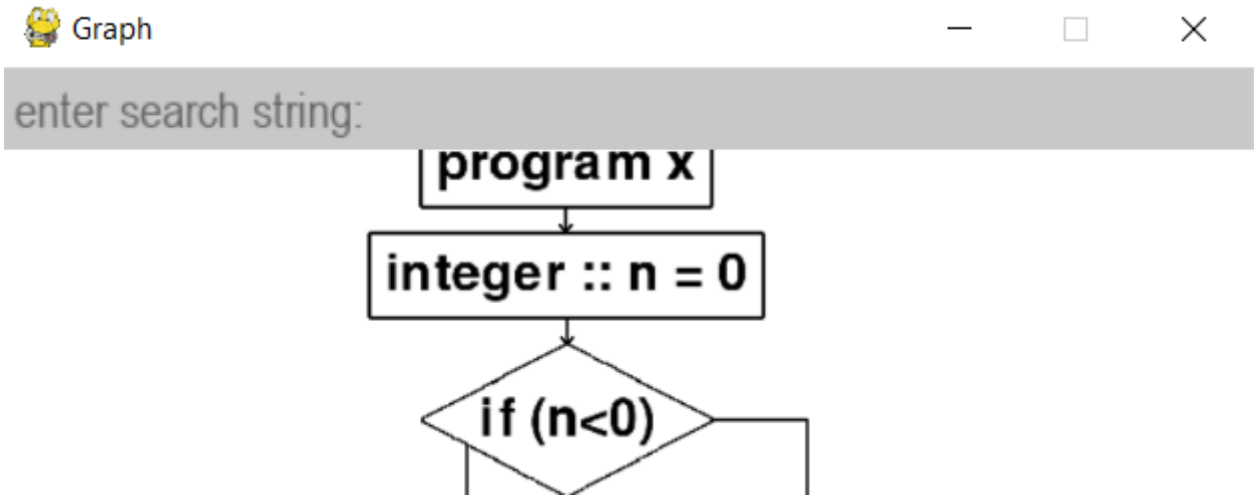

Рисунок 6.8 – Поле текстового пошуку

Для редагування тексту необхідно «підписати» створений об’єкт TextBox на вхідні події натиску клавіш у класі Window та обробляти всі клавіші-символи клавіатури. Додатково збережено поточний стан клавіші Shift для оброблення великих літер. При введенні користувачем певного символа він додається до значення

тексту та за допомогою об’єкту-відвідувача, описаного вище, здійснюється обхід діаграм та виділення блоків, які містять зазначений текст. Підписавши текстове поле на подію натиску клавіші Backspace, додамо можливість видалення останнього символа. Вихідний код класу TextBox наведено на рис. 6.9.

```
import pygame
from pygame import Rect
from pygame.event import Event


class TextBox:
    _box: Rect = Rect(0, 0, 0, 0)
    _text: str = ''
    _target = None
    _hint = 'enter search string:'

    def __init__(self, size):
        self._back = pygame.Surface(size)
        self._back.fill(pygame.Color(200, 200, 200))
        self._update_target()

    def _process_event(self, event: Event):
        if event.type != pygame.KEYDOWN or not event.unicode:
            return

        if event.key == pygame.K_BACKSPACE:
            if self._text:
                self._text = self._text[:-1]
        else:
            symbol = event.unicode
            if event.mod & (pygame.KMOD_LSHIFT | pygame.KMOD_RSHIFT):
                symbol = symbol.upper()
            self._text += symbol

        self._update_target()

    def _update_target(self):
        self._target = pygame.font.SysFont('arial', 20).render(
            self._text or self._hint, True, pygame.Color('black') if self._text else pygame.Color(100, 100, 100))

    def draw(self, camera):
        camera(self._back, (0, 0))
        camera(self._target, (5, 5))
```

Рисунок 6.9 – Вихідний код класу TextBox

6.6 Емуляція виконання

Для полегшення орієнтування у побудованій діаграмі та алгоритмі, що лежить в її основі, додано можливість переходу між вузлами. Для цього створено клас RunEmulator, в якому зберігається поточний стан емуляції алгоритму. Для вибору стартового елемента використано подвійний натиск на елемент. У класі RunEmulator зберігається час останнього натиску на елемент, отримані за допомогою методів, описаних вище у цьому розділі. При повторному натиску на той же блок він

записується у внутрішню змінну active_block. Для відображення активного блоку на екрані цей клас виводить окремий індикатор поверх елемента (рис. 6.10).

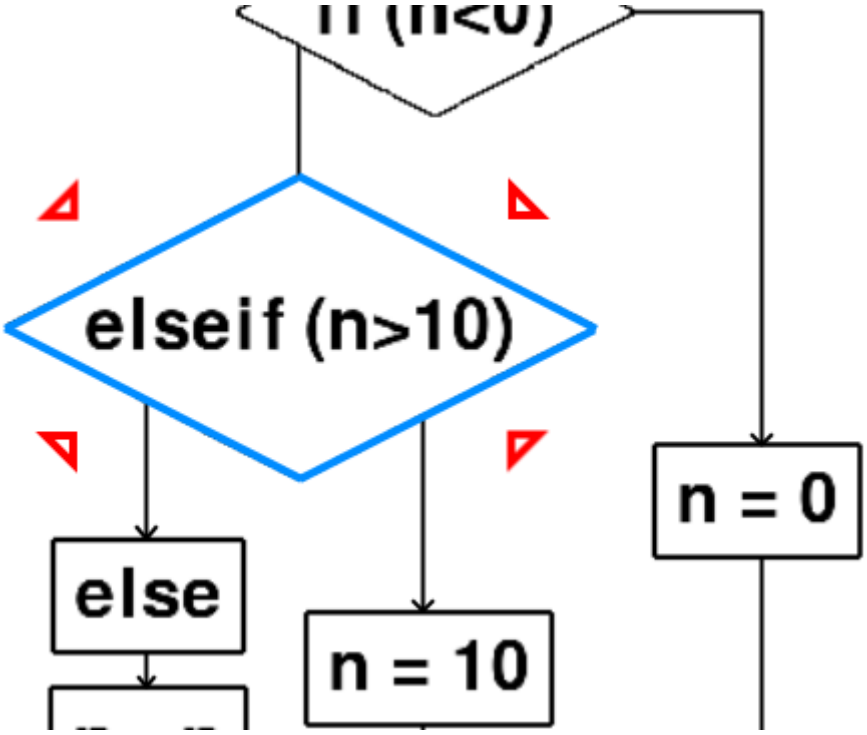

Рисунок 6.10 – Індикатор поточного кроку алгоритму

Для здійснення навігації алгоритмом використані клавіші стрілок. Для вузла з переходом послідовності активний елемент переходить на наступний при натиску стрілки «донизу». Для вузлів – умовних переходів стрілки «донизу» та «ліворуч» здійснюють перехід на хибний нащадок, стрілка «праворуч» – перехід на істинний нащадок.

Стрілка «вгору здійснює перехід до попередньо вибраного елемента. Для цього список попередньо вибраних елементів зберігається у об’єкті RunEmulator, та оновлюється при кожному переході. При виборі нового вузла цей список буде очищено.

6.7 Історія навігації

При користуванні застосунком можлива ситуація коли необхідно швидко перемикатися між нещодавно вибраними елементами. Наприклад, користувач переглядає пов’язані між собою ділянки діаграми для визначення особливостей їх взаємодії. В такому випадку необхідно надати можливість повернутися до попередньо вибраних елементів.

Для реалізації такої поведінки необхідно перемістити керування виділенням в окремий клас SelectionManager. Цей клас виконуватиме всі дії з вибору елементів, описані вище в цьому розділі. Додатково у класі зберігатиметься історія вибраних елементів. Для переключення між елементами в історії використано стандартні комбінації клавіш Ctrl+Z, Ctrl+Shift+Z для переходу назад та вперед відповідно.

ВИСНОВКИ

В ході роботи над магістерською дисертацію виконано розробку програмного засобу автоматизації відлагодження вихідного коду високої зв’язаності. Цей засіб вирішує проблему відсутності інструмента, який би дозволив розробникам програмного забезпечення, перед якими стоїть задача рефакторингу заплутаної архітектури виконувати візуалізацію потоку керування програм у вигляді блок-схеми.

Для постановки проблеми виконано аналіз наявних рішень. Порівнявши засоби, які доступні сьогодні на ринку інструментів візуалізації, було встановлено що більшість таких інструментів не мають всього необхідного переліку функцій, якими користуються розробники при написанні програм.

Було визначено розповсюджені методи інтеграції, які застосовуються для вирішення цієх проблеми. Деякі інструменти вирішують її за допомогою інтеграції у середовища розробки у вигляді розширень-плагінів. Такий підхід має свої недоліки, оскільки весь код зі збору інформації про вихідний код, перетворення її на структуру даних потоку керування та виведення цієї структури у вигляді блок-схеми на екран неможливо повторно використати. Для вирішення цієї проблеми у ході виконання магістерської дисертації було розроблено програмний засіб, що дозволяє повторно використовувати частини свого функціоналу на різних платформах, операційних системах та виконувати інтеграції з популярними середовищами розробки для довільної мови програмування.

На основі результатів цього аналізу було поставлено задачі, що має виконати розроблюваний модуль, та базуючись на них сформовано вимоги до системи. Функціональними вимогами є можливість будувати блок-схему на основі вибраного сегмента вихідного коду, переміщення діаграми та її вузлів, наближення та віддалення, навігація між структурами вихідного коду та їх відповідниками на блок-схемі. Нефункціональними вимогами є платформо-незалежність та швидкодія.

Для планування реалізації системи визначено сценарії використання. Розроблюваний засіб буде використовуватися одним актором – користувачем, та мати два способи взаємодії – інтерфейс до вихідного коду та інтерфейс до діаграми.

Сценарії використання є дещо більш розширеним та структурованим записом вимог до системи.

Складниками засобу є три компоненти – модуль керування вихідним кодом, модуль побудови діаграми та модуль відображення. Перед реалізацією було проаналізовано та вибрано технології для створення цих трьох модулів та налагодження зв’язку між ними. Модуль керування вихідним кодом виконано як плагін до середовища розробки Photran та написано мовою Java на базі фреймворку плагінів Eclipse IDE та фреймворку рефакторингів Photran. Для розробки модуля побудови діаграми було запропоновано скорочений запис блок-схеми за допомогою компактної структури даних, якою користуватимуться всі модулі для передачі інформації. Модуль побудови діаграми та модуль відображення виконано мовою python для забезпечення крос-платформеності та простоти запуску. Для розробки модуля відображення було використано бібліотеку pygame, що надає інструменти з точнного керування головним циклом програми, виведенням зображень на екран а також користувацьким вводом.

Для підключення модулів один до одного було використано сервісний підхід на основі моделі «клієнт-сервер». Зв'язок між модулями організовано за домогою розробленого протоколу на основі TCP-каналу. Було використано бібліотеку struct у python а також PrintStream та ProcessBuilder у Java для серіалізації даних та запуску модуля відображення відповідно.

Розроблено користувацький інтерфейс для модулів керування вихідним кодом та відображення в якому реалізовано операції побудови діаграми, вибору активного сегмента коду, навігація між діаграмою та текстом програми а також емуляцію виконання алгоритму.

В процесі розробки засобу було застосовано досвід роботи з життєвим циклом продукту, популярними технологіями та мовами програмування, отриманий за час проходження магістерської програми. Внаслідок реалізації поставлених задач отримано досвід з проектування розподіленої системи, роботи з графічними бібліотеками, планування архітектури, створення ефективних та компактних

структур даних для забезпечення швидкодії та універсальності взаємодії компонентів системи.

Розроблений засіб містить у своїй основі архітектуру, яка заохочує програміста розвивати та розширбвати функціонал засобу шляхом реалізації модулів керування вихідним кодом, що можуть працювати з іншими текстовими редакторами та середовищами розробки. Модуль керування вихідним кодом в процесі вдосконалення архітектури за необхіності можливо буде розділити на парсер мови програмування та інтеграційний модуль, що буде лише передавати команди з інтегрованої системи та до неї. Це дозволить повторно використовувати не лише код побудови та відображення діаграми, а й парсер мови програмування, що полегшить та пришвидшить процес інтеграції.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Myler H. R. Fundamentals of Engineering Programming with C and FORTRAN / Harley Myler. – USA: Cambridge University Press, 1998. – 223 с.

2. Graphviz documentation [Електронний ресурс] – Режим доступу до ресурсу: https://graphviz.org/documentation

3. About Mermaid [Електронний ресурс] – Режим доступу до ресурсу: https://mermaid-js.github.io/mermaid

4. PlantUML Language Reference Guide [Електронний ресурс] – Режим доступу до ресурсу: http://plantuml.com/en/guide

5. Fortran 77 Flowcharting Utility [Електронний ресурс] – Режим доступу до ресурсу: http://www.deater.net/weave/vmwprod/f77_diagram

6. Code2flow [Електронний ресурс] – Режим доступу до ресурсу: https://code2flow.com/

7. Visustin v8 Flow chart generator [Електронний ресурс] – Режим доступу до ресурсу: https://www.aivosto.com/visustin.html

8. Code Visual to Flowchart [Електронний ресурс] – Режим доступу до ресурсу: http://www.fatesoft.com/s2f/

9. Flow Chart Generation-Based Source Code Similarity Detection Using Process Mining / Z.Feng, L. Li, C. Liu, Q. Zeng. // Scientific Programming. – 2020. – №2020. – С. 1–15.

10. Stevens W. Structured design / W. Stevens, G. Myers, L. Constantine. // IBM Systems Journal. – 1974. – №13. – С. 115–139.

11. Refactorings for Fortran and high-performance computing. // SE-HPCS ’05. – 2005. – С. 37–39.

12. Metcalf M. Fortran 90/95 explained / M. Metcalf, J. K. Reid. – USA: Oxford University Press, Inc., 1999. – 341 с. – (2).

13. Photran Developer's Guide [Електронний ресурс] – Режим доступу до ресурсу: https://git.eclipse.org/c/ptp/org.eclipse.photran.git/plain/org.eclipse.photran-dev-docs/dev-guide/dev-guide-specialized.pdf.

14. TIOBE Index for April 2021 [Електронний ресурс] – Режим доступу до ресурсу: https://www.tiobe.com/tiobe-index/

15. Home repository for .NET Core [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/dotnet/core

16. Benatallah B. Web service conversation modeling: A cornerstone for E-business automation / B. Benatallah, F. Casati, F. Toumani. // Internet Computing, IEEE. – 2004. – №8. – С. 46 – 54.

17. Design Patterns: Elements of Reusable Object-Oriented Software / E.Gamma, R. Helm, R. Johnson, J. M. Vlissides., 1994. – 431 с.

18. Research and Application of Code Automatic Generation Algorithm Based on Structured Flowchart / X.Wu, M. Qu, Z. Liu, J. Li. // Journal of Software Engineering and Applications. – 2011. – №4. – С. 534–545.

19. Leatongkam A. Towards Extraction of UML Sequence Diagrams from Modern Fortran / A. Leatongkam, A. Nanthaamornphong. // IT Journal. – 2019. – №15. – С. 8–20.

20. Extracting UML Class Diagrams from Object-Oriented Fortran: ForUML / A.Nanthaamornphong, J. Carver, K. Morris, S. Filippone. // Scientific Programming. – 2015. – №2015. – С. 1–15.

21. Automatic Conversion of Structured Flowcharts into Problem Analysis Diagram for Generation of Codes / X.Wu, M. Qu, Z. Liu, J. Li. // Journal of Software. – 2012. – №7. – С. 1109–1120.

22. Program constructs [Електронний ресурс] – Режим доступу до ресурсу: https://cgi.csc.liv.ac.uk/~frans/OldLectures/2CS45/progCons/progCons.html