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

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»
УДК 004.93

«До захисту допущено»

Завідувач кафедри СПСКС

_____ В. П. Тарасенко
(підпис) (ініціали, прізвище)

"___"_____2018р.

Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123. "Комп'ютерна інженерія"
Системне програмування

на тему

СИСТЕМА РОЗПІЗНАВАННЯ ВИКОРИСТАННЯ ШАБЛОНІВ ПРОЕКТУВАННЯ В ПРОГРАМАХ

Виконав: студент 6 курсу, групи КВ-62м

Лиман Дмитро Миколайович _____
(підпис)

Науковий керівник доц., к.т.н. Марченко О. І. _____
(підпис)

Рецензент_____ _____
(підпис)

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

Київ – 2018

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

Факультет прикладної математики

Кафедра системного програмування і спеціалізованих комп'ютерних систем

Рівень вищої освіти – другий (магістерський)
Спеціальність 123. "Комп'ютерна інженерія"
Системне програмування

ЗАТВЕРДЖУЮ
Завідувач кафедри СПСКС
_____ _В.П.Тарасенко__
(підпис) (ініціали, прізвище)
«___»_____2018 р.

ЗАВДАННЯ
на магістерську дисертацію студенту

Лиман Дмитро Миколайович

1. Тема дисертації: СИСТЕМА РОЗПІЗНАВАННЯ ВИКОРИСТАННЯ ШАБЛОНІВ ПРОЕКТУВАННЯ В ПРОГРАМАХ,
науковий керівник дисертації: Марченко Олександр Іванович, к.т.н., доцент,
затверджені наказом по університету від «22» березня 2018 р. №986-с

2. Термін подання студентом дисертації 11 травня 2018 р.

3. Об'єкт дослідження: процес визначення використання шаблонів проектування у вихідному коді програм.

4. Предмет дослідження: способи визначення використання шаблонів проектування у вихідному коді програм на основі деревовидних структур даних.

5. Перелік завдань:
- провести аналіз існуючих систем для розпізнавання використання шаблонів проектування;
- розробити систему для розпізнавання використання шаблонів проектування з модульною архітектурою;
- розробити модифікований спосіб розпізнвання подібності дерев;
- проаналізувати ефективність розробленого модифікованого способу та інших існуючих способів за часом виконання.

6. Перелік ілюстративного матеріалу

- порівняльна таблиця існуючих інструментів для розпізнавання використання шаблонів проектування;
- схема використання структур даних у розробленому модифікованому способі;
- псевдокод модифікованого способу;
- графік залежності ширини дерева пошуку подібних дерев від часу виконання для модифікованого способу перебору, способу Ульмана та способу Чунга;
- графік залежності глибини дерева шаблону подібного дерева від часу виконання для модифікованого способу перебору, способу Ульмана та способу Чунга;
- графік залежності ширини дерева шаблону подібного дерева від часу виконання за умови відсутності подібності для модифікованого способу перебору, способу Ульмана та способу Чунга;

7. Перелік публікацій

- Стаття “Модифікований спосіб повного перебору для визначення відповідності дерев”
- Тези доповіді “Система визначення використання шаблонів проектування в програмах”

8. Дата видачі завдання 23 вересня 2016 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання	Примітка
1.	Ґрунтовне ознайомлення з предметною галуззю	15.12.2016	
2.	Визначення структури магістерської дисертації; вивчення літератури, пошук додаткової літератури, патентний пошук	01.03.2017	
3.	Робота над першим розділом магістерської дисертації; проведення наукового дослідження	15.05.2017	
4.	Проведення наукового дослідження; робота над другим розділом магістерської дисертації	15.10.2017	
5.	Проведення наукового дослідження; робота над третім розділом магістерської дисертації; розроблення програмного забезпечення	15.12.2017	
6.	Проведення наукового дослідження; робота над четвертим розділом магістерської дисертації; підготовка матеріалів доповіді на конференції ПМК-2018	01.03.2018	
7.	Завершення роботи над основною частиною магістерської дисертації; підготовка ілюстративного матеріалу	10.04.2018	
8.	Оформлення текстової і графічної частини магістерської дисертації	25.04.2018	
9.	Попередній розгляд магістерської дисертації	26.04.2018	

Студент _____ _____
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ _____
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Актуальність теми. Із розвитком комп'ютерної інженерії у інших сферах життя людини, зростає кількість та розмір розроблених систем. Із плином часу, створена система підтримується та розширюється за рахунок нових функціональних вимог до неї різними людьми або навіть компаніями. Це впливає на характеристики програмного забезпечення за різними показниками: кількість рядків вихідного коду, цикломатична складність, метрики Халстеда та інші. Для впорядкування вихідного коду та для вирішення типових задач при розробці використовують шаблони проектування. Такий підхід дозволяє прискорити розуміння зв'язаності частин коду за типовим використанням зв'язків спадкування та реалізації інженерам, що тільки підтримують розроблені системи та не приймали участі у створені. Тому створення засобів автоматизованого виявлення шаблонів проектування у існуючому вихідному є актуальним.

Об’єктом дослідження є процес визначення використання шаблонів проектування у вихідному коді програм.

Предметом дослідження є способи визначення використання шаблонів проектування у вихідному коді програм на основі деревовидних структур даних.

Мета роботи: прискорення процесу визначення наявності шаблонів проектування у програмах, розробка більш швидкого способу визначення наявності шаблонів проектування у програмах, ніж спосіб повного перебору.

Наукова новизна:

1. Проаналізовано існуючі системи та способи визначення використання шаблонів проектування у вихідному коді програм і показано, що ці системи мають недоліки у їх використанні за різними показниками: застарілість або відсутність оновлень, прив'язаність до конкретних версій компілятора та мови програмування, необхідність вивчення проміжної мови для опису шаблонів проектування.

2. Запропоновано модифікований спосіб повного перебору для визначення відповідності дерев, який відрізняється від стандартного тим, що відповідність вершин двох дерев визначається на основі часткового обходу дерев та побудови словника повної або часткової узгодженості їх вершин.

3. Виконано порівняльний аналіз модифікованого способу повного перебору з існуючими аналогами для вирішення задачі виявлення використання шаблонів проектування у вихідному коді мовою програмування Java і наведені приклади використання при яких модифікований спосіб має вищі та нижчі показники у швидкості роботи системи, використання ресурсів комп'ютерної системи порівняно з існуючими способами.

Практична цінність отриманих в роботі результатів полягає в тому, що розроблений модифікований спосіб дозволяє прискорити процес визначення використання шаблонів проектування у вихідному коді програм у випадках, коли шуканий шаблон має більшу висоту або ширину за середні аналогічні показники у програмній системі, в якій виявляються шаблони проектування, за рахунок поступового порівняння дерев за ярусами та перевірки умов для дострокового закінчення процесу виявлення шаблону в коді програми. Крім того, запропонований спосіб може бути використаний у різних сферах для визначення відповідності деяких моделей, за умови, що ці моделі можуть бути представлені у вигляді дерева. Розроблена в роботі програмна реалізація системи для виявлення шаблонів проектування може бути використана для автоматизованого або напівавтоматизованого рефакторингу вихідного коду.

Апробація роботи. Система для визначення шаблонів проектування у програмах була представлена та обговорювалась на науковій конференції магістрантів та аспірантів “Прикладна математика та комп’ютинг” ПМК-2018 (Київ, 21-23 березня 2018 р.). Модифікований спосіб повного перебору опублікований у науковому фаховому виданні

“Комп’ютерно-інтегровані технології: освіта, наука, виробництво” №28-29, що реферується наукометричною базою РІНЦ (Російський індекс наукового цитування), а також індексується в міжнародних базах даних Index Copernicus Journal Master List, Open Academic Journals Index, Academic Resource Index ResearchBib, Rootindexing, Information Matrix for the Analysis of Journals.

Структура та обсяг роботи. Магістерська дисертація складається з вступу, чотирьох розділів та висновків.

У вступі подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи.

У першому розділі розглянуто існуючі системи для визначення використання шаблонів проектування у програмах, їхні особливості, недоліки та переваги, розглянуто різні реалізації.

У другому розділі розглянуто типи шаблонів проектування, способи їх виявлення, наведено приклади таких способів. Запропоновано модифікований спосіб повного перебору для визначення ізоморфності дерев.

У третьому розділі наведено особливості реалізації розробленої системи.

У четвертому розділі представлено підходи до тестування системи в цілому та окремих модулів.

У висновках представлені результати проведеної роботи.

Робота представлена на 80 аркушах, містить посилання на список використаних літературних джерел.

Ключові слова: виявлення шаблонів проектування, дерева, граф, подібність дерев, ізоморфність.

ABSTRACT

Topicality. The number and size of developed systems grow during the development of computer engineering in different areas of human life. Over time, an established system is supported and expanded with new functional requirements. As a result, different people or even companies develop and extend old systems. This affects the characteristics of the software on various indicators: the number of lines of source code, cyclomatic complexity, Halsted metrics, and others. To organize the source code and to solve typical tasks, designing templates are used for design. This approach allows you to accelerate the understanding of the connectivity of code parts by typically using class inheritance or interface implementing. In such case, engineers that just started supporting the system are able to understand its structure easier. Therefore, the creation of automated detection of design templates in the existing source is an important task.

Object of research is the process of determining the use of design patterns in the source code of the programs.

Subject of research is techniques for determining the use of design patterns in the source code of applications based on tree-like data structures.

Research objective: accelerating the process of determining the presence of design patterns in the programs by developing a faster way to determine the presence of design patterns in the programs than the brute force method.

Scientific novelty:

1. The existing systems and techniques of determining the use of design patterns in the source code of the programs are analyzed and disadvantages of these systems are shown: obsolete or lack of updates, dependence on specific versions of the compiler and the programming language, the need to study the intermediate language to describe design patterns.

2. Proposed a modified technique for the complete selection for determining trees conformity. The technique differs from the brute force in matching approach. The matching of vertices of two input trees is determined on the basis of partial tree crawling and constructing a dictionary of the complete or partial coherence of

their vertices.

3. A comparative analysis of the modified technique with existing analogues has been made. Each technique solves the problem of identifying design patterns usage in the source code in the Java programming language. Examples of use cases with higher and lower resource usage are presented in comparative analysis with existing techniques too.

Practical value of the results obtained in the work is that the developed modified technique allows to accelerate the process of determining the use of design patterns in the source code of applications in cases where template tree-like structure has a higher height or width than the average similar indicators in the software system, which reveals design patterns, at the expense of gradual tree-level comparison and verification of the conditions for the early completion of the template detection process in the program code. In addition, the proposed technique can be used in various spheres to determine the conformity of some models, provided that these models can be presented as a tree. The program implementation of the system for revealing design patterns developed in the work can be used for automated or semi-automated refactoring of the source code.

Testing of work. The main provisions and results were presented and discussed at a scientific conference undergraduates and graduate students “Applied mathematics and computing”, AMC-2018 (Kyiv, March 21-23, 2018). The modified technique is published in the scientific publication “Computer-Integrated Technologies: Education, Science, Production” No. 28-29, which is referenced by the science-centered RSCI (Russian Scientific Citation Index), as well as indexed in the international databases of the Index Copernicus Journal Master List, Open Academic Journals Index, Academic Resource Index, ResearchBib, Rootindexing, Information Matrix for the Analysis of Journals.

The structure and scope of work. The master's thesis consists of an introduction, four chapters and conclusions.

The introduction gives a general description of the work, assesses the current state of the problem, substantiates the relevance of the research direction,

formulates the purpose and objectives of the research, shows the scientific novelty of the results obtained and the practical value of the work.

The first section examines the existing systems for determining the use of design patterns in programs, their features, drawbacks and advantages, and discusses different implementations.

The second section deals with the types of design templates, ways of their detection, examples of such techniques are given. There is proposed the modified method of brute force for determining isomorphic trees.

The third section presents the peculiarities of the implementation of the developed system.

The fourth section presents approaches to testing the system as a whole and individual modules.

In conclusion, the findings of the work.

Work submitted 80 pages, containing a link to a list of used literature.

Keywords: detection of design patterns, trees, graph, tree similarity, isomorphic.

РЕФЕРАТ

Актуальность темы. С развитием компьютерной инженерии в других сферах жизни человека, растет количество и размер разработанных систем. С течением времени, созданная система поддерживается и расширяется за счет нового функционала разными людьми или даже компаниями. Это влияет на характеристики программного обеспечения по различным показателям: количество строк исходного кода, цикломатическая сложность, метрики Халстеда и другие. Для упорядочения исходного кода и для решения типичных задач при разработке используют шаблоны проектирования. Такой подход позволяет ускорить понимание связанности частей кода по использованию связей наследования и реализации инженерам, которые поддерживают разработанные системы и не принимали участия в их создании. Поэтому создание средств автоматизированного распознования шаблонов проектирования в существующем исходном является актуальным.

Объектом исследования является процесс определения использования шаблонов проектирования в исходном коде программ.

Предметом исследования являются способы определения использования шаблонов проектирования в исходном коде программ на основе древовидных структур данных.

Цель работы: ускорение процесса определения наличия шаблонов проектирования в программах, разработка более быстрого способа определения наличия шаблонов проектирования в программах, чем способ полного перебора.

Научная новизна:

1. Проанализированы существующие системы и способы определения использования шаблонов проектирования в исходном коде программ и показано, что эти системы имеют недостатки в их использовании по разным показателям: устарелость или отсутствие обновлений, привязанность к конкретным версиям компилятора и языка программирования, необходимость изучения промежуточного языка для описания шаблонов проектирования.

2. Предложен модифицированный способ полного перебора для определения соответствия деревьев, который отличается от стандартного тем, что соответствие вершин двух деревьев определяется на основе частичного обхода деревьев и построения словаря полной или частичной согласованности их вершин.

3. Выполнен сравнительный анализ модифицированного способа полного перебора с существующими аналогами для решения задачи обнаружения использования шаблонов проектирования в исходном коде на языке программирования Java и приведены примеры использования при которых модифицированный способ имеет более высокие и более низкие показатели в скорости работы системы, использование ресурсов компьютерной системы по сравнению с существующими способами.

Практическая ценность полученных в работе результатов заключается в том, что разработан модифицированный способ позволяет ускорить процесс определения использования шаблонов проектирования в исходном коде программ в случаях, когда искомый шаблон имеет большую высоту или ширину чем средние аналогичные показатели в программной системе, в которой ищутся шаблоны проектирования, за счет постепенного сравнения деревьев по ярусами и проверки условий для досрочного окончания процесса выявления шаблона в коде программы. Кроме того, предложенный способ может быть использован в различных сферах для определения соответствия некоторых моделей, при условии, что эти модели могут быть представлены в виде дерева. Разработанная в работе программная реализация системы для выявления шаблонов проектирования может быть использована для автоматизированного или полуавтоматизированного рефакторинга исходного кода.

Апробация работы. Система для определения шаблонов проектирования в программах была представлена и обсуждалась на научной конференции магистрантов и аспирантов “Прикладная математика и компьютинг” ПМК-2018 (Киев, 21-23 марта 2018). Модифицированный

способ полного перебора опубликован в научном профессиональном издании “Компьютерно-интегрированные технологии: образование, наука, производство” №28-29, что реферируется наукометрическими базой РИНЦ (Российский индекс научного цитирования), а также индексируется в международных базах данных Index Copernicus Journal Master List, Open Academic Journals Index, Academic Resource Index ResearchBib, Rootindexing, Information Matrix for the Analysis of Journals.

Структура и объем работы. Магистерская диссертация состоит из введения, четырех глав и выводов.

Во введении представлена общая характеристика работы, произведена оценка современного состояния проблемы, обоснована актуальность направления исследований, сформулированы цели и задачи исследований, показано научную новизну полученных результатов и практическую ценность работы.

В первом разделе рассмотрены существующие системы для определения использования шаблонов проектирования в программах, их особенности, недостатки и преимущества, рассмотрены различные реализации.

Во втором разделе рассмотрены типы шаблонов проектирования, способы их выявления, приведены примеры таких способов. Представлен модифицированый метод полного перебора для определения изоморфности деревьев.

В третьем разделе приведены особенности реализации разработанной системы.

В четвертом разделе представлены подходы к тестированию системы в целом и отдельных модулей.

В выводах представлены результаты проведенной работы.

Работа представлена на 80 листах, содержит ссылки на список использованных литературных источников.

Ключевые слова: выявление шаблонов проектирования, дерево, граф, сходство деревьев, изоморфность.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ.....3
ВСТУП.....4
1.АНАЛІЗ СПОСОБІВ РЕФАКТОРИНГУ ТА МЕТОДІВ ВИЯВЛЕННЯ ШАБЛОНУ ПРОЕКТУВАННЯ.....5
1.1.Аналіз способів рефакторингу програмного забезпечення.....5
1.2.Шаблони проектування та їх представлення.....10
1.3.Аналіз існуючих засобів виявлення шаблонів проектування у вихідному коді.....23
2.МОДИФІКОВАНИЙ СПОСІБ РОЗПІЗНАВАННЯ ПОДІБНОСТІ ДЕРЕВ 27
2.1.Використання синтаксичного дерева для розпізнавання шаблонів проектування.....27
2.2.Задача розпізнавання подібності дерев.....29
2.3.Модифікований спосіб повного перебору для розпізнавання відповідності дерев.....41
3.СТРУКТУРА СИСТЕМИ ДЛЯ ВИЯВЛЕННЯ ШАБЛОНІВ ПРОЕКТУВАННЯ ТА ЇЇ ТЕСТУВАННЯ.....45
3.1.Контекстно-вільна граматика мови Java.....45
3.2.Синтаксичний аналізатор та побудова дерева розбору.....47
3.3.Побудова графів зв'язків між класами для вихідного коду.....48
3.4.Інтерфейс користувача.....51
3.5.Підхід до тестування системи.....54
3.6.Тестування системи.....55
4.ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕФЕКТИВНОСТІ СПОСОБІВ.....60
4.1. Задачі для тестування.....60
4.2.Порівняння роботи способів за часом виконання.....63
4.3.Порівняння часу роботи для синтетичних та реальних вхідних

даних.....76
ВИСНОВКИ.....79
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....80
Додаток А. Алгоритм Ульмана без процедури оптимізації.....82
Додаток Б. Порівняльна таблиця існуючих інструментів для розпізнавання використання шаблонів проектування.....84
Додаток В. Схема використання структур даних у розробленому модифікованому способі.....85
Додаток Г. Псевдокод модифікованого способу.....86
Додаток Д. Графік залежності ширини дерева пошуку подібних дерев від часу виконання для модифікованого способу перебору, способу Ульмана та способу Чунга.....87
Додаток Е. Графік залежності глибини дерева шаблону подібного дерева від часу виконання для модифікованого способу перебору, способу Ульмана та способу Чунга.....88
Додаток Є. Графік залежності ширини дерева шаблону подібного дерева від часу виконання за умови відсутності подібності для модифікованого способу перебору, способу Ульмана та способу Чунга.....89

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

АСД — абстрактне синтаксичне дерево

КВГ — контекстно-вільна граматика

ТС — таблиця символів

FUJABA — from UML to Java and back again

GoF — Gang of Four

JPINOT — javac Pattern INference recOvery Tool

MUSCAT — minimal UML SpecifiCATion language

PINOT — pattern INference recOvery Tool

UML — Unified Modeling Language

WOP — WEB OF PATTERNS

ВСТУП

Процеси перетворення програмного коду, зміни внутрішньої структури програмного забезпечення для полегшення розуміння коду і легшого внесення подальших змін без зміни зовнішньої поведінки самої системи є нагальними проблемами у сфері рефакторингу вихідного коду. Наслідком зміни внутрішньої структури програмного коду може бути зменшення витрат часу необхідних для додавання нових функціональних можливостей або виправлення помилок в існуючому коді.

Складність виконання такої задачі полягає в тому, що виконувати перетворення неможливо без розуміння структури та мети створення конкретних модулів системи, що піддаються рефакторингу. Для полегшення розуміння укладу сукупності елементів у виділеній частині проекту та їх взаємозв'язків між собою часто використовують методи візуалізації. У якості елементів виступають структури даних, що використовуються мовою програмування, або відбувається перетворення до узагальнених елементів.

Незважаючи на велику кількість засобів для полегшення процесу зазначеного перетворення вихідного коду, відсутність в них достатньої автоматизованості спричиняє великі трудові затрати, тому існує необхідність створення нових способів для вирішення задач автоматизованого рефакторингу.

Однією з таких задач є автоматизація процесу зміни структури внутрішнього коду, що використовує шаблони проектування. Дана магістерська дисертації присвячена рішенню задачі прискорення процесу визначення наявності шаблонів проектування у програмах та розробці більш швидкого способу визначення наявності шаблонів проектування у програмах, ніж спосіб повного перебору.

1. АНАЛІЗ СПОСОБІВ РЕФАКТОРИНГУ ТА МЕТОДІВ ВИЯВЛЕННЯ ШАБЛОНУ ПРОЕКТУВАННЯ

1.1. Аналіз способів рефакторингу програмного забезпечення

Рефакторинг – це процес такої зміни програмної системи, за якої не змінюється зовнішня поведінка коду, але покращується його внутрішня структура. Це спосіб систематичного впорядкування програми, при якому шанси появи нових помилок є мінімальними. При проведенні рефакторингу коду покращується дизайн вже готової програми [1].

Покращення коду після його написання може виявитись дещо дивним, оскільки в сьогоднішньому розумінні розробки програмного забезпечення спочатку робиться дизайн системи, а потім пишеться код. З часом програма модифікується, і цілісність системи, відповідність її структури початковому плануванню поступово погіршується.

За допомогою рефакторингу можна взяти проблемну програму і переробити її в добре спроектовану. Кожен крок цього процесу є надзвичайно простим. Переміщається поле з одного класу в інший, забирається частина коду з методу і поміщається в окремий метод, якийсь код переміщується по ієрархії в різних напрямках. Але сумарний ефект цих невеликих змін може радикально покращити проект. Це є протилежністю звичайному процесу розпаду програми.

При проведенні рефакторингу виявляється, що відношення різних етапів роботи змінюється. Процес проектування розтягнуто на весь період розробки, а не сконцентровано лише на її початковому етапі. В процесі роботи над проектом стає зрозумілим, як його можна покращити. Така взаємодія призводить до створення програми, якість програмного коду якої залишається високою під час доповнення та розвитку продукту.

Основна ціль рефакторингу – зробити код простішим і зрозумілішим. Причому, рефакторинг і оптимізація це далеко не одне і те ж. В результаті оптимізації код працює швидше, але не обов’язково є простішим і зрозумілішим. Рефакторинг – це спрощення і покращення розуміння коду розробниками, які будуть підтримувати поточний функціонал та розширювати його. Рефакторинг можна застосовувати до будь-якої програми. Частіше за все це робиться інтуїтивно. Якщо якийсь код є не дуже вдалим, або через певний період часу незрозуміло, що робить та чи інша частина програми, – потрібно здійснити рефакторинг. Можна виокремити деякі ознаки коду, над яким необхідно виконати перетворення для зниження складності за метриками. Типовими метриками для оцінювання коду у числових показниках є [2]:

- порядок зростання (мається на увазі аналіз алгоритмів, в термінах теорії складності обчислень);
- кількість рядків коду;
- цикломатична складність;
- кількість помилок на рядок коду;
- ступінь покриття коду тестуванням;
- покриття вимог;
- кількість класів та інтерфейсів;
- зв'язність;
- пов'язаність;
- розмір бінарних файлів.

Дублювання коду - одна з найпоширеніших причин для виконання рефакторингу. Код, який дублюється в програмі, є основним джерелом помилок. Якщо якась дія виконується у декількох різних місцях, але однаковою послідовністю команд, то цей код потрібно винести в окрему функцію і замінити старі місця на виклики нової функції. Інакше висока

ймовірність того, що при виправленні помилки в одному місці копії команд залишаться не відредагованими, і у них залишиться помилка.

Як правило, людина не може повністю сприймати і оцінювати правильність коду, якщо він займає більше ніж 2-3 десятки рядків. Такі методи і функції потрібно розбивати на декілька дрібніших і робити одну загальну функцію, що буде послідовно викликати ці методи. Ніколи не варто скорочувати розмір функції, вписуючи декілька операторів в один рядок. Ймовірність допустити помилку в такому коді зростатиме.

Велика кількість параметрів зазвичай не тільки ускладнює розуміння того, що робить даний метод чи функція, але і ускладнює розуміння коду, що використовує цю функцію. Якщо ж дійсно є потреба у великій кількості параметрів функції, то їх необхідно винести в окрему структуру або клас і передавати вказівник чи посилання на об’єкт цієї структури чи класу.

Якщо у розробленому програмному забезпеченні є один або декілька великих класів, то їх необхідно розділити на менші, та включити об’єкти цих класів в один загальний клас.

Велика кількість тимчасових змінних також є ознакою проблемного коду, якому необхідний рефакторинг. Як правило, багато тимчасових змінних зустрічаються у занадто великих функціях, після рефакторингу яких, кількість змінних в них стане меншою і код стане значно зрозумілішим і зручнішим для подальшої модифікації.

Велика кількість звертань об’єктів одного класу до даних об’єкта іншого класу вказує на високу зв'язність структур даних. Потрібно переглянути функціонал об’єктів оскільки можливо, було прийняте невірне архітектурне рішення, і його потрібно поміняти якомога раніше, щоб помилка не поширилась по всьому коду.

Не доцільно створювати занадто багато глобальних змінних. Якщо ж така необхідність є, то потрібно спробувати згрупувати їх в структури, класи, або хоча б винести їх в окремий простір імен. Тоді шанс

помилкового використання якоїсь змінної стане значно нижчим.

Основними стимулами для проведення рефакторингу є [2]:

- додавання нової функції, яка недостатньо вкладається в прийняте архітектурне рішення;
- виправлення помилки, причини виникнення якої не є очевидними;
- подолання труднощів командної розробки, які зумовлені складною логікою програми.

Основними прийомами рефакторингу, що дозволяють поділити код на менші та зрозуміліші частини, є:

- відокремлення методу (Extract Method);
- відокремлення базового класу (Extract Superclass).

Основними прийомами рефакторингу, що дозволяють забезпечити додаткову абстракцію, є:

- інкапсуляція поля (Encapsulate Field) — замінює прямий доступ до поля на доступ через методи, що надають доступ для читання та запису;
- узагальнення типу (Generalize Type) — заміна типів, з якими працює клас, на більш узагальнені;
- заміна блоків перевірки типів на шаблони «Стан» (State) або «Стратегія» (Strategy);
- заміна умовних операторів поліморфізмом ⊏;
- створення поля або локальної змінної (Introduce Field/Introduce Local Variable).

Основними прийомами рефакторингу, що змінюють назви членів та їх розташування, є:

- переміщення метода класу (Move Method) або переміщення поля класу в інші класи або файли коду;

- перейменування члена класу (Rename) — зміна імені, з автоматичною заміною всіх посилань на старе ім’я в коді;
- переміщення члена класу до базового/дочірнього класу (Pull Up/Push Down).

Ще одним способом є зміна архітектури проекту. Цей процес сильно залежить від методології керування проектами та проектування і тому така класифікація відсутня.

Рефакторинг вихідного коду можна поділити на два види за принципом виконання [1]:

- рефакторинг виконаний людиною або групою людей;
- автоматизований рефакторинг програмними засобами.

Переваги неавтоматизованого рефакторингу:

- дозволяє контрольованим способом перетворювати програмний код;
- вищий рівень якості зміни внутрішньої структури програмного забезпечення порівняно з автоматизованим рефакторингом;
- покращення розуміння коду для легшого внесення подальших змін без зміни зовнішньої поведінки самої системи.

Недоліки неавтоматизованого рефакторингу:

- тривалий час виконання;
- високий рівень складності при виконанні декомпозиції великого проекту;
- при залученні сторонніх осіб для рефакторингу можуть виникнути проблеми порушення конфіденційності створеного проекту.

Переваги перетворюються у недоліки та навпаки у випадку застосування автоматизованого процесу рефакторингу вихідного коду.

При поєднанні таких двох ідей, як зміна архітектури проекту та

використання шаблонів проектування, рефакторинг перетворюється на маніпуляції з шаблонами проектування.

Виділяють наступні види роботи з шаблонами у вихідному коді:

- додавання шаблонів для структуризації процесу;
- зміна структури шаблонів проектування для спрощення подальшої підтримки продукту та полегшення розуміння вихідного коду;
- повна відмова від шаблонів проектування. [3]

1.2. Шаблони проектування та їх представлення

У розробці програмного забезпечення шаблон проектування є загальним, багаторазовим рішенням для широко поширеної проблеми в заданому контексті при розробці програмного забезпечення. Це не закінчене рішення, яке можна перетворити безпосередньо в вихідний або машинний код. Це опис або шаблон для вирішення проблеми, яка може бути використана у багатьох різних ситуаціях. Шаблони проектування є формалізованими найкращими практиками, які програміст може використовувати для вирішення спільних проблем при розробці програми або системи.

Об'єктно-орієнтовані шаблони проектування, зазвичай демонструють взаємозв'язки та взаємодію між класами або об'єктами, не вказуючи застосування кінцевих класів програми або об'єктів, які беруть участь. Шаблони, що передбачають змінний стан, можуть бути неприйнятними для функціональних мов програмування. Деякі шаблони можуть бути винесені непотрібними в мовах, які мають вбудовану підтримку для вирішення проблеми, яку вони намагаються вирішити. Об'єктно-орієнтовані шаблони не завжди підходять для не об'єктно-орієнтованих мов програмування.

Шаблони проектування можуть розглядатися як структурований підхід

до проміжного етапу у програмуванні між рівнями парадигми програмування та конкретним алгоритмом.

Шаблони проектування, які ми знаємо сьогодні, були створені Еріком Ґаммою, Річардом Хелмом, Ральфом Джонсоном та Джоном Вліссідсом. Вони опублікували книгу «Design Patterns — Elements of Reusable Object-Oriented Software» [4]. У цій книзі описані 23 шаблони проектування. Також команда авторів цієї книги відома суспільству під назвою банда чотирьох (англ. Gang of Four - GoF). Саме ця книга послужила приводом до широкого поширення шаблонів проектування. З плином часу, додавалися нові шаблони проектування, які допомагали вирішувати інші типові архітектурні питання.

Класифікація шаблонів проектування за трьома типами була представлена у книзі банди чотирьох [4]:

- твірні (породжуючі) шаблони;
- структурні шаблони;
- шаблони поведінки.

Кожен із шаблонів проектування може бути формально описаний за допомогою структурної UML діаграми, а саме діаграми класів.

Діаграма класів — статичне представлення структури моделі. Відображає статичні (декларативні) елементи, такі як: класи, типи даних, їх зміст та відношення. Діаграма класів, також, може містити позначення для пакетів та може містити позначення для вкладених пакетів. Також, діаграма класів може містити позначення деяких елементів поведінки, однак їх динаміка розкривається в інших типах діаграм. Діаграма класів служить для представлення статичної структури моделі системи в термінології класів об'єктно-орієнтованого програмування. На цій діаграмі показують класи, інтерфейси, об'єкти, а також їх відносини [5].

Декларативні частини діаграми мають методи та атрибути, як складові

елементи. Для кожного типу елементу можна виділити дві характеристики: межі використання та видимість. Під межами використання розуміється використання методів або атрибутів: статичний, той що належить класу, або об'єктний, той що належить створеному об'єкту класу. Для задання видимості використовують позначення, що зображені на табл. 1.1. Одну з позначок необхідно вказати перед ім'ям методу або атрибуту.

Позначення	Пояснення позначення
+	публічний
-	приватний
#	захищений
/	наслідуваний
~	пакетний
*	випадковий

Таблиця 1.1. Позначки для вказання видимості елементів діаграми класів

Існує декілька видів зв'язків між елементами діаграми класів. Їх можна поділити на два типи за елементами між якими вони існують: зв'язок на рівні об'єктів та зв'язок на рівні класів або інтерфейсів. Нижче наведено список можливих зв'язків між елементами на діаграмі класів.

Відношення залежності в загальному випадку вказує деяке семантичне відношення між двома елементами моделі або двома множинами таких елементів, яке не є відношенням асоціації, узагальнення або реалізації. Воно стосується тільки самих елементів моделі і не потребує множини окремих прикладів для пояснення свого змісту. Відношення залежності використовується в такій ситуації, коли деяка зміна одного елемента моделі

може потребувати зміну іншого залежного від неї елемента моделі. [5]

Відношення залежності графічно зображається пунктирною лінією між відповідними елементами зі стрілкою на одному з її кінців. На діаграмі класів дане відношення зв'язує окремі класи між собою, при цьому стрілка направлена від класу-клієнта залежності до незалежного класу або класу-джерела. [5]

В якості класу-клієнта і класу-джерела залежності можуть виступати цілі множини елементів моделі. У цьому випадку одна лінія зі стрілкою, що виходить від джерела залежності, розщеплюється в деякій точці на декілька окремих ліній, кожна з яких має окрему стрілку для класу-клієнта.

Стрілка може помічатися необов'язковим, але стандартним ключовим словом в лапках і необов'язковим індивідуальним іменем. Для відношення залежності визначені ключові слова, які позначають деякі спеціальні види залежностей. Ці ключові слова записуються в лапках поруч зі стрілкою, яка відповідає даній залежності. Прикладами стереотипів для відношення залежності можуть бути:

- «access» — служить для позначення доступності відкритих атрибутів і операцій класу-джерела для класів-клієнтів;
- «bind» — клас-клієнт може використовувати деякий шаблон для своєї наступної параметризації;
- «derive» — атрибути класу-клієнта можуть бути обчислені за атрибутам класу-джерела;
- «import» — відкриті атрибути і операції класу-джерела стають частиною класу-клієнта, так як би вони були оголошені безпосередньо в ньому;
- «refine» — вказує, що клас-клієнт служить уточненням класу-джерела з історичних причин, коли з'являється додаткова інформація в ході роботи над проектом. [5]

Відношення асоціації відповідає наявності деякого відношення між класами. Дане відношення позначається суцільною лінією з додатковими спеціальними символами, які характеризують окремі властивості конкретної асоціації. Як додаткові спеціальні символи можуть використовуватися ім'я асоціації, а також імена і кратність класів-ролей асоціації. Ім'я асоціації є необов'язковим елементом її позначення. Якщо воно задане, то записується з великої букви поруч з лінією відповідної асоціації. [5]

Найбільш простий випадок даного відношення — бінарна асоціація. Вона зв'язує два класи і, як виняток, може зв'язувати клас з самим собою. Для бінарною асоціації на діаграмі може бути вказано порядок слідування класів з використанням трикутника в формі стрілки поруч з іменем даної асоціації. Напрямок цієї стрілки вказує на порядок класів, один із яких є першим (зі сторони трикутника), а інший — другим (зі сторони вершини трикутника). Відсутність даної стрілки поруч з іменем асоціації означає, що порядок слідування класів у цьому відношенні не визначений.

Тернарна асоціація і асоціації з більшою кількістю зв'язуваних класів, в загальному випадку називаються N-арною асоціацією. Така асоціація зв'язує деяким відношенням 3 і більше класів, при цьому один клас може брати участь в асоціації більше ніж один раз. Клас асоціації має певну роль у відповідному відношенні, що може бути явно вказано на діаграмі. Кожен екземпляр N-арної асоціації представляє собою N-арний кортеж значень об'єктів із відповідних класів. Бінарна асоціація є частковим випадком N-арної асоціації, коли N=2, і має своє власне позначення. [5]

N-арна асоціація графічно позначається ромбом, від якого ведуть лінії до символів класів даної асоціації. У цьому випадку ромб з'єднується з символами відповідних класів суцільними лініями.

Порядок класів в N-арній асоціації, на відміну від порядку множин у відношенні, на діаграмі не фіксується. Деякий клас може бути приєднаний

до ромба пунктирною лінією. Це означає, що даний клас забезпечує підтримку властивостей відповідної N-арної асоціації, а сама N-арна асоціація має атрибути, операції, асоціації. Іншими словами, така асоціація, у свою чергу, є класом з відповідним позначенням у виді прямокутника і є самостійним елементом мови UML — асоціацією-класом (Association Class). [5]

Окремий клас асоціації має власну роль у відношенні. Ця роль може бути зображена графічно на діаграмі класів. Для цієї мети у мові UML вводиться спеціальний елемент — кінець асоціації (Association End), який графічно відповідає точці з'єднання лінії асоціації з окремим класом. Кінець асоціації є частиною асоціації, але не класу. Кожна асоціація має два або більше кінців асоціації. Найбільш важливі властивості асоціації вказуються на діаграмі поруч з цими елементами асоціації. [5]

Одним із таких додаткових позначень є ім'я ролі окремого класу, який входить в асоціацію. Ім'я ролі представляє собою стрічку тексту поруч з кінцем асоціації для відповідного класу. Вона вказує специфічну роль, яку відіграє клас, який є кінцем асоціації. Ім'я ролі не є обов'язковим елементом позначень.

Наступний елемент позначень — кратність окремих класів, які є кінцями асоціації. Кратність окремого класу позначається у виді інтервалу цілих чисел, аналогічно кратності атрибутів і операцій класів. Інтервал записується поруч з кінцем асоціації і для N-арної асоціації означає потенційне число окремих екземплярів або значень кортежів цієї асоціації, які можуть мати місце, коли інші N-1 екземплярів або значень класів фіксовані. [5]

Частковим випадком відношення асоціації є так звана виключна асоціація (Xor-association). Семантика даної асоціації вказує на той факт, що із декількох потенційно можливих варіантів даної асоціації в кожен момент часу може використовуватися тільки один її екземпляр. На діаграмі

класів виключна асоціація зображається пунктирною лінією, яка з'єднує дві і більше асоціації, поруч з якою записується стрічка-обмеження «{хог}».

Спеціальною формою або частковим випадком відношення асоціації є відношення агрегації, яке, в свою чергу, також має спеціальну форму — відношення композиції.

Відношення агрегації може бути між декількома класами у тому випадку, якщо один із класів представляє собою деяку сутність, що включає в себе як складові частини інші сутності. [5]

Дане відношення має фундаментальне значення для опису структури складених систем, оскільки застосовується для представлення системних взаємозв'язків типу «частина-ціле». Розкриваючи внутрішню структуру системи, відношення агрегації показує, з яких компонентів складається система і як вони зв'язані між собою. З точки зору моделі окремі частини системи можуть виступати як у виді елементів, так і у виді підсистем, які, у свою чергу, також можуть утворювати складові компоненти або підсистеми. Це відношення за своєю суттю описує декомпозицію або поділ складної системи на більш прості складові частини, які також можуть бути поділені, якщо у цьому виникне необхідність. [5]

Розглянутий поділ системи на складові частини представляє собою деяку ієрархію її компонентів, проте дана ієрархія принципово відрізняється від ієрархії, яка породжується відношенням узагальнення. Відмінність полягає у тому, що частини системи ніяк не зобов'язані успадковувати її властивості і поведінку, оскільки є самостійними сутностями. Більше того, частини цілого мають свої власні атрибути і операції, які суттєво відрізняються від атрибутів і операцій цілого.

Графічно відношення агрегації зображається суцільною лінією, один із кінців якої представляє собою незамальований ромб. Цей ромб вказує на той з класів, який представляє собою «ціле». Інші класи є його «частинами». [5]

Відношення композиції є частковим випадком відношення агрегації. Це відношення використовується для виділення спеціальної форми відношення «частина-ціле», при якому складові частини в деякому змісті знаходяться в середині цілого. Специфіка взаємозв'язків між ними полягає у тому, що частини не можуть виступати у відриві від цілого, тобто зі знищенням цілого знищуються і всі його складові частини.

Графічно відношення композиції зображається суцільною лінією, один із кінців якої представляє собою зафарбований ромб. Цей ромб вказує на той з класів, який представляє собою клас-композицію або «ціле». Інші класи є його «частинами». [5]

Для відношення композиції і агрегації можуть використовуватися додаткові позначення, які застосовуються для відношення асоціації. А саме, вказівка кратності класу асоціації і імені даної асоціації, які не є обов'язковими.

Відношення узагальнення є звичайним таксономічним відношенням між більш загальним елементом (батьком або предком) і більш частковим або спеціальним елементом (дочірнім або нащадком). Дане відношення може використовуватися для представлення взаємозв'язків між пакетами, класами, варіантами використання і іншими елементами мови. [5]

Стосовно до діаграм класів дане відношення описує ієрархічну будову класів і успадкування їх властивостей і поведінки. При цьому передбачається, що клас-нащадок має всі властивості і поведінку класу-предка, а також має свої власні властивості і поведінку. На діаграмах відношення узагальнення позначається суцільною лінією з трикутною стрілкою на одному з кінців. Стрілка вказує на більш загальний клас (клас-предок або суперклас), а її відсутність — на більш спеціальний клас (клас-нащадок або підклас).

Як правило, на діаграмі може вказуватися декілька ліній для одного відношення узагальнення, що зображає його таксономічний характер. У

цьому випадку більш загальний клас поділяється на підкласи одним відношенням узагальнення. [5]

З метою спрощення позначень на діаграмі класів сукупність ліній, які позначають одне й те ж відношення узагальнення, може об'єднуватися в одну лінію. У цьому випадку дані окремі лінії зображаються такими, що сходяться до єдиної стрілки.

Поруч з стрілкою узагальнення може розміщуватися текст, який вказує на деяку додаткову властивість цього відношення. Даний текст буде відноситися до всіх ліній узагальнення, які йдуть до класів-нащадків. Іншими словами, відмічена властивість стосується всіх підкласів даного відношення. При цьому текст потрібно розглядати як обмеження, і тоді він записується в фігурних дужках. [5]

Розроблена система використовує зв'язки діаграми класів для визначення залежності між елементами програмного забезпечення, що аналізується. Після отримання зв'язків діаграми класів, використовується модифікований спосіб розпізнавання подібності дерев для розпізнавання твірних шаблонів проектування. Доцільно навести опис шаблонів проектування, що належать лише до цього типу.

Породжуючі шаблони — це шаблони проектування, що абстрагують процес побудови об'єктів. Вони допоможуть зробити систему незалежною від способу створення, композиції та представлення її об'єктів. [6]

Шаблон, який породжує класи, використовує успадкування, щоб варіювати створюваний клас, а шаблон, що створює об'єкти, делегує конструювання іншому об'єктові. Ці шаблони важливі, коли система більше залежить від композиції об'єктів, ніж від успадкування класів. Таким чином, замість безпосереднього опису фіксованого набору поведінок, визначається невеликий набір фундаментальних поведінок, за допомогою композиції яких можна отримувати складніші. Таким чином, для створення об'єктів з конкретною поведінкою використовуються інші

механізми, у порівняння з простим інстанціюванням екземпляру класу. [6]

Твірні шаблони приховують інформацію про конкретні класи, які застосовуються у системі та деталі того, як ці класи створюються і стикуються між собою. Єдина інформація про об'єкти, що відома системі — їх інтерфейси. Нижче наведено породжуючі шаблони проектування.

Абстрактна фабрика (англ. Abstract Factory) — шаблон проектування, що забезпечує інкапсуляцію окремих фабрик під єдиною схемою, приховуючи їх деталізацію. Належить до класу твірних шаблонів.

В типових випадках застосування, розробник, що використовує готову абстрактну фабрику, створює її конкретну реалізацію, а потім використовує загальний універсальний інтерфейс фабрики, для створення екземплярів об'єктів, які є частиною схеми. Клієнтський вихідний код не знає, які саме об'єкти він отримує від цих фабрик, оскільки він використовує універсальний інтерфейс для їх створення. Шаблон розмежовує деталі реалізації множини об'єктів від їх загального використання в коді, оскільки створення об'єкта здійснюється за допомогою методів, що забезпечуються інтерфейсом фабрики.[6]

Типовими прикладами застосування шаблону проектування абстрактна фабрика є такі вимоги до системи:

- система не повинна залежати від того, як утворюються, компонуються та представляються вхідні до неї об'єкти;
- об'єкти, що входять до однієї групи за тематикою їх використання, повинні використовуватися разом і необхідно забезпечити виконання цього обмеження;
- налаштування системи повинно бути забезпечено однією з груп об'єктів, що входять до її складу;
- необхідно представити бібліотеку об'єктів, розкриваючи тільки їхні інтерфейси, але не реалізацію. [6]

Представлення шаблону проектування у вигляді діаграми класів наведено на рис. 1.1.

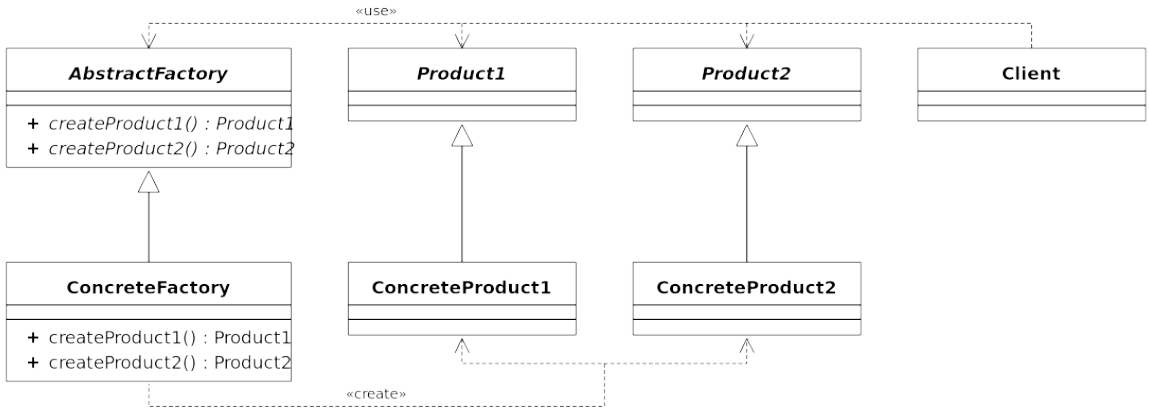


Рисунок 1.1. UML діаграма класів для абстрактної фабрики

Фабричний метод — твірний шаблон, який використовує фабричні методи для вирішення проблеми створення об'єктів, не маючи необхідності вказувати точний клас об'єкта, який буде створено. Це відбувається шляхом створення об'єктів за допомогою фабричного методу, який вказано в інтерфейсі і реалізованого дочірніми класами, або реалізованого в базовому класі, і можливо перевизначене дочірніми класами. Цей метод є альтернативою до безпосереднього виклику конструктора. [6]

Призначення цього шаблону є визначення інтерфейсу для створення об'єкта. Водночас фабричний клас залишає підкласам рішення про те, який саме клас інстанціювати. Фабричний метод дозволяє класу делегувати інстанціювання підкласам.

Слід використовувати шаблон коли:

- класу не відомо заздалегідь, об'єкти яких саме класів йому потрібно створювати;
- клас спроектовано так, щоб об'єкти, котрі він створює,

специфікувалися підкласами;

- клас делегує свої обов'язки одному з кількох допоміжних підкласів, та потрібно локалізувати знання про те, який саме підклас приймає ці обов'язки на себе. [6]

Для виявлення цього шаблону використовується UML діаграма класів, що зображена на рис. 1.2.

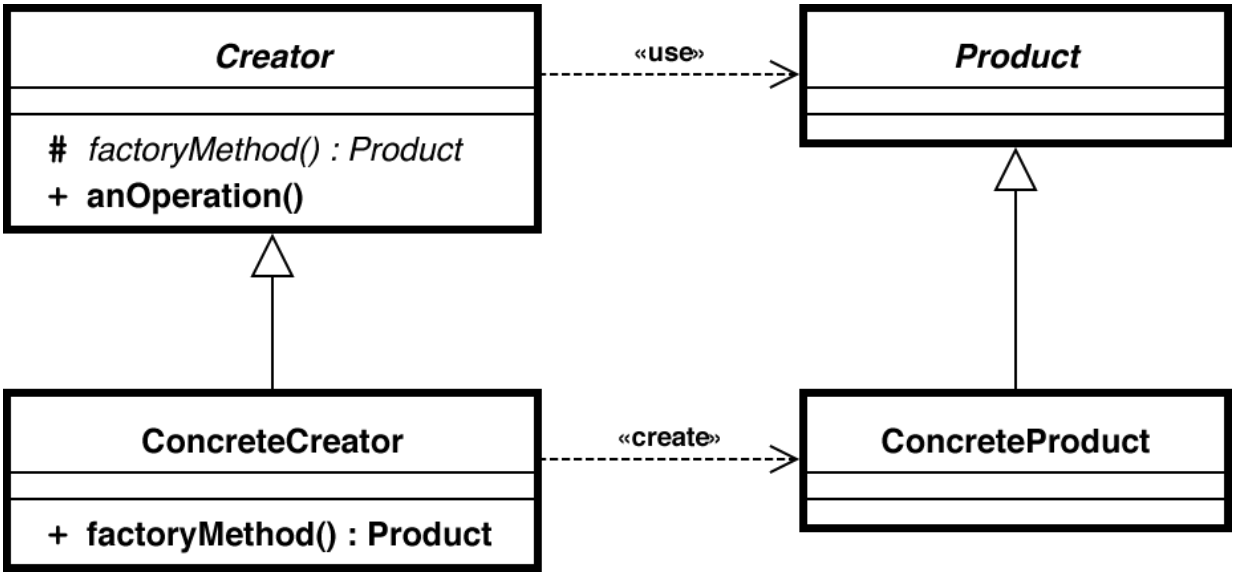


Рисунок 1.2. UML діаграма класів шаблону фабричний метод

На рис. 1.2 зображено декілька елементів:

- Product — продукт, який визначає інтерфейс об'єктів, що створюються фабричним методом.
- ConcreteProduct — конкретний продукт, який реалізує інтерфейс Product.
- Creator — творець оголошує фабричний метод, що повертає об'єкт класу, який реалізовує інтерфейс Product.
- ConcreteCreator — конкретний творець, який перевизначає фабричний метод, що повертає об'єкт ConcreteProduct.

Будівельник (англ. Builder) — шаблон проектування, відноситься до

класу твірних шаблонів. На відміну від шаблону абстрактної фабрики і фабричного методу, ціль яких є застосування поліморфізму, задачею шаблону будівельника є забезпечення реалізації антишаблону багатоступеневого конструювання. Антишаблон "телескопічний конструктор" коли різна комбінація параметрів конструктора призводить до появи експоненційної множини конструкторів. Замість того, щоб використовувати набір конструкторів, використовують шаблон будівельник, згідно якого використовують інший об'єкт (будівельник), що отримує на вхід параметри по одному, крок за кроком, і повертає за один прохід створений об'єкт. [6]

Шаблон проектування призначається для відокремлення конструювання складного об'єкта від його подання. Таким чином, у результаті одного й того ж процесу конструювання, можуть бути отримані різні подання.

Слід використовувати шаблон будівельник коли:

- алгоритм створення складного об'єкта не повинен залежати від того, з яких частин складається об'єкт та як вони стикуються поміж собою;
- процес конструювання повинен забезпечити різні подання об'єкта, що конструюється. [6]

Приклад UML діаграми класів для цього шаблону проектування наведена нижче на рис. 1.3.

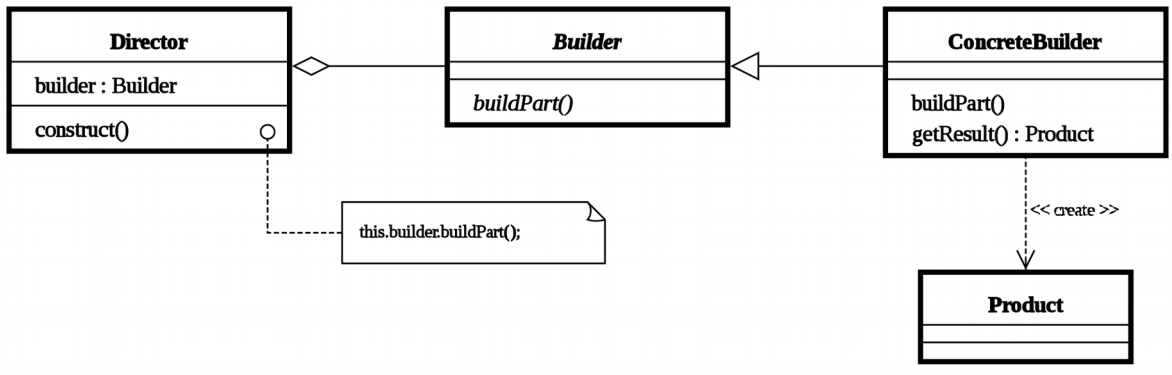


Рисунок 1.3. UML діаграма класів шаблону будівельник

Елементи діаграми класів на рис. 1.3 можуть бути пояснені таким чином:

- Builder — будівельник, який визначає абстрактний інтерфейс для створення частин об'єкта Product.
- ConcreteBuilder — конкретний будівельник, який конструює та збирає докупи частини продукту шляхом реалізації інтерфейсу Builder;
- Director — керівник, який конструює об'єкт, користуючись інтерфейсом Builder;
- Product — продукт, який подає складний конструйований об'єкт.

1.3. Аналіз існуючих засобів виявлення шаблонів проектування у вихідному коді

Для полегшення внесення змін у структуру шаблонів проектування на сьогоднішній день створено декілька інструментів для виявлення шаблонів проектування у вихідному коді.

Існує декілька інструментів для виявлення шаблонів проектування у вихідному коді мовою Java:

- PINOT (Pattern INference recOvery Tool)
- MUSCAT
- JPINOT
- FUJABA
- WOP

PINOT

Ця утиліта використовує Jikes (Java-компілятор написаний на C++ з відкритим вихідним кодом) з вбудованим аналізатором шаблонів та може розпізнати усі структурні та поведінкові шаблони проектування описані у книзі «Шаблони проектування: Елементи повторно використовуваного об'єктно-орієнтованого програмного забезпечення» спільного авторства Еріха Гамма, Річарда Хелма, Ральфа Джонсона, Джона Вліссідеса [4].

Переваги:

- результативність підтверджена тестами на проектах з відкритим вихідним кодом мовою Java: Java AWT 1.3, JHotDraw 6.0b1, Java Swing 1.4, java.io 1.4.2, java.net 1.4.2, javac 1.4.2, Apache Ant 1.6.2, ArgoUML 0.18.1;
- наявність розширень для збільшення можливостей та покращення якості роботи.

Недоліки:

- відсутність підтримки, останні зміни були внесені 27 листопада 2007 року;
- жорстка залежність від компілятора через прив'язку до абстрактного синтаксичного дерева (АСД), таблиці символів (ТС), що генеруються. [7]

MUSCAT

Даний інструмент є розширеною версією PINOT з виділенням мови візуального уточнення MUSCAT (Minimal UML SpecifiCATion language), яка дозволяє користувачам визначати структурні та поведінкові аспекти шаблонів проектування.

Переваги:

- візуалізація шаблонів проектування;
- можливість модифікації аспектів шаблонів проектування.

Недоліки:

- необхідність вивчення створеної мови для налаштування;
- відсутність можливості додавання нових шаблонів проектування;
- залежність від програмного забезпечення, підтримка якого вже закінчилась. [7]

JPINOT

Цей проект являється копією PINOT, виконаний як надбудова до javac та слугує фундаментом для подальшого покращення виявлення шаблонів проектування за допомогою PINOT. [7]

FUJABA

Програмний продукт дозволяє підтримку моделе-орієнтованої методики створення програмного забезпечення та підтримки створеного проекту. Спочатку Fujaba мав на меті полегшення розробки та підтримки створених проектів, оскільки його назва є акронімом “From UML to Java and back again”.

Переваги:

- використання потужного апарату Unified Modeling Language (UML) діаграм класів та зв'язків між класами;
- широка міжнародна підтримка даного проекту.

Недоліки:

- зміщення акценту з виявлення шаблонів проектування на підтримку моделе-орієнтованої методики розробки;
- нижчі показники виявлення шаблонів проектування порівняно із аналогічною утилітою PINOT;
- зміна вектору підтримки, перехід до реалізації будови діаграм історій. [7]

WOP

WebOfPatterns (WOP) - інструментарій, який має на меті поширення

знань про архітектуру програмного забезпечення. WOP складається з двох частин:

- мово-незалежного формату для опису шаблонів проектування за стандартами World Wide Web Consortium (W3C), а саме за допомогою моделі для опису даних (англ. Resource Description Framework, RDF) та мови опису онтології для семантичного павутиння (англ. Web Ontology Language, OWL);
- набір Eclipse-додатків, які використовують вищезгадані мово-незалежні формати.

Переваги:

- мово-незалежна формалізація шаблонів проектування;
- можливість оцінювати та брати участь у створенні опису до шаблонів проектування усіма учасниками проекту та онлайн підтримка.

Недоліки:

- нижчі показники виявлення шаблонів проектування порівняно з аналогічною утилітою PINOT;
- припинення підтримки, останнє оновлення проекту датується 18 березням 2007 року. [7]

2. МОДИФІКОВАНИЙ СПОСІБ РОЗПІЗНАВАННЯ ПОДІБНОСТІ ДЕРЕВ

При проектуванні системи однією з головних вимог була відсутність проміжного рівня для представлення шаблонів проектування, які необхідно розпізнавати у вихідному коді програмного забезпечення. Було вирішено для задання шаблону проектування використовувати безпосередньо вихідний код, який представляє собою реалізацію шаблону проектування. Для того щоб розпізнавати шаблон, використовується абстрактне синтаксичне дерево (АСД) для перетворення вихідного коду з текстового представлення до структурованих даних. Після цього виконується перетворення АСД до деревоподібної структури, яка в якості вершин представляє класи та інтерфейси системи, що аналізується, а в якості ребер — зв'язки діаграми класів. Після побудови зв'язків, виконується порівняння деревоподібних структур даних, використовуючи модифікований спосіб повного перебору для визначення відповідності дерев.

2.1. Використання синтаксичного дерева для розпізнавання шаблонів проектування.

Синтаксичний аналіз — це процес аналізу вхідної послідовності символів, з метою розбору граматичної структури згідно із заданою формальною граматикою. Синтаксичний аналізатор — це програма або частина програми, яка виконує синтаксичний аналіз. [8]

Під час синтаксичного аналізу текст оформлюється у структуру даних, зазвичай — в дерево, яке відповідає синтаксичній структурі вхідної послідовності, і підходить для подальшої обробки. Синтаксичний

аналізатор — це програмний компонент, який приймає вхідні дані і створює структуру даних — часто дерево розбору, абстрактне дерево синтаксису або іншу ієрархічну структуру — забезпечує структурне представлення вводу, перевіряє правильність синтаксису в процесі. У якості вхідних даних може бути подана послідовність токенів, які були отримані після виконання лексичного аналізу, або безпосередньо вихідний код. В останньому випадку, лексичний та синтаксичний аналізатори об'єднані в єдину систему. Аналізатори можуть бути запрограмовані вручну, автоматично або напівавтоматично генератором парсерів. [8]

Абстрактні синтаксичні дерева — це структури даних, широко використовувані у компіляторах, для представлення структури програмного коду. АСД зазвичай є результатом етапу аналізу синтаксису компілятора. Синтаксичне дерево служить проміжним поданням програми, яке вимагає компілятор або інше програмне забезпечення для аналізу вихідного коду.

АСД володіє декількома властивостями, які допомагають подальшим крокам процесу компіляції:

- АСД можна редагувати та розширювати, використовуючи атрибути та анотації для кожного елемента, який він містить. Таке редагування та анотацію неможливо виконати з вихідним кодом програми, оскільки це буде означати його зміну.
- у порівнянні з вихідним кодом АСД не включає несуттєві розділові знаки та розділювачі (фігурні дужки, крапкові крапки, круглі дужки тощо).
- АСД, як правило, містить додаткову інформацію про програму через послідовний етап аналізу компілятором. Наприклад, він може зберігати положення кожного елемента у вихідному коді, дозволяючи компілятору друкувати корисні повідомлення про помилки.

Побудова АСД необхідний етап аналізу вихідного коду через властивості мов програмування та їх документації. Мови часто є неоднозначними за своїм характером. Щоб уникнути цієї неоднозначності, мови програмування часто визначаються контекстно-вільною граматикою (КВГ). Проте часто існують аспекти мов програмування, які КВГ не може виразити. Однак вони є частиною мови та документовано в його специфікації. Такі деталі вимагають контекст для визначення їхньої поведінки та впливу на роботу програми. Наприклад, якщо мова дозволяє оголосити нові типи, КВГ не може передбачити імена таких типів, а також спосіб їх використання. Навіть якщо мова має попередньо визначений набір типів, для забезпечення належного використання потрібний певний контекст.

У розробленій системі використовується стороння бібліотека Instaparse для генерації синтаксичного аналізатора мови програмування Java. На вхід до генератору може бути подана будь-яка контекстно-вільна граматика. У ході розробки було створено граматику, яка продукує синтаксичне дерево з меншою кількістю вершин порівняно з повною граматикою мови програмування, для пришвидшення роботи першого етапу системи - побудови АСД.

2.2. Задача розпізнавання подібності дерев

Задача розпізнавання подібності дерева може бути сформована таким чином: задано два дерева T та P, необхідно знайти усі входження дерева P у дереві T. На рис. 2.1 [9] зображено приклад двох дерев T та P.

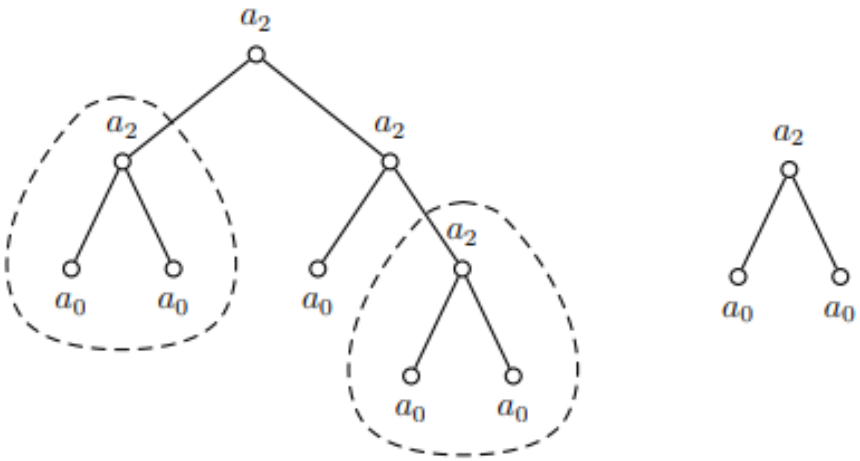


Рисунок 2.1. Графічне представлення задачі розпізнавання подібності дерев.

На рис. 2.1 штрихованою лінією виділені піддерева P (зображено справа) у дереві T (зображено зліва).

Для розпізнавання двох дерев необхідно визначити необхідно визначити дві функції:

- Функція (traverse), яка забезпечить однаковий порядок обходу дерева. Ця функція являє собою генератор, який повертає чергову вершину дерева та null у випадку закінчення вершин.
- Функція (compare), яка порівнює дві вершини дерева. Результатом цією функції є булеве значення, яке вказує на еквівалентність вершин.

На основі цих функцій можна представити базовий спосіб розпізнавання дерева P у дереві T у вигляді алгоритму 1 верхнього рівня, який у випадку виявлення друкує вершину дерева T з якої починається дерево P.

Алгоритм 1. Розпізнавання подібності дерев

```
Вхідні дані: T, P дерева
loop (для кожної вершини t_vertex дерева T) do
  match ← true
    loop (для кожної вершини p_vertex дерева P) do
      if compare(t_vertex, p_vertex) == false then
        match ← false
        break
      end if
    end loop
    if match == true then
      output(t_vertex)
      break
    end if
end loop
```

Складність алгоритму визначається як $O(t*p)$, де t - кількість вершину дереві T, p - кількість вершин у дереві P. [9]

Розглянемо приклад використання вищезгаданого алгоритму для розпізнавання подібності шаблонів проектування. Функція обходу по дереву може бути визначена будь-яким чином для отриманої деревоподібної структури зі зв'язками діаграми класів. Функція порівняння двох вершин не може бути використана оскільки вершини між двома деревоподібними структурами рівнозначні. В такому випадку, задача розпізнавання подібності дерев ускладнюється, коли необхідно знайти не конкретне піддерево, а шаблон піддерева, який визначається зв'язками між вершинами дерева.

Для розглянутої задачі розпізнавання використання шаблонів проектування задача розпізнавання відповідності дерев є частковим випадком більш загальної задачі визначення ізоморфності графів.

Загальний випадок рішення задачі ізоморфізму є задачею, яку відносять до групи NP-повних задач. Частковий випадок цієї задачі, коли визначається відповідність між деревами, може бути спрощений завдяки відсутності будь-якого виду циклічності у деревах, наявності однієї кореневої вершини, від якої будуються зв'язки з іншими вершинами, та іншими характеристиками дерева згідно теорії графів. [10]

Здача визначення ізоморфності графів використовується у багатьох сферах де можна спростити реальну модель до подання у вигляді графу. Наприклад, одним із застосувань у сфері автоматизації проектування електронних схем є верифікація представлення різних схем, що виконується завдяки пошуку відповідності між графами, при статичному аналізі вихідного коду деякі способи знаходження дублікатів побудовані на основі пошуку подібності між деревами.

Класичним алгоритмом розпізнавання ізоморфізму є спосіб Ульмана [11]. У статті, представлення способу розділено на дві частини: представлення простого алгоритму перебору для визначення ізоморфізму графів та його модифікація за допомогою процедури обробки матриці для виконання перестановок.

Задачею алгоритму перебору є пошук усіх ізоморфізмів між заданим графом $G_\alpha = (V_\alpha, E_\alpha)$ та підграфами для заданого графа $G_\beta = (V_\beta, E_\beta)$. Кількість вершин та ребер графів G_α та G_β позначимо як p_α, q_α та p_β, q_β відповідно. Матриця суміжності графів G_α та G_β позначимо як $A = [a_{ij}]$ та $B = [b_{ij}]$ відповідно.

Визначимо матрицю M^1 з розмірністю $p_\alpha \times p_\beta$ та елементами одиниця або нуль. Кожний рядок цієї матриці має тільки одну одиницю та кожен рядок має не більше однієї одиниці. Матриця $M^1 = [m_{ij}^1]$ буде використана для того, щоб виконувати перестановки рядків та колонок матриці B для отримання матриці C. Визначимо матрицю C як $C = [c_{ij}] = M^1(M^1 B)^T$, де T

означає транспонування. Якщо виконується умова (2.1), то можна стверджувати, що матриця M^1 визначає ізоморфізм між графом G_α та підграфом G_β.

$$(\forall i \ \forall j) = (a_{ij} = 1) \Rightarrow (c_{ij} = 1), \quad (2.1)$$
$$1 \leqslant i \leqslant p_\alpha, \quad 1 \leqslant i \leqslant p_\beta$$

В такому випадку, якщо $m_{ij}^1 = 1$, то вершина *j* у графі G_β відповідає вершині *i* у графі G_α для цього ізоморфізму.

На початку роботи алгоритму необхідно побудувати матрицю розмірності $p_\alpha \times p_\beta$, яку ми позначимо як $M^0 = [m_{ij}^0]$, де

$$m_{ij}^0 = \begin{cases} 1 \text{ якщо ранг } j \text{ вершини графа } G_\beta \text{ більший або дорівнює рангу вершини } i \text{ графа } G_\alpha \\ 0 \text{ у інших випадках} \end{cases}$$

Необхідно використовувати нуль, якщо ми можемо бути впевнені, що *j* вершина графа G_β не може відповідати вершині і у ізоморфізмі графа G_α.

Робота алгоритму полягає у генеруванні усіх можливих матриць M^1 для яких виконується така умова (2.2) для кожного елементу m_{ij}^1 матриці.

$$(m_{ij}^1 = 1) \Rightarrow (m_{ij}^0 = 1), \quad (2.2)$$

Для кожної такої матриці M^1 алгоритм перевіряє ізоморфізм за умовою (2.1). Матриці M^1 генеруються за допомогою систематичної заміни на нуль усіх одиниць за виключенням однієї для кожного рядку матриці M^0, виконуючи умову про те, що жодний стовпчик матриці M^1 не може мати більше за одну одиницю. У дереві пошуку, термінальні вершини мають глибину *d* яка дорівнює p_α і вони співвідносяться до матриць M^1. Кожна нетермінальна вершина, для якої виконується умова $d < p_\alpha$ співвідносяться до матриці *M*, яка відрізняється від M^0 тим, що у *d* рядків усі одиниці за винятком однієї замінені на нуль.

Алгоритм використовує бінарний вектор $\{F_1, \ldots, F_i, \ldots, F_{p_\beta}\}$ розміром

p_β бітів для запису які стовпчики були використані у проміжному стані обчислення. $F_i = 1$ якщо i стовпчик був використаний. Алгоритм використовує бінарний вектор $\{H_1, \dots, H_d, \dots, H_{p_\alpha}\}$ для збереження інформації про те який стовпчик було обрано на якій глибині. $H_d = k$ Якщо стовпчик k було використано на глибині d.

Якщо в описі алгоритму вказано вираз виду $M \leftarrow M_d$, то це означає, що присвоїти усій матриці M значення усієї матриці M_d. Матриця M_d означає повну копію матриці M до глибини d. Вищеописані дії сформовані у вигляді алгоритму що наведено у додатку А.

Для того, щоб зменшити кількість операцій, які необхідно виконати для пошуку ізоморфного графу, було розроблено процедуру, яка видаляє деякі одиниці з матриці M^1. Внаслідок цього відбувається зменшення кількості вершин у дереві пошуку.

Для того щоб визначити процедуру, необхідно розглянути матрицю M, яка зв'язана з усіма нетермінальними вершинами у дереві пошуку. Будь-який ізоморфний граф відповідає конкретній матриці M^1. Отже ізоморфізм графу це ізоморфізм у матриці M, якщо термінальна вершина у дереві пошуку з'єднана з вершиною, яка має суміжність у матриці M. Нулі в матриці М просто заважають з'єднанню між вершинами V_α і V_β. Якщо $m_{ij}^1 = 0$ для усіх ізоморфізмів у матриці M, тоді для випадку $m_{ij}^1 = 1$ ми можемо змінити на $m_{ij}^1 = 0$ без втрати ізоморфізмів у матриці M, оскільки ізоморфізми все одно будуть знайдені деревом пошуку. Наступною задачею є створення умови за якою можна визначити випадки коли $m_{ij}^1 = 1$ визначає будь-який ізоморфізм у матриці M. Якщо ця умова не виконується а $m_{ij}^1 = 1$, то розроблена процедура може замінити $m_{ij}^1 = 1$ на $m_{ij}^1 = 0$.

Нехай $v_{\alpha i}$ це i вершина у V_α, а $v_{\beta j}$ це j вершина у V_β. В такому випадку можна визначити множину вершин, які з'єднані між $v_{\alpha i}$ та графом G_α, як множину вершин графа G_α: $\{v_{\alpha 1}, \dots, v_{\alpha x}, \dots v_{\alpha y}\}$. Розглянемо

матрицю M^1, яка з'єднана з будь-яким ізоморфізмом у матриці M. З визначення ізоморфізму графів слідує, що виконання наступної умови є обов'язковим: якщо вершина $v_{\alpha i}$ відповідає вершині $v_{\beta j}$ у ізоморфному графі, тоді для кожного значення $x=1, \ldots, \gamma$ повинна існувати вершина $v_{\beta y}$ у V_β, що з'єднана з вершиною $v_{\beta j}$, так що вершина $v_{\beta y}$ відповідає вершині $v_{\alpha i}$ у ізоморфному графі. Якщо $v_{\beta y}$ відповідає вершині $v_{\alpha x}$ у ізоморфному графі, тоді одиниця повинна бути елементом матриці M^1, що відповідає ребру між вершинами $v_{\alpha x}$ і $v_{\beta y}$. Отже, якщо $v_{\alpha i}$ відповідає вершині $v_{\beta j}$ у будь-якому ізоморфному графі у матриці M, тоді для кожного значення $x=1, \ldots, \gamma$ повинна бути одиниця у матриці M, яка відповідає ребру між вершинами $v_{\alpha x}$ і $v_{\beta y}$ за умови що вершина $v_{\beta y}$ поєднана з вершиною $v_{\beta j}$. Іншими словами, якщо $v_{\alpha i}$ відповідає $v_{\beta j}$ у будь-якому ізоморфізмі у матриці M, тоді виконується умова (2.3)

$$\underset{1\le x\le p_\alpha}{(\forall x)}\left((\alpha_{\alpha x}=1)\Rightarrow \underset{1\le y\le p_\beta}{(\exists y)}\ (m_{xy}\cdot b_{yj}=1)\right), \qquad (2.3)$$

Процедура для заміни одиниць на нулі у матриці M, почергово перевіряє кожну одиницю у матриці M на виконання умови (2.3). Для будь-якого елемента матриці $m_{ij}=1$ для якого умова (2.3) не виконується, відбувається заміна з $m_{ij}=1$ на $m_{ij}=0$. Така заміна може спричинити невиконання умови для інших одиниць у матриці, тому подальші заміни можуть бути виконані. Процедура для заміни повинна виконуватися до тих пір, поки на певному кроці жодна одиниця не може бути замінена на нуль. Варто зауважити, що порядок виконання процедури може бути будь-яким і не впливає на кінцевий результат. Така гнучкість процедури дозволяє її виконувати паралельно для зменшення часу роботи усіх кроків та отримання кінцевої матриці M.

Метою процедури є заміна одиниць матриці M на нулі, але можливий випадок, коли процедура не змінить значення матриці M. Це важливо для випадку, коли матриця M є матрицею M^1. Необхідною та достатньою умовою для визначення ізоморфних графів є випадок, коли процедура залишає матрицю M^1 без змін. З цього слідує, що матриця M^1 задає відповідність один до одного між V_α та V_β таку, що для випадку коли дві вершини поєднані у графі G_α, тоді дві відповідні вершини графа G_β також поєднані. Тому можна використовувати процедуру як перевірку на ізоморфність графів замість умови (2.1). Нову умову можна сформулювати наступним чином: якщо процедура змінює хоча б одну одиницю у матриці M^1, можна стверджувати, що матриця M^1 не задає ізоморфний граф.

У ході роботи процедури відбувається перевірка, що ніякий рядок матриці M^1 не містить одиниць. Якщо хоча б один рядок матриці M^1 містить одиницю, то виконання процедури завершується достроково, оскільки немає сенсу продовжувати роботу далі. У випадку, коли усі рядки пройдені, процедура завершується успішно.

Розглянутий спосіб Ульмана показує свою ефективність для роботи з графами, однак існують способи, які покращують процес пошуку ізоморфних графів, для випадку, коли вхідні графи є деревами. Одним з таких способів є спосіб Чунга [12].

Зробимо деякі визначення для позначень, що будуть використовуватись надалі при описі способу. Дерево з корном — набір трьох значень $G = (V, E, r)$, де $G = (V, E, r)$ — дерево без кореня з множиною вершин V та множиною ребер E, а r — вершина з V, яка є коренем дерева. Надалі буде використовуватись позначення виду G^r, яке визначає дерево з коренем у якості вершини r. Позначення виду G_v^r означає піддерево дерева з коренем G^r, в якому усі вершини є нащадками вершини v та вершина v є

коренем цього піддерева. Два дерева з коренями G^r і H^s є ізоморфними, якщо існує ізоморфізм між деревами G і H, який задає відповідність між вершинами r та s. Відношення виду $H^s \subseteq G^r$ означає, що існує дерево з коренем J^r, яке є піддеревом дерева G^r, ізоморфне до дерева H^s. Варто відзначити, що дерева J^r та G^r мають одну і ту й саму вершину в якості кореня.

Надалі буде використовуватися поняття відкритого сусідства для вершини v у графі G позначенням $N(v) = \{ u : uv \in E \}$, яке задає множину вершин, які з'єднані з вершиною v у графі G з виключенням самої вершини. Закрите сусідство буде позначатися $N[v] = N(v) \cup \{ v \}$ і буде означати відкрите сусідство із включенням вершини v до множини. Позначення виду v визначає глибину вершини v у графі G.

Нехай задано дерево G з множиною вершин V та множиною ребер E та дерево H з множиною вершин V_H та множиною ребер E_H. В такому випадку, дерева можна представити як $G = (V, E)$ та $P = (V_H, E_H)$ відповідно. Позначимо корінь дерева G як вершину r. Необхідно визначити чи для кожного вершини $v \in V$ та вершини $u \in V_H$ виконується умова $H^u \subseteq G_v^r$. Для визначення виконання цієї умови більш ефективно, також необхідно визначити для кожної вершини w, яка є сусідом вершини v чи виконується твердження $H_u^w \subseteq G_v^r$. Варто відзначити, що дерево H_u^w отримане видаленням графа H_w^u з H^u. Ця інформація необхідна оскільки умова $H^u \subseteq G_v^r$ може бути виконана у випадку, якщо кожної дитини u^1 вершини u існує дитина v^1 вершини v для якої виконується умова $H_{u^1}^u \subseteq G_{v^1}^r$.

В якості узагальнення, можна сформувати підзадачу розпізнавання ізоморфного дерева як таку: для кожної вершини v дерева G^r, вершини u у дереві H та вершини w для якої виконується умова $w \in N[u]$ ми можемо стверджувати $H_u^w \subseteq G_v^r$ за умови, якщо для кожної вершини-дитини u^1

вершини u у дереві H_u^w існує вершина-дитина v^1 вершини v для якої виконується умова $H_{u^1}^u \subseteq G_{v^1}^r$.

Ця інформація зберігається у множини $S(v,u)$ визначених таким чином, що для кожної вершини $v \in V$ та для кожної вершини $u \in V_H$:

$$S(v,u) = \{ w \in N[u] : H_u^w \subseteq G_v^r \}$$

На рис. 2.2 [12] наведено приклад для визначення вище. Варто зауважити декілька пунктів:

1. Для вершина u виконується умова $u \in S(v,u)$ якщо і тільки якщо виконується умова $u \in S(v,u)$
2. Виконання умови $u \in S(v,u)$ означає, що виконується рівність $u \in S(v,u)$
3. Виконання умови $d(v) < d(u)-1$ означає, що $S(v,u) = \emptyset$.

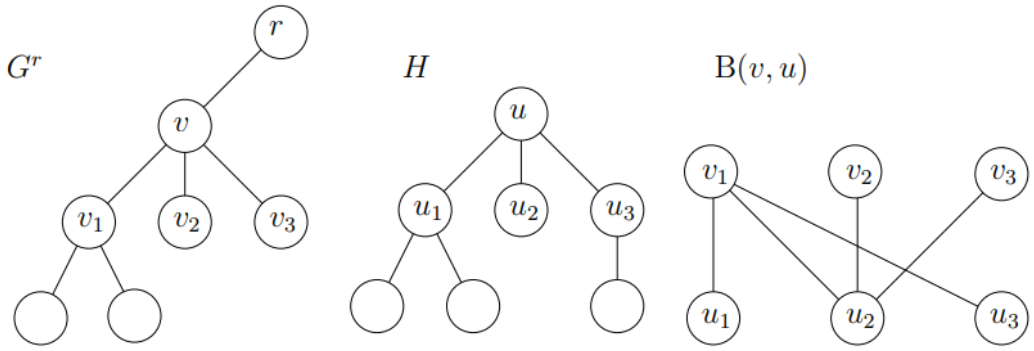


Рисунок 2.2. Приклад ізоморфізму графів

На представленому рис. 2.2 ми бачимо виконання умов $H^u \not\subseteq G_v^r$, $H^u \not\subseteq G_v^r$, а також $H_u^{u^1} \subseteq G_v^r$, $H_u^{u^3} \subseteq G_v^r$. З цього слідує, що шукана множина $S(v,u)$ має у собі дві вершини: u^1 та u^3, тобто $S(v,u) = \{u^1, u^3\}$. Граф $B(v,u)$ це біграф, який було сконструйовано для отримання множини $S(v,u)$. У цьому біграфі є ребро u^1 у випадку, якщо виконується умова $S(v,u)$.

Робота способу полягає у визначенні множин $S(v,u)$ для усіх вершин v та u. Алгоритм 2, що описує спосіб Чунга, формулює послідовність дій для визначення $S(v,u)$ індуктивно виконуючи прохід по усім вершинам v графа G^r від термінальних вершин до кореня та пошуком множини $S(v,u)$ для усіх вершин u, що належать дереву V_h. Основою індукції є випадок, коли вершина v є термінальною для дерева G^r. Відштовхуючись від цього множина $S(v,u)$ може бути визначена наступним чином: якщо вершина u є термінальною вершиною графа H, тоді до $S(v,u)$ додається одна вершина, яка є сусідом вершини u у графі H. У випадку, якщо вершина u не є термінальною вершиною, тоді множина $S(v,u)$ є пустою. Наступним індуктивним кроком є розгляд нетермінальної вершини v. Спочатку нам необхідно визначити множину $S(v^1,u)$ для усіх вершин-дітей v^1 вершини v для усіх вершин u для яких виконується умова $u \in V_H$. Для того, щоб обрати вершину w для якої виконується умова $w \in N[u]$, якщо $w \in S(v,u)$, необхідно побудувати біграф $B_w(v,u)$ з двох частин $B_w(v,u)$ і $Y^{v,u}$, де $X_w^{v,u}$ це множина вершин-дітей для вершини u у дереві H_u^w, а $Y^{v,u}$ це множина вершин-дітей для вершини v, а u^1v^1 є ребром графа $B_w(v,u)$, коли одна з рівнозначних умов $H_{u^1}^u \subseteq G_{v^1}^r$ або $u \in S(v^1,u^1)$ виконується. Для того, щоб знайти $S(v,u)$ необхідно знайти максимальну кількість збігів у біграфах для рівня $d(u)+1$. Варто зазначити, що ці бінарні графи схожі один до одного, адже кожен граф $B_w(v,u)$ отримано шляхом видалення вершини w з графу $B_u(v,u)$. На рис. 2.2 зображено зв'язок між множиною $S(v,u)$ та біграфом $B(v,u)$.

Алгоритм 2. Спосіб Чунга для виявлення ізоморфних піддерев

Вхідні дані: дерева G та H

Взяти кореневу вершину r графа G

for $u \in H, v \in G$ **do** $S(v, u) \leftarrow \varnothing$

for термінальні вершини v графа G^r **do**

for термінальні вершини u графа H **do** $S(v, u) \leftarrow N(u)$

for нетермінальні вершини v G^r у напряму від термінальних вершин **do**

нехай v вершини-діти вершини v

for усі вершини u графа H з глибиною не більше за $t + 1$ **do**

нехай v вершини-діти вершини u

побудувати біграф $B(v, u) = (X, Y, E_{vu})$, де

$X = \{ u_1, \ldots, u_s \}$,

$Y = \{ v_1, \ldots, v_t \}$,

$E_{vu} = \{ u_i v_j : u \in S(v_j, u_i) \}$

позначимо $X_o = X$ та $X_i = X - \{ u_i \}$

for $0 \leqslant i \leqslant s$ **do**

визначити максимальну кількість збігів m_i між X та Y

end for

$S(v, u) \leftarrow \{ u_i : m_i = |X_i|, 0 \leqslant i \leqslant s \}$

if $u \in S(v, u)$ **then**

output (“YES”)

exit

end if

end for

end for

output (“NO”)

2.3. Модифікований спосіб повного перебору для розпізнавання відповідності дерев

Розглянуті способи у розділі 2.2 вирішують задачу розпізнавання відповідності дерев керуючись пошуком ребер які необхідно згрупувати, для того, щоб отримати шукане дерево. У представленому модифікованому способі використовується підхід за яким допускається, що будь-які вершини та їх зв’язки можуть утворити ізоморфне дерево [14]. Зв’язки між вершинами дерев можна представити у вигляді списку пар ключ-значення, так званого словника, де ключем є вершина-батько, а значенням — вершина-дитина. Використовуючи структуру даних як словник спрощується визначення кінцевого рішення. Ним має бути один або декілька словників відповідності в яких ключами є вершини дерева з яким визначається відповідність, а значеннями — вершини дерева в якому знаходиться відповідність. Тобто представлені словники відповідності є кандидатами для верифікації, процес порівняння яких, має складність O(1) у кращих випадках та O(log n) у гірших випадках. В свою чергу змінюється й критерій перевірки закінчення алгоритму. Наявність усіх вершин дерева, що порівнюється у якості ключів в словнику відповідності між вершинами двох дерев являється критерієм завершення роботи.

Основою способу, що пропонується, є спуск по ярусам дерев з побудовою усіх можливих варіантів відповідності між деревами, оскільки вхідні структури даних є невідсортованими і необхідно виконати усі перестановки для визначення можливих варіантів рішення. Наступний етап – пошук помилки у побудованих варіантах відповідності та збільшення кількості елементів словника, поки кількість записів у кандидаті не буде дорівнювати кількості вершин дерева, з яким відбувається процес пошуку відповідності. Процес перебору згідно даного способу складається з трьох

кроків:

1. Взяття чергового ярусу дерева.
2. Побудова усіх можливих варіантів відповідності між деревами на ярусі.
3. Злиття кожного варіанту з існуючим словником відповідності.

Відмінність даного способу від способу повного перебору полягає у зменшенні кількості можливих варіантів відповідності пошуком невизначеності у словнику після кожної ітерації. Невизначеність у словнику – це двозначність між ключами або значеннями, тобто два ключі відповідають одному значенню, або навпаки два значення відповідають одному ключу. Внаслідок використання критерію невизначеності у словнику, кількість ітерацій зменшується порівняно зі способом повного перебору. Наведений спосіб закінчує свою роботу у двох випадках: пройдені усі вершини або знайдена двозначність словника.

У разі вичерпання ярусів обох вхідних дерев, можна зробити висновок, що було знайдено шукану відповідність між двома структури. В загальному випадку, може бути більше ніж один словник повної відповідності між вершинами дерев, оскільки повна перестановка дає велику кількість можливих варіантів. Випадок, коли кожен з можливих варіантів продукує колізію в словнику відповідності, визначає неповну відповідність між двома деревами. В залежності від кількості вершин, що були отримані в результаті визначення гомоморфізму структур даних, можна підрахувати ступінь відповідності.

Наведемо опис послідовних дій запропонованого способу:

1. Ініціалізація двох списків повної та часткової відповідності пустими словниками.
2. Паралельний обхід двох вхідних дерев, починаючи з першого ярусу. Далі будуть циклічно виконуватися наступні дії:
   2.1. Виконання перестановки вершин на поточному ярусі другого

дерева по кількості вершин з поточного ярусу першого дерева.

2.2. Створення словника відповідності для кожної перестановки з п.2.1., де ключем є вершина з першого дерева поточного ярусу, а значенням – вершина з другого дерева поточного ярусу.

2.3. Кожен словник з п.2.2. поєднується з кожним елементом зі списку відповідності. Словники, в яких при поєднанні не виникла колізія, є новим значенням списку словників відповідності. Словники, в яких при поєднанні виникла колізія між ключами або значеннями, вважаються словниками часткової відповідності і додаються до списку часткової відповідності, після чого вже не приймають участі у подальшому поєднанні.

2.4. Якщо усі словники з п.2.2 продукують невідповідність у результуючих словниках відповідності, то необхідно перенести усі словники зі списку повної відповідності до списку часткової відповідності, і виконання способу закінчується. В інакшому випадку, виконується взяття наступного ярусу з вхідних дерев та перехід до п. 2.1. У разі, якщо наступного ярусу в одному з вхідних дерев немає, виконання способу закінчується.

3. Результатом виконання дій способу є два списки. В залежності від наявності або відсутності словників в списку повної відповідності, можна зробити висновок про еквівалентність між вхідними деревами. [14]

Оскільки представлений спосіб на виході надає не тільки словник з повною відповідністю вершин, а й множину словників з частковою відповідністю, то за допомогою даних словників можна визначити відсоткове значення співпадіння між вхідними деревами двох типів: за кількістю безпомилкових варіантів відповідності між вершинами та повних ярусів дерев. Відсоткове значення може бути знайдене як відношення кількості вершин з безпомилковою відповідністю до кількості вершин

дерева, з яким визначалась відповідність.

Наведене вище відношення може бути модифіковане для пошуку відсотку відповідності між деревами за кількістю повних ярусів дерев, що відповідають безпомилково. Для цього необхідно замінити лише підрахунок кількості вершин на підрахунок кількості ярусів. Якщо інформація про показники дерев, що порівнюються, відсутня, то для повної оцінки результату пошуку відповідності необхідно використовувати обидва показники, а саме: середнє значення вершин у ярусі, середня кількість вершин-дітей та інші.

Отримані словники відповідності можуть бути використані у різних сферах, в яких необхідно визначити відповідність між двома моделями, за умови, що моделі можуть бути представлені у вигляді дерева. При проектування електронних схем існує необхідність автоматизація процесу перевірки різних представлень електронних схем. Також існують способи повної автоматизації оптимізації програмного забезпечення за допомогою пошуку загальних абстрактних синтаксичних дерев для зменшення кількості дублювання у коді, а також часткової автоматизації програмного забезпечення у випадку аналізу зв’язків між структурами даних, змінними та іншими логічними частинами програмного забезпечення. Іншим прикладом використання пошуку відповідності між деревами є аналіз шаблонів проектування у вихідному коді програмного забезпечення. Такий підхід активно використовується у випадку аналізу вихідного коду значного розміру для подальшого автоматизованого або ручного рефакторингу.

Визначені часткові відповідності можуть бути використані для рішення задач пов'язаних з ізоморфністю дерев та інших видів направлених графів. Прикладами таких задач є пошук найбільших або найменших спільних піддерев та визначення мінімального розширення вершин та ребер для повної відповідності між вхідними направленими графами.

3. СТРУКТУРА СИСТЕМИ ДЛЯ ВИЯВЛЕННЯ ШАБЛОНІВ ПРОЕКТУВАННЯ ТА ЇЇ ТЕСТУВАННЯ

Система для виявлення шаблонів проектування складається з трьох окремих модулів, що не залежать один від одного.

Призначення першого модуля - побудова абстрактного синтаксичного дерева на основі констекстно-вільної граматики. Вхідний параметр для цього модуля - розширена або доповнена нотація Бекуса-Наура, яка описує граматику об'єктно-орієнтованої мови. Вихідне значення - АСД, яке описане структурою даних мовою Clojure.

Другий модуль будує графи, які зображують зв'язки між класами використовуючи типи зв'язків із діаграм класів. Вхідний параметр - результат роботи першого модулю.

Третій модуль відповідальний за виявлення шаблону проектування. Вхідними параметрами для цього модулю - два набори графів, отриманих в результаті роботи другого модулю. Перший набір графів описує шаблон проектування, а другий набір представляє аналізовану систему. Результатом роботи є повідомлення про повну чи часткову присутність або відсутність шаблону проектування в аналізованій системі.

Перевагою даної системи є те, що вона не прив'язана до конкретної реалізації шаблону проектування та може аналізувати присутність будь-якого шаблону в аналізованому вихідному коді.

3.1. Контекстно-вільна граматика мови Java

Для реалізації системи виявлення шаблонів проектування було обрано аналіз мови програмування Java. Створена граматика описує лише ті частини мови, що необхідні для побудови графів зв'язків: оголошення

класу, інтерфейсу, пакету, імпорту пакету, методу, атрибуту, локальної змінної, формального параметру та значення, що повертається з методу.

У графі, що відображає зв'язки, що використовуються у діаграмах класів, вершинами являються оголошені класи, інтерфейси. Через це була необхідність додавання в граматику правила, що описують створення класів або інтерфейсів. В мові програмування Java реалізація зв'язків узагальнення і реалізації є простою.

Для спадкування існує ключове слово extends, завдяки якому по оголошенню класу зрозуміти, який клас успадковується. Для опису зв'язку реалізації введено ключове слово implements. В оголошенні класу, усі ідентифікатори, що стоять після implements є ідентифікаторами інтерфейсів. Таких ідентифікаторів може бути декілька. Оголошений клас повинен реалізувати усі методи, що оголошені у відповідних інтерфейсах.

У мові програмування Java є поняття повністю кваліфікованого ідентифікатора, який є унікальним для усього проекту. Для побудови коректної відповідності ідентифікаторів системи, що аналізується, необхідно знати в якому пакеті оголошений клас і місце його використання. Для цього було додано правило для оголошення пакетів та їх імпорту до граматики.

За допомогою оголошення методів класу можна зробити висновок про те, які типи були використані у ньому. Завдяки доданому правилу оголошення методу можна будувати ребра, які описують взаємозв'язки між класами, графу. Всередині оголошення методу існують усі інші правила, завдяки яким можна розділити які взаємозв'язки існують між об'єктами класів.

Якщо тип атрибуту класу не є примітивним, то це означає, що між типом, який має оголошений атрибут, та класом, в якому атрибут оголошено, існує зв'язок. Якщо даний атрибут не являється контейнером, то це означає, що між класами існує залежність. В інакшому випадку, це

означає, що зв'язок є асоціацією, а саме композицією.

Якщо локальна змінна або формальний параметр використовує певний клас в якості типу контейнера, це показує зв'язок асоціації, а саме агрегацію. Якщо ж тип змінної є не примітивний тип, це означає, що існує залежність між класом, який використовує цю змінну та типом, що використовується.

У разі, якщо значення, що повертає метод, не є контейнером, то це вказує на зв'язок залежності. В інакшому випадку, значення, що повертається створює зв'язок асоціації між двома класами, якщо тип контейнеру не є примітивним.

3.2. Синтаксичний аналізатор та побудова дерева розбору

Граматика є вхідним параметром для створення функції, яка поєднує в собі лексичний та синтаксичний аналізатори, з подальшим її використанням для побудови АСД. Комбінація аналізаторів будується за допомогою сторонньої бібліотеки Instaparse.

Дана бібліотека було обрана за ряд переваг:

- жодна граматика не залишається осторонь: функція може бути побудована для будь-якої контекстно-вільної граматики, включаючи ліворекурсивні, праворекурсивні та неоднозначні граматики;
- поєднання лексичного та синтаксичного аналізаторів завдяки можливості використання регулярних виразів для опису правил граматики;
- розширено клас граматики, що підтримуються, крім контекстно-вільних граматик можна працювати з PB-граматиками, що розбирають вирази за допомогою попереднього прямого перегляду та оберненого попереднього перегляду;

- деталізоване повідомлення помилок розбору;
- може продукувати ліниву послідовність усіх можливих АСД;
- аналізує увесь вихідний текст програми, де підрядки, що не піддаються розбору вставляються у дерево розбору.

Час побудови та аналіз абстрактного синтаксичного дерева спрощується за рахунок неповного опису мови у створеній контекстно-вільній граматиці.

3.3. Побудова графів зв'язків між класами для вихідного коду

Задачею другого модуля є побудова графа зв'язків, що використовуються у діаграмах класів, на основі АСД.

Для побудови кожного із видів зв'язків необхідно проаналізувати відповідну вершини дерева для того. Усі функції, що будують зв'язки об'єднує рекурсивне використання вбудованої бібліотеки match в мову Clojure, а саме однойменного макросу для пошуку відповідних вершин дерева, що необхідні для аналізу.

Даний макрос реалізує алгоритм компіляції зіставлення з шаблоном, який використовує поняття "необхідність", при виявленні відповідності зі зразком. Бібліотека використовує алгоритм, який описаний у статті Лука Меренгата (Luc Maranget). Стаття має назву “Трансляція співставлення шаблону в гарне дерево рішень” (Compiling Pattern Matching to good Decision Trees).

Вхідні параметри для усіх функцій однакові - абстрактне синтаксичне дерево одного файлу та словник з ключем-ідентифікатором та значенням - інформація про ідентифікатор. В кожній реалізації будови графа зв'язку виконується аналіз оголошення класу та інтерфейсів, однак відрізняються аналізовані дочірні вершини.

Для побудови зв'язку узагальнення необхідно проаналізувати лише вершини, що відповідають правилу граматики extends в оголошеннях класів та інтерфейсів.

Розглянемо конкретний приклад. З лістингу 3.1 програми мовою Java можна отримати граф зображений на рис. 3.1.

```
private interface MainInterface { }
private interface AnotherInterface { }
private interface SubInterface extends MainInterface { }
public interface Gen2 extends MainInterface, AnotherInterface { }
```

Лістинг 3.1. Приклад реалізації зв'язку узагальнення між інтерфейсами MainInterface та SubInterface, Gen2 і MainInterface, AnotherInterface та Gen2.

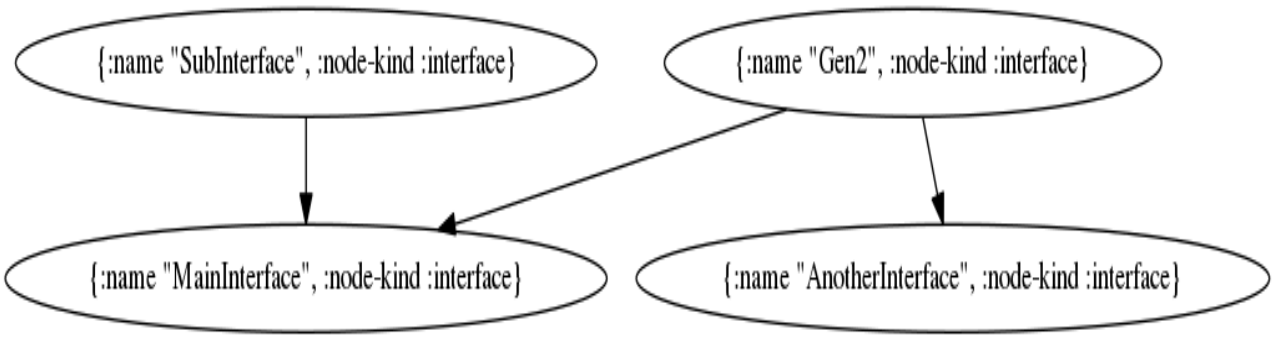


Рисунок 3.1. Граф, що показує зв'язок узагальнення між інтерфейсами з лістингу 3.1

Для побудови зв'язку реалізації необхідно проаналізувати лише вершини, що відповідають правилу граматики implements в оголошеннях класів та інтерфейсів.

Побічним результатом роботи системи при аналізі лістингу 3.2 буде граф зображений на рис. 3.2.

```
private interface Test { }
private interface AnotherTest { }
private class OneImplementation implements Test { }
public class TwoImplementation implements Test, AnotherTest { }
```

Лістинг 3.2. Приклад вихідного коду зі створенням звязків реалізації між інтерфейсом Test та класом OneImplementation, між інтерфейсами Test, AnotherTest та класом TwoImplementation

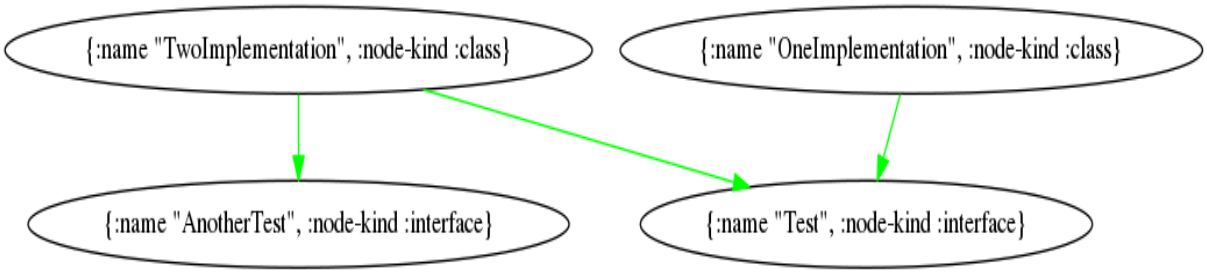


Рисунок 3.2. Граф, що показує зв'язок реалізації між інтерфейсами та класами з лістингу 3.2.

Для побудови зв'язку залежності необхідно аналізувати типи, які мають атрибути класу, локальні змінні всередині методів, об'єкти, що повертаються в методах. Якщо даний тип не належить до пакету java.lang, тобто не є вбудованим для мови програмування, то додається ребро між аналізованим класом та вершиною, яка представляє використаний тип.

З лістингу 3.3. програми мовою Java буде отримано граф зображений на рис. 3.3.

```
private class Test { }
private interface TestConstructorI { }
public class Dep1 {
   public Dep1 (TestConstructorI test) { }
   public void foo (int par, Test parameter) { }
}
```

Лістинг 3.3. Приклад зв'язку залежності між класом Dep1 та інтерфейсом TestConstructorI і класом Test.

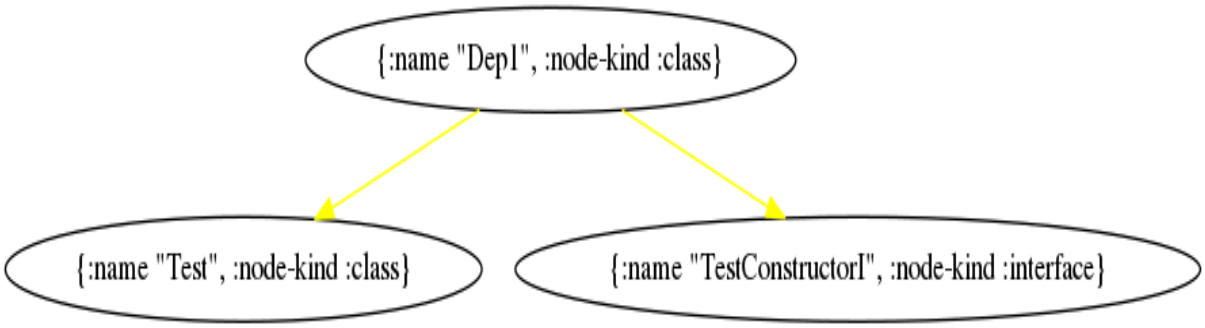


Рисунок 3.3. Граф, що показує зв'язок залежності між одним класом з інтерфейсом та іншим класом з лістингу 3.3

3.4. Інтерфейс користувача

Система має текстовий користувацький інтерфейс. Для запуску необхідно передати два параметри: шлях до шаблону проектування та шлях до системи, в якій буде визначатись використання шаблону проектування.

Програма виконає ланцюжок перетворень шаблону проектування до послідовності графів, які відображають зв'язки між класами, та аналогічну послідовність дій для системи. Створені графи являються вхідними параметрами для способу виявлення використання шаблонів проектування. Результатом роботи є повідомлення про співпадіння, яке складається з

трьох частин: точне (exact), можливе (possible) та відсутнє (absent).

Кожна з частин повідомлення має спільну структуру: набір рядків, де першою частиною є ідентифікатор класу або інтерфейсу з шаблону проектування, символ відповідності (->), ідентифікатор класу або інтерфейсу з аналізованої системи.

Точне співпадіння можливе у випадку, коли було не знайдено жодних протиріч у результуючій відповідності між графом шаблону проектування та системи. Можливе співпадіння показує, наявність протирічь у результаті роботи. Відсутність співпадіння стверджує неможливість наявності шаблону проектування у аналізованому коді.

Відмінність між можливим співпадінням та його відсутністю полягає в тому, що в можливому співпадінні є протиріччя між зв'язками, проте структура ярусів графів - однакова. Відсутність співпадіння можлива у випадку, коли взагалі відсутні певні види зв'язків у проаналізованій системі порівняно з шуканим шаблоном проектування.

Приклад результати роботи системи наведено на рис. 3.4.

```
dmytro@laptop:~/study/clojure/desisg_patterns/diplom$ ./detect /home/dmytr
o/study/clojure/desisg_patterns/diplom/test/diplom/java/design_pattern/Abs
tractFactory /home/dmytro/study/clojure/desisg_patterns/diplom/test/diplom
/java/system/AbstractFactory
exact:
AbstractFactory -> SquadronFactory
AbstractProductA -> Mage
AbstractProductB -> Warrior
Client -> CreateSquadron
ConcreteFactory1 -> OrcSquadronFactory
ConcreteFactory2 -> ElfSquadronFactory
ProductA1 -> OrcMage
ProductA2 -> ElfMage
ProductB1 -> OrcWarrior
ProductB2 -> OrcMage

possible:
--

absent:
--
```

Рисунок 3.4. Результат роботи при точному співпадінні шаблону

проектування з аналізованою системою

Для ілюстрації роботи було використано шаблон “Абстрактної фабрики” (Abstract Factory) та системи, що є моделлю використання в ігровій розробці, яка використовує вищезгаданий шаблон проектування.

Виявлення шаблону використання шаблону «Будівник» (Builder) для тієї ж системи виведе повідомлення про можливе співпадіння (рис. 3.5), оскільки в шуканому шаблоні присутні зв'язки залежності та узагальнення з аналогічної структурою, як і в абстрактній фабриці.

Відповідність в можливому співпадінні є не повною. Оскільки спосіб розрахований на зупинку процесу пошуку у разі виявлення першої невідповідності між деревоподібними представленнями вихідного коду та шаблону проектування, така помилка зіставлення може виникнути на будь-якій ітерації. Тому результат роботи для аналогічних систем може бути різним.

Рисунок 3.5. Результат роботи при можливому співпадінні шаблону проектування з аналізованою системою

Для наведеної системи можливість співпадіння можлива у разі відсутності одного з видів зв’язків: успадкування або залежності. Шаблон проектування одинак не використовує узагальнення, тому в результаті усі

частини повідомлення будуть пустими, окрім розділу про відсутність збігу. Результат наведено на рис. 3.6.

Рисунок 3.6. Результат роботи при відсутності співпадіння шаблону проектування з аналізованою системою

3.5. Підхід до тестування системи

Техніка тестування включає як процес пошуку помилок або інших дефектів, так і випробування програмних складових з метою оцінки. При тестуванні оцінювались критерії:

- відповідність вимогам до системи, що розробляється;
- коректність роботи модулів;
- виконання функцій за прийнятний час;

Компоненти системи було протестовано за допомогою методу структурного тестування, або тестування «білого ящика», на основі потоку даних. Дана методика забезпечує аналіз вихідного коду програми. Існує три різновиди структурного тестування [13]:

- на основі потоку керування програми;
- на основі потоку даних;
- мутаційне тестування.

При використанні першого типу тестується логіка програми, що

представлена у вигляді графа керування: вершинами є оператори, а гілками - переходи між ними.

При тестування на основі потоку даних увага приділяється взаємозв'язкам між змінними. Виділяються вершини, у яких змінна ініціалізується та в яких використовується, і вивчаються переходи й взаємозв'язки між такими вершинами.

Мутаційне тестування полягає у внесенні помилок у вихідний код програми та порівняння роботи вихідної програми та програми мутанта. Оскільки здійснити вичерпне структурне тестування вкрай важко, необхідно вибрати такі критерії його повноти, які допускали б їхню просту перевірку й полегшували б цілеспрямований підбір тестів [13].

Для виконання тестів використовувались вбудовані можливості мови програмування, а саме модуль clojure.test та можливості системи для управління проектом Leiningen. Для розширення наборів тестування необхідно лише у відповідній теці test оголосити тест за допомогою ключове слово deftest та додати перевірки. Запуск та перевірка коректності створеного набору тестів виконується за допомогою виконання команди lein test в директорії проекту.

3.6. Тестування системи

Кожен з модулів системи був покритий певними тестовими випадками.

Для перевірки коректності побудови абстрактного синтаксичного дерева було створено декілька файлів з вихідним кодом мовою Java. Кожен з файлів перевіряє коректність роботи певного правила з граматики. Такий підхід необхідний для того, щоб полегшити налагодження роботи синтаксичного аналізатора.

Даний модуль має найбільшу схильність до спричинення зупинки

роботи усієї системи при різних вхідних даних. Також тестові набори допомагають зрозуміти які типи вихідного коду можуть бути сконвертовані до абстрактного синтаксичного дерева.

Також було покрито декількома тестовими випадками модуль, який відповідальний за побудову зв'язків діаграми класів між класами та інтерфейсами з абстрактного синтаксичного дерева. Тестовими випадками покриті такі типи зв'язків:

- узагальнення (generalization);
- реалізація (realization);
- залежність (dependency).

Для типу зв'язку узагальнення створено декілька тест-функцій. Основна функція для тестування збирання зв'язку є test-gener-match-decl. Вона перевіряє коректність роботи виявлення в абстрактному синтаксичному дереві правил classDeclaration та interfaceDeclaration з викликом відповідних функцій для побудови вершин майбутнього графу. Створено два тестових випадки:

- позитивні дані. Тестова функція викликається двічі для двох правил відповідно. На виході повинні отримати список словників, які представляють вершини майбутнього графа з ключем у вигляді структури даних з ім'ям вершини (назва класу або інтерфейсу) та її типом (клас або інтерфейс), а значення - список вершин, які з'єднані з цією вершиною.
- Негативні дані: на вхід подається абстрактне синтаксичне дерево з відсутніми правилами classDecalartion та interfaceDeclaration. На виході повинні отримати аналогічний список словників, де значеннями являються пусті списки.

Існує декілька спільних функцій, для двох типів зв'язку: узагальнення і реалізації, оскільки вершини для них будуються на основі однакових

правил classDeclaration та interfaceDeclaration. Основною задачею є перевірка коректності збирання назв користувацьких типів після ключових слів extends та implements в оголошенні класів та інтерфейсів.

Для перевірки створено декілька тестових випадків з наявністю або відсутністю ключових слів для успадкування або розширення певного класу іншим класом та інтерфейсом відповідно.

- відсутність ключових слів extends та implements. На вхід функції передається вершина абстрактного синтаксичного дерева, що відповідає правилу classDeclaration з відсутніми ключовими словами. На виході отримуємо список словників, з пустими списками яв якості значень, що відповідає відсутності зв'язку узагальнення або реалізації для цього класу з іншими класами.
- Наявне ключове слово extends. Тестова функцій викликається двічі: з поданням вершини абстрактного синтаксичного дерева, що відповідає правилам classDeclaration або interfaceDeclartion. Для обох випадків результат очікується однаковим - список словників з відповідними значеннями ідентифікаторами класів або інтерфейсів.
- Наявне ключове слово implements. За створеною граматикою, такий випадок можливий лише для інтерфейсів, тому на вхід і подається вершина абстрактного синтаксичного дерева, що відповідає правилу interfaceDeclartion. Очікуваний результат є аналогічним до попереднього випадку.
- Наявне ключові слова extends та implements. Такий випадок можливий лише для оголошення класу, тому на вхід подається вершина абстрактного синтаксичного дерева, що відповідає правилу classDeclaration. Очікуваний результат залежить від параметру, який передається до викликаної функції. Програміст явно вказує для якого ключового слова хоче отримати список ідентифікаторів.

Для типу зв'язку реалізації створено декілька тест-функцій. За

граматикою мови Java даний зв'язок можливий лише для інтерфейсів, тому тестові випадки перевіряються лише для оголошення інтерфейсів. Вони створені для перевірки коректності роботи з відсутністю та наявністю ключового слова implements. Назва тестової функції - test-impl-match-decl.

- наявне слово implements. На вхід подається вершина абстрактного синтаксичного дерева, що відповідає правилу оголошення класу. На виході повинні отримати список словників, які представляють вершини майбутнього графа з ключем у вигляді структури даних з ім'ям вершини та її типом (інтерфейс), а значення - список вершин, які з'єднані з цією вершиною.
- Негативні дані: на вхід подається абстрактне синтаксичне дерево з відсутнім правилом classDecalartion. На виході повинні отримати список словників зі значеннями у вигляді пустих списків.

Для типу зв'язку залежності основною функцією для тестування є test-get-dependency-node. Вона перевіряє коректність роботи виявлення у вершинах абстрактного синтаксичного дерево, які відповідають правилам classDeclaration та interfaceDeclaration зв'язку залежності з іншими користувацькими типами даних. Для перевірки наявності зв'язку необхідно перевіряти усі оголошення локальних змінних в тілі методів, атрибутів та формальних параметрів, що зазначені в сигнатурах методів, тому для кожного випадку створено власний тест-кейс.

Для перевірки встановлення залежності через використання локальних змінних створено два тестових випадки:

- оголошення локальних змінних із користувацьким типом даних. На вхід подається вершина абстрактного синтаксичного дерева, що відповідає правилу methodDeclaration. Зв'язок залежності повинен з'явитися між класом, в якому оголошено метод з локальною змінною та класом або інтерфейсом, який використовує змінна. На виході отримано список вершин майбутнього графа, які мають

структуру словника.

- Оголошення локальних змінних із вбудованим типом даних. На вхід подається аналогічна вершина абстрактного синтаксичного дерева, всередині якого присутні оголошення змінних лише з вбудованими типами даних. На виході отримано список словників, з пустими списками в якості значень словників.

Тест-кейси для тестування коректності виявлення зв'язків у сигнатурах методів та атрибутах класів мають аналогічні випадки: з використанням користувацьких типів даних та з оголошенням параметрів, які мають вбудовані типи даних.

Модуль пошуку відповідності між графами покритий тестами для того, щоб показати представлені можливості. Покрито тестами функції для розбиття дерева на яруси, пошуку відповідності між ярусами двох дерев, поєднання результатів побудови для ярусів. Також покрито тестами композицію вище перерахованих функцій, тобто створено інтеграційний тест для перевірки коректності роботи модуля в цілому.

4. ПОРІВНЯЛЬНИЙ АНАЛІЗ ЕФЕКТИВНОСТІ СПОСОБІВ

4.1. Задачі для тестування

Для виконання порівняльного аналізу ефективності існуючих способів Ульмана, Чанга та представленого модифікованого способу повного перебору, що були представлені у розділі 2, було використано два види задач: пошук ізоморфного піддерева у дереві згенерованих синтетично та пошук ізоморфного дерева, що представляє собою шаблон проектування абстрактна фабрика, у лісі отриманому в результаті обробки систем з відкритим вихідним кодом мовою програмування Java.

Розглянемо більш детально спосіб для генерації дерев. Існує два основних параметра за якими задається граф: кількість вершин та матриця ребер між вершинами. З властивостей дерева випливає, що кількість вершин також задає і кількість ребер. Нехай v це кількість вершин у дереві T, тоді $v-1$ – кількість зв'язків у дереві. Для дерева специфічними є поняття глибини d та ширини b, де глибина – довжина шляху від кореня до вузла, ширина – потужність множини вузлів дерева на одному рівні [9]. Саме за цими параметрами відрізняються згенеровані дерева. На рисунку 4.1 зліва зображено дерева з мінімальною шириною 1 та глибиною d, а справа – з максимальною глибиною, що дорівнює двом та шириною b.

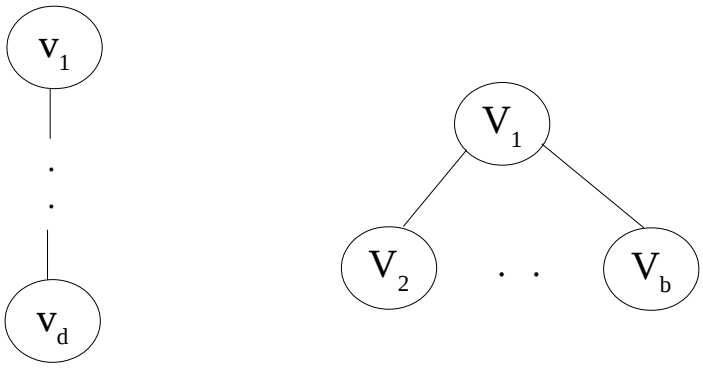


Рисунок 4.1. Дерева з заданою глибиною d (зліва) та шириною b (справа)

Спосіб для генерації псевдовипадкових дерев може бути представлений наступним алгоритмом 3.

Алгоритм 3. Генерація тестових дерев

Вхідні дані: глибина та ширина дерева d і b відповідно

створити кореневу вершину r

$prev \leftarrow r$

for для кожного $i \leqslant d$ **do**

for для кожного $j \leqslant b$ **do**

створити вершину v

з'єднати $prev$ з v

end for

$prev \leftarrow v$ за псевдовипадковим індексом від 1 до b включно

end for

Результатом роботи алгоритму 4.1 буде гарантована наявність ізоморфного піддерева з висотою d_1 та шириною b_1 за умови, що виконуються дві умови: глибина ізоморфного дерева менше або дорівнює глибині дерева в якому буде відбуватись процес виявлення ізоморфного піддерева ($d_1 \leqslant d$) та аналогічна умова для ширини двох дерев ($b_1 \leqslant b$). Твердження виконується оскільки структура для випадку $d_1 \leqslant d$ і $b_1 \leqslant b$ структура дерева з глибиною d та шириною b буде повторювати структуру ізоморфного дерева. Використання псевдовипадкового значення необхідне для того, щоб виключити ситуацію з точним співпадінням піддерева. Для того, щоб гарантувати відсутність ізоморфного піддерева достатньо зменшити параметр глибини або ширини порівняно з відповідними значеннями дерева-шаблону з яким виявляється

відповідність.

Іншим видом задачі для порівняльного аналізу є використання справжніх проектів з відкритим вихідним кодом мовою програмування Java. У якості проектів для порівняння було обрано навчальний проект який містить в собі шаблони проектування представлені у книзі [4] та інші. Цей проект було обрано для того, щоб отримати реалістичні деревоподібні структури зв'язків діаграми класів з низьким рівнем спадкування. Рівень спадкування — це максимальна глибина дерева, що отримане в результаті побудови зв'язків узагальнення. Значення від нуля до чотирьох вважається задовільним для підтримки вихідного коду, що було розроблено з використанням об'єктно-орієнтованої парадигми [15]. Тим не менш, це значення ніяк не обмежене компілятором мови програмування Java, тому можливе написання систем, у яких рівень спадкування досягає більшого значення.

Іншими проектами для порівняння були наступні системи з відкритим вихідним кодом:

- Java Swing — бібліотека для створення графічного інтерфейсу;
- java.io — стандартний пакет для роботи з вхідними та вихідними даними;
- java.net — стандартний пакет для створення додатків з використанням мережевої взаємодії.

Swing — це інструментарій для створення графічного інтерфейсу у додатках написаних мовою програмування Java. Це частина бібліотеки базових класів Java (Java Foundation Classes), яка створена компанією Oracle. Бібліотеку було розроблено для забезпечення більш гнучкого інтерфейсу з більшою кількістю функціональних частин порівняно з інструментарієм AWT. Основною особливістю цієї бібліотеки є реалізація без вихідного коду, що залежить від платформи на якій буде виконаний. Це є його і перевагою, оскільки бібліотека працює на будь-якій платформі, що

має підтримку середи виконання мови програмування Java, але в той же час є його недоліком через відносно повільну роботу.

Пакети java.io та java.net входять до JDK (Java SE Development Kit) починаючи з версії 1.0, є одними з основних пакетів.

4.2. Порівняння роботи способів за часом виконання

Задача розпізнавання ізоморфного піддерева у іншому дереві залежить від двох вхідних структур даних, кожна з яких залежить від двох параметрів: ширина та глибина. Для повноти та наочності порівняльного аналізу, одна деревоподібна структура даних повинна бути незмінною, а інша — має змінюватися за одним з двох параметрів.

Нехай дерево за яким буде визначатись ізоморфізм піддерев, буде задано десятьма вузлами, де на кожному рівні, окрім кореневого, буде три вершини та з висотою, що дорівнює чотирьом. Приклад зображення такого дерева наведено на рис. 4.2.

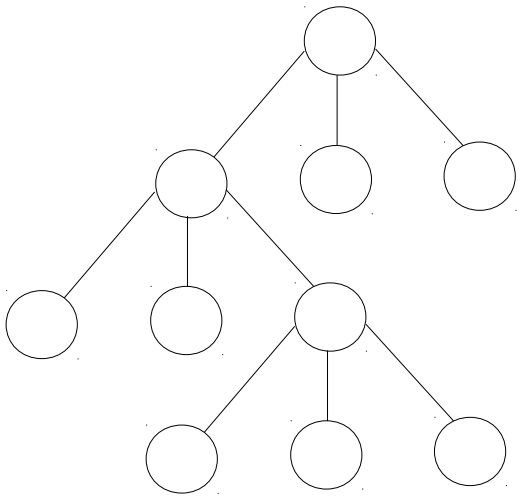


Рисунок 4.2. Дерево-шаблон для виявлення ізоморфізму

Для отримання першого графіку, дерево в якому буде відбуватись пошук ізоморфного піддерева буде збільшуватися за параметром глибини. Дерево по якому буде відбуватись пошук піддерев буде мати ізоморфний

граф для відображення поведінки способів у найкращому випадку. Для цього, кількість вершин буде збільшуватися від 10 до 100 за рахунок збільшення кількості ярусів. На рис. 4.3 зображено залежність часу виконання процесу визначення ізоморфного піддерева від глибини дерева по якому відбувається пошук. Для кожного з графіків буде зображено зростання значення параметру глибини або ширини по осі абсцис, а зростання часу у секундах по осі ординат.

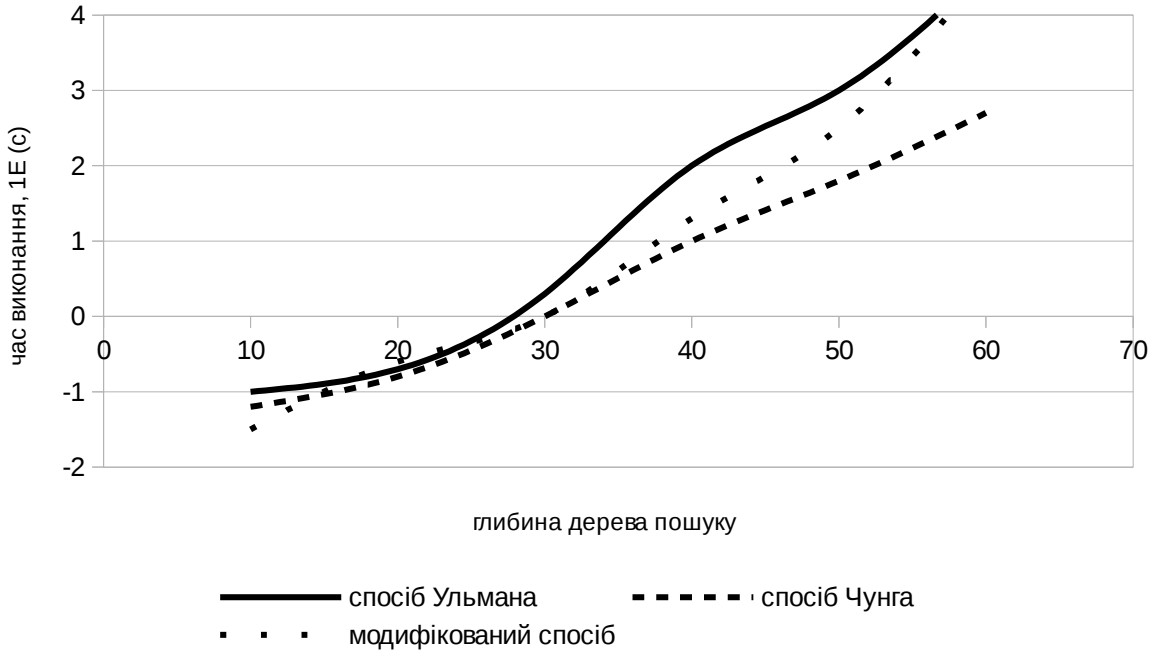


Рисунок 4.3. Зростання кількості вершин за рахунок максимальної глибини

З графіку на рис. 4.3 можна зробити висновок, що модифікований спосіб надає змогу розпізнавати ізоморфне дерево швидше за розглянуті альтернативи при стабільній ширині дерева 3 у якому відбувається пошук тільки при кількості вершин менших за 30. Після цього, поведінка зростання часу модифікованого способу розпізнавання подібності дерев аналогічно до способу Ульмана, за рахунок виконання подібних операцій перестановок. Процедура, що зменшує кількість одиниць у рядках не

проявляє свою ефективність, оскільки кількість одиниць на кожному з рядків однакова і дорівнює трьом через константне значення ширина. Кожна вершина має лише три вершини-дитини і матриця суміжності вершин буде збільшуватись за рахунок збільшення параметри глибини. Спосіб Чунга дає можливість вирішити задачу при збільшені кількості вершин дерева, в якому відбувається пошук ізоморфного піддерева.

Розглянемо випадок, коли у шуканому дереві відсутнє ізоморфне піддерево до заданого. Графік зростання часу роботи трьох способів подано на рис. 4.4. Час роботи способів буде перевірено на такому ж дереві, що використовується в якості шаблону. У дереві в якому відбувається пошук зменшено кількість вершин для того, щоб визначення ізоморфізму давало негативний результат.

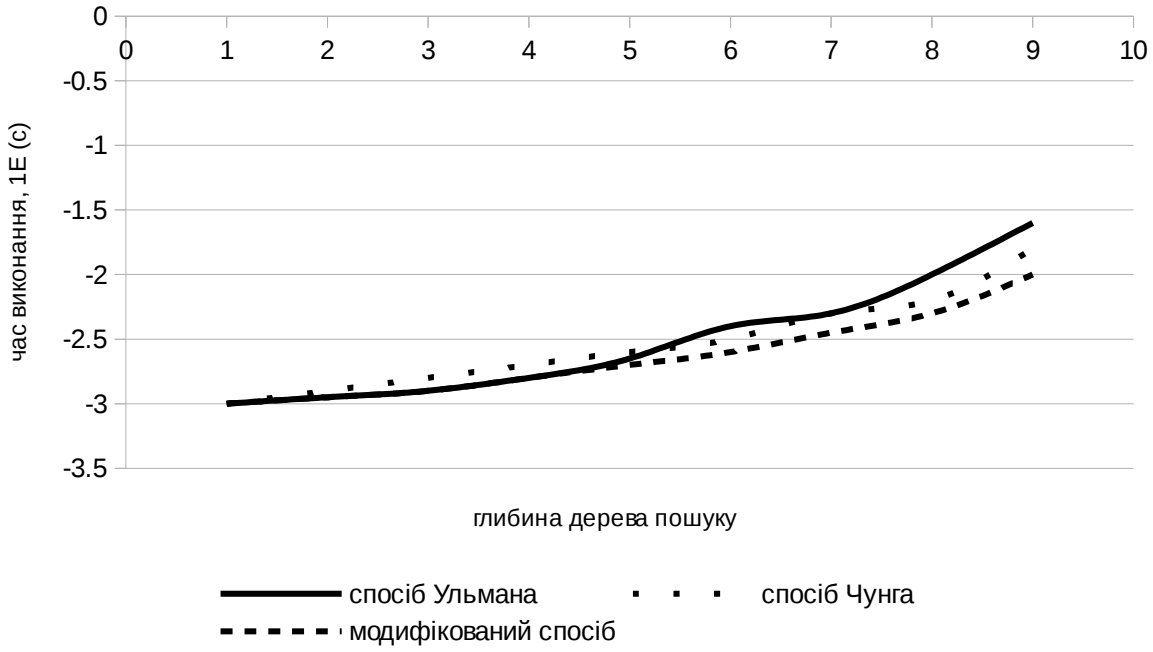


Рисунок 4.4. Час виконання у разі відсутності ізоморфного підграфу

З рис. 4.4 можна зробити висновок, що час роботи модифікованого способу нижчий порівняно з іншими способами. Цього досягнуто за рахунок додаткових перевірок за критерієм глибини на етапах побудови

можливих варіантів для ізоморфних піддерев. Аналогічні умови представлені також в способі Чунга, через це ми бачимо приблизно однакові результати, порівнюючи з модифікованим способом повного перебору. Спосіб Ульмана буде виконувати генерацію додаткових матриць без необхідного аналізу вхідних даних, через це ми бачимо найгірші показники саме в цього способу.

Наступним набором вхідних даних для аналізу часу виконання алгоритмів є зміна параметри ширини вхідного дерева для розпізнавання ізоморфних піддерев. Параметри вхідного дерева для якого буду виявлені ізоморфні дерева також будуть змінені. Глибина цього дерева буде дорівнювати двом, тобто дерево буде складатися з кореня та його вершин-дітей. Ширина дерева буде дорівнювати п'яти вершинам. Ширина дерева буде збільшуватись відповідно від п'яти до двадцяти.

На рис. 4.5 зображено графік зростання часу виконання процесу виявлення ізоморфного піддерева залежно від зростання ширини дерева, у якому відбувається пошук.

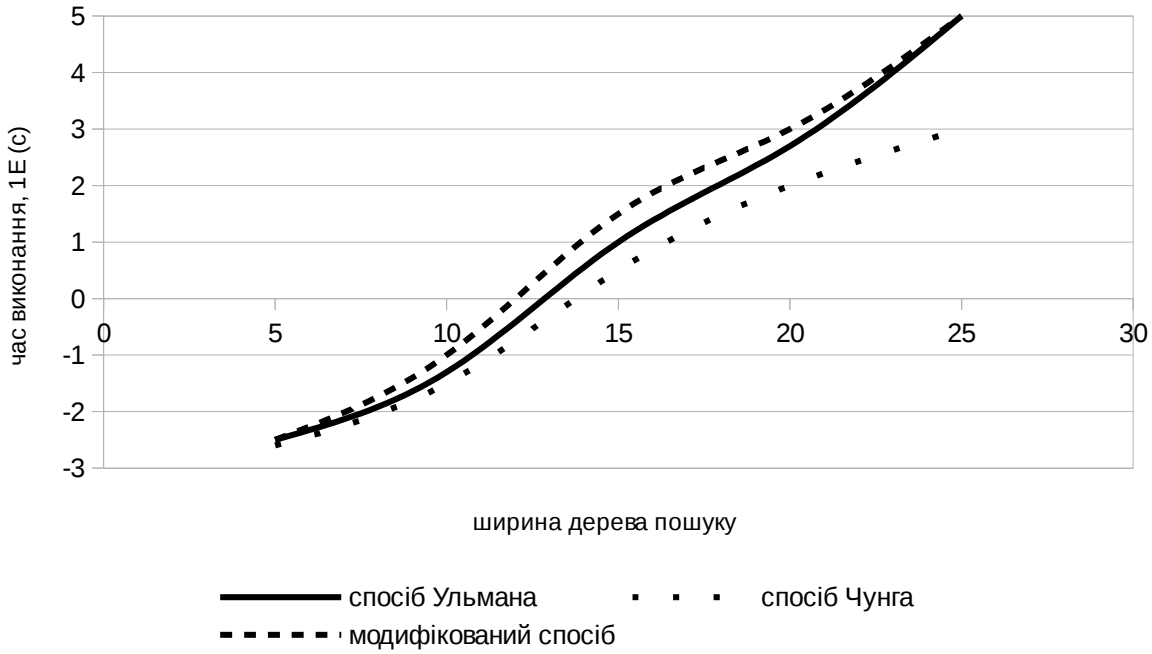


Рисунок 4.5. Зростання кількості вершин за рахунок максимальної ширини

При збільшенні кількості вузлів дерева у якому відбувається пошук ізоморфного підграфа за рахунок збільшення кількості дітей на кожному з ярусів, час роботи виконання модифікованого способу розпізнавання подібності дерев та спосіб Ульмана мають схожу поведінку. Це пояснюється тим, що в обох способах схожим чином відбувається перебір ребер на одному рівні для виявлення ізоморфного піддерева. Проте спосіб Ульмана дає кращі результати за рахунок виконання процедури заміни одиниць на нулі у матриці, що відображає зв'язки між вузлами. Однак таке зменшення можливих варіантів не дало суттєвого зменшення виконання оскільки метою системи є розпізнавання використання шаблонів проектування, що означає пошук усіх можливих варіантів ізоморфних піддерев. Для досягнення цієї мети, було видалено безумовні переходи на переривання роботи у розглянутому способі Ульмана в частині 2.2. Спосіб Чунга показує найкращих результат з розглянутих способів за рахунок окремої обробки термінальних вершин, які у синтетичних варіантах присутні у великій кількості в незалежності від збільшення кількості вузлів за рахунок ширини чи глибини. Проте збільшення нетермінальних вершин за рахунок збільшення ширини дерева в якому відбувається пошук ізоморфних підграфів відображається на швидкодії способу Чунга. Такі висновки можна зробити при порівнянні відповідних графіків на рис. 4.3 та рис. 4.5. З цього можна зробити висновок, що послідовність дій, які виконуються при обході вхідного дерева-шаблону, займає найбільше часу виконання роботи способу.

Для усіх способів розглянуті вхідні дані, у яких відбувається збільшення кількості вершин за рахунок збільшення потужності множини вершин-дітей на кожному рівні, є одним з найгірших випадків оскільки кількість ізоморфних піддерев є дуже високою.

Наступним етапом є порівняння виконання процесу розпізнавання ізоморфних піддерев при збільшені кількості вершин за рахунок параметру

ширини дерева, в якому відбувається пошук. Для такого випадку необхідно розглянути вхідні дерева, що дорівнює ширині від одного до двох. Кількість вершин вхідного дерево у якому буде відбуватись пошук ізоморфних піддерев настільки малий, що час виконання усіх трьох способів приблизно однаковий. Для такого випадку, побудова графіку є недоцільною, оскільки графіки залежності часу виконання від кількості вершин накладаються.

Для виконання подальшого аналізу необхідно порівняти час виконання роботи розглянутих трьох способів за рахунок збільшення вершин у дереві за яким відбувається пошук ізоморфного піддерева, тобто дерева-шаблону. Збільшення вершин буде виконуватись аналогічним чином: за глибиною та шириною. У наступних синтетичних результатах константним буде дерево у якому буде відбуватись пошук ізоморфних піддерев. Кількість вершин та значення глибини та ширини буде змінюватися відповідно до змін значення відповідних параметрів у дереві, яке буде слугувати шаблоном у процесі розпізнавання ізоморфних піддерев.

Нехай дерево у якому буде відбуватись пошук ізоморфного піддерева буде мати ширину, що дорівнює 3, та глибину, що дорівнює 30. Дерево за яким буде визначатись ізоморфізм піддерев, буде мати незмінну ширину, яка дорівнює значенні ширини графу в якому буде відбуватись пошук ізоморфного піддерева. Максимальна глибина дерева-шаблону буде збільшуватись від 10 до 30. Значення глибини дерева за яким буде визначатись ізоморфізм не перевищує значення глибини іншого вхідного дерева для того, щоб отримати результати часу виконання трьох способів для випадку наявності ізоморфного піддерева. На рис. 4.6 зображено результати виміру часу виконання процесу визначення ізоморфних піддерев.

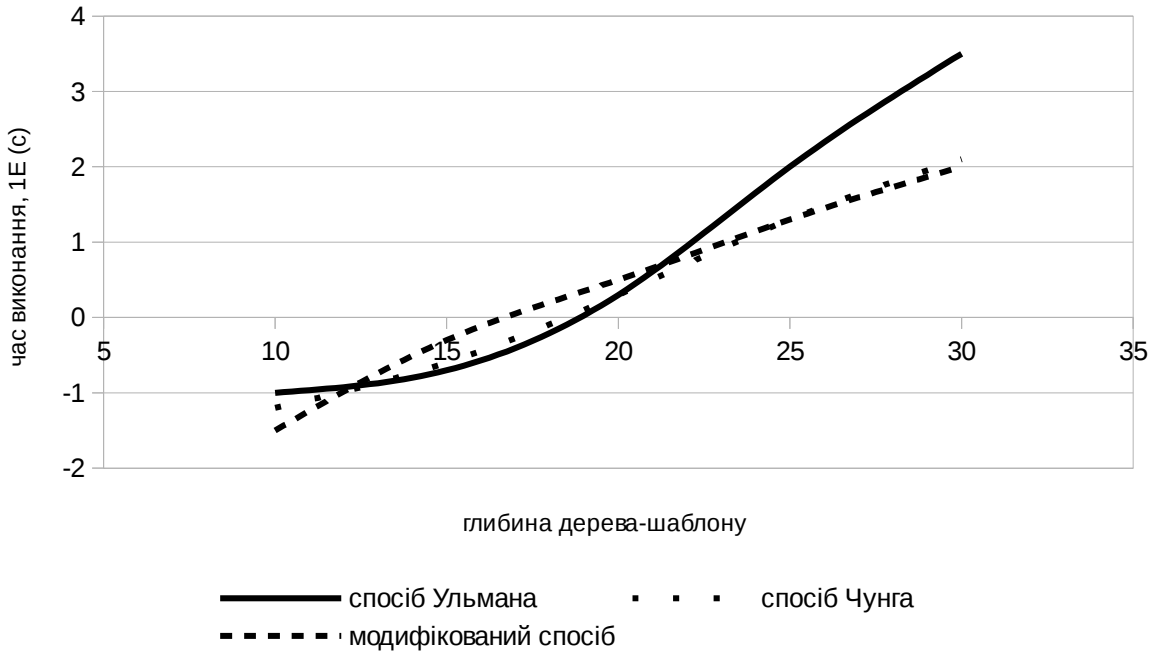


Рисунок 4.6. Час роботи при збільшенні глибини дерева-шаблону

Час виконання роботи способу Ульмана аналогічний до часу виконання випадку при зміні глибини дерева в якому відбувається пошук ізоморфного піддерева. Це можна пояснити аналогічними операціями, що виконуються при роботі з матрицями зв'язків між вершинами дерева за яким робиться пошук ізоморфні піддерева та між вершинами дерева у якому розпізнаються ізоморфні графи. Така поведінка пояснюється тим, що спосіб розрахований на роботу з будь-яким графом, а не деревом, як частковий випадок графу. Загальний випадок може мати цикли та ребра, які з'єднують будь-які дві вершини. Час виконання способу Чунга показує збільшення часу виконання при збільшенні кількості вершин дерева-шаблону. Така поведінка може бути пояснена тим, що при збільшені розміру дерева-шаблону збільшується кількість кроків у найглибшому циклі способу Чунга. Тим не менше, час виконання розпізнавання ізоморфних піддерев менший порівняно з модифікованим способом за рахунок наявності перевірок для дострокового завершення роботи у випадках збільшення параметру глибини дерева-шаблону. Час роботи

модифікованого способу не збільшується суттєво при збільшені кількості вершин за рахунок аналогічних перевірок, однак витрачається більше часу порівняно з роботою способу Чунга, за рахунок більшої кількості перестановок.

Надалі варто розглянути поведінки трьох способів для випадку, коли не існує ізоморфних піддерев у дереві, за рахунок збільшення глибини дерева з яким відбувається порівняння для розпізнавання ізоморфних піддерев. Вхідні дані для аналізу залишаються аналогічними до попереднього випадку, за виключенням інтервалу збільшення параметра глибини дерева-шаблону. Для гарантування відсутності ізоморфних піддерев необхідно згенерувати дерево з параметрами глибини від 31 до 50. Графік залежності часу роботи трьох способів від збільшення параметру глибини дерева-шаблону зображено на рис. 4.7.

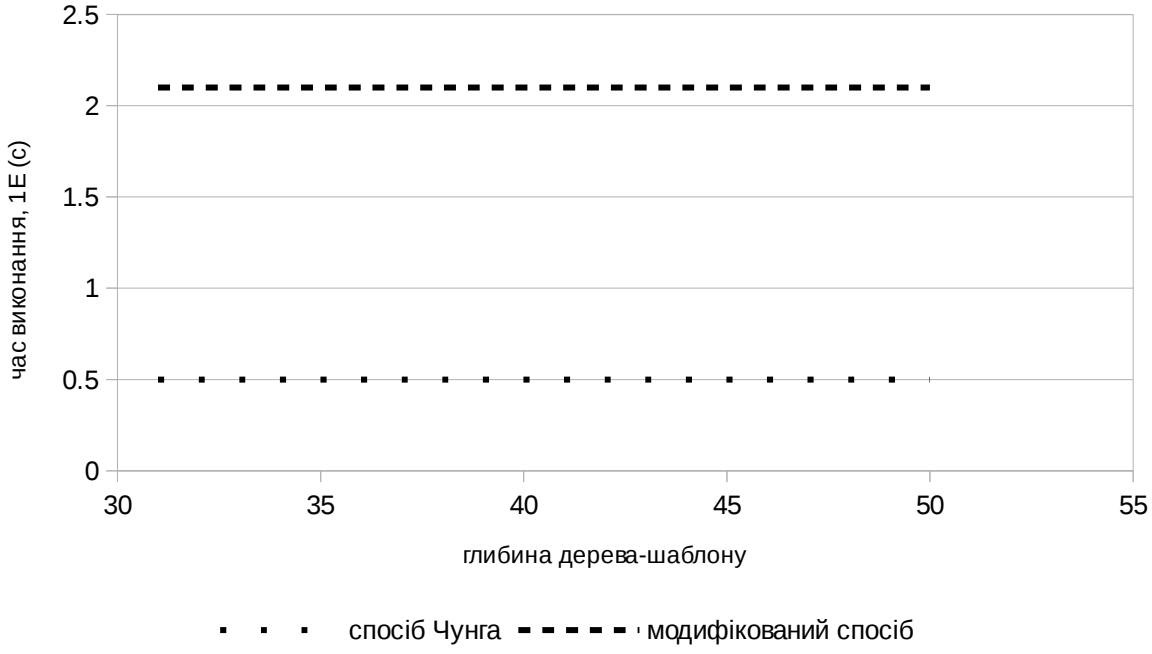


Рисунок 4.7. Залежність часу виконання при відсутності ізоморфізму

Варто зазначити, що у результатах представлених на рис. 4.7 відсутній спосіб Ульмана через те, що час його виконання був занадто

великий. Спосіб Чунга проявляє себе з найкращої сторони і показує менший час витрачений для виявлення ізоморфних підграфів (рис. 4.6). Також варто відзначити константний час виконання способу Чунга незважаючи на збільшення кількості вершин у дереві за яким відбувається розпізнавання ізоморфних піддерев. Така поведінка була досягнута за рахунок перевірок на глибину для вершин, що порівнюються. Не витрачається час виконання на процес побудови біграфів, які необхідні для визначення ізоморфності, оскільки можна стверджувати, що ізоморфні підграфи відсутні, порівнюючи параметр глибини. Модифікований спосіб не має у собі такої перевірки, тому час його роботи збільшується. Час виконання однаковий при збільшенні кількості вершин у дереві-шаблону, оскільки паралельний обхід дерев завершується достроково за рахунок константної кількості вершин у дереві з яким відбувається порівняння часу.

Наступним етапом є порівняння часу роботи трьох способів для випадку, коли глибина дерева-шаблону залишається незмінною, а збільшується параметр ширини. Дерево в якому відбувається пошук ізоморфних піддерев буде змінена за рахунок збільшення ширини та зменшення висоти порівняно з двома попередніми прикладами вхідних даних. Для подальшого аналізу буде використовуватися дерево у якому відбувається пошук ізоморфних дерев з параметрами ширини, що дорівнює 8 та глибини, що дорівнює 10.

Синтетичні дані в яких буде змінюватися ширина дерева, яке виступає шаблоном для пошуку ізоморфних піддерев, можуть бути розділені на дві частини аналогічно до минулих результатів аналізу: з наявністю ізоморфних піддерев та їх відсутністю.

Розглянемо випадок, коли ізоморфні піддерева присутні у вхідному дереві. Для такого випадку, ширина дерева з яким буде визначатись ізоморфність буде змінювати кількість вершин за рахунок зміни параметри ширини від 3 до 8, а глибина піддерева буде однаковою і дорівнюватиме 4

вершинам. Глибина дерева для виявлення подібності обрана меншою за глибину дерева в якому будуть виявлені ізоморфні піддерева для того, щоб збільшити кількість можливих варіантів, що задовольняють умові ізоморфізму та як наслідок, час виконання процесу розпізнавання. На рис. 4.8 зображено результати для виявлення залежності часу роботи від значення ширини у дереві-шаблоні.

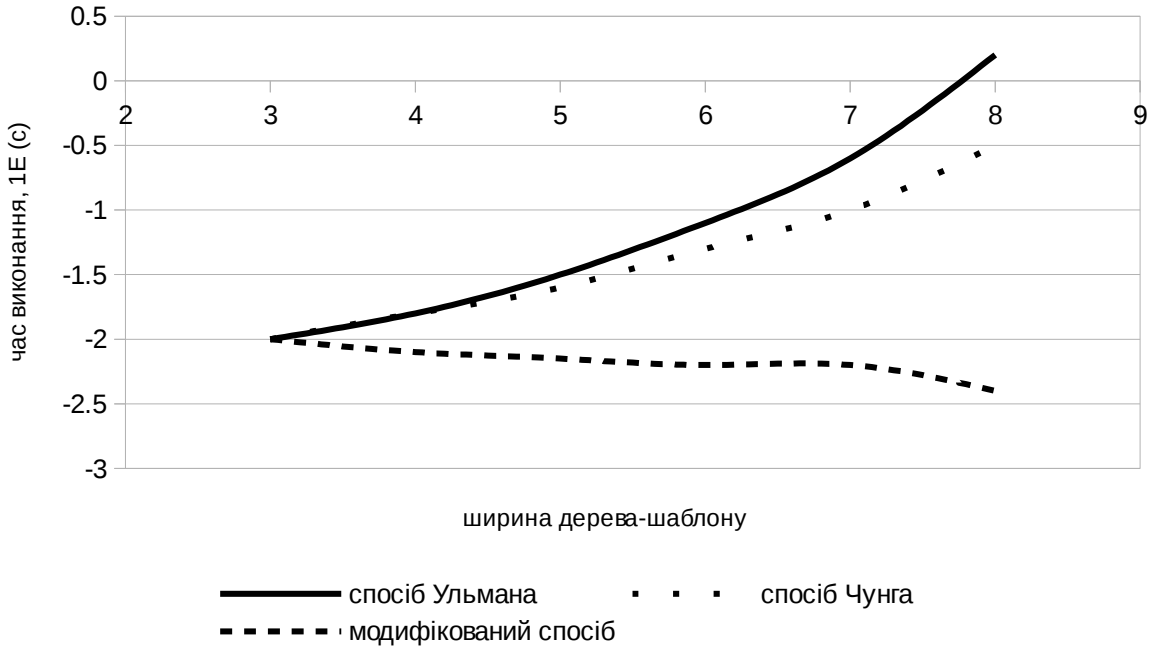


Рисунок 4.8. Залежність часу виконання при збільшені ширини дерева-шаблону

Для розглянутого випадку, модифікований спосіб показує поступове зменшення виконання часу при збільшенні вершин на одному ярусі дерева для якого виконується пошук ізоморфних піддерев, але потім відбувається аналогічне зростання. Це пов'язане з тим, що представлений спосіб для виявлення подібності дерев при збільшені кількості виконує меншу кількість перестановок для кожного ярусу. При досягненні результату, коли ширина дерева-шаблону дорівнює ширині дерева в якому відбувається пошук ізоморфного піддерева є найкращим варіантом для представленого

модифікованого способу для розпізнавання подібності дерев. Розглянутий спосіб Чунга виконує побудови множин для подальшого пошуку ізоморфних піддерев для кожної нетермінальної вершини. При збільшені кількості нетермінальних вершин, кількість таких побудов також зростає. Додаткові умови для перевірки необхідності побудови таких множин відсутні, через це ми можемо бачити поступове зростання часу роботи розглянутого способу. Спосіб Ульмана показує себе найгірше оскільки для того, щоб визначити ізоморфність піддерев виконуються перестановки усіх можливих варіантів. На відміну від вхідних даних, що були розглянуті вище, спосіб все одно може надати коректний результат за прийнятний час за рахунок процедури заміни одиниць на нулі у матрицях суміжності. Для заданих синтетичних вхідних даних, при збільшені кількості вершин на одному ярусі, збільшується й кількість відповідних одиниць, що позначають зв'язки між вузлом та вершинами-дітьми.

Наступним та останнім етапом для порівняння роботи розглянутих трьох способів є перевірка їх часу роботи для випадку, коли збільшується значення ширини дерева-шаблону, однак у дереві в якому відбувається процес розпізнавання ізоморфних піддерев останніх немає в наявності. Для того, щоб гарантувати відсутність виявлення ізоморфності з піддеревами, необхідно збільшити ширину дерева з яким буде визначатись ізоморфність до значення більшого за задану ширину дерева в якому відбувається пошук. На рис. 4.9 зображено залежність часу роботи від збільшення параметру ширини від 9 до 14.

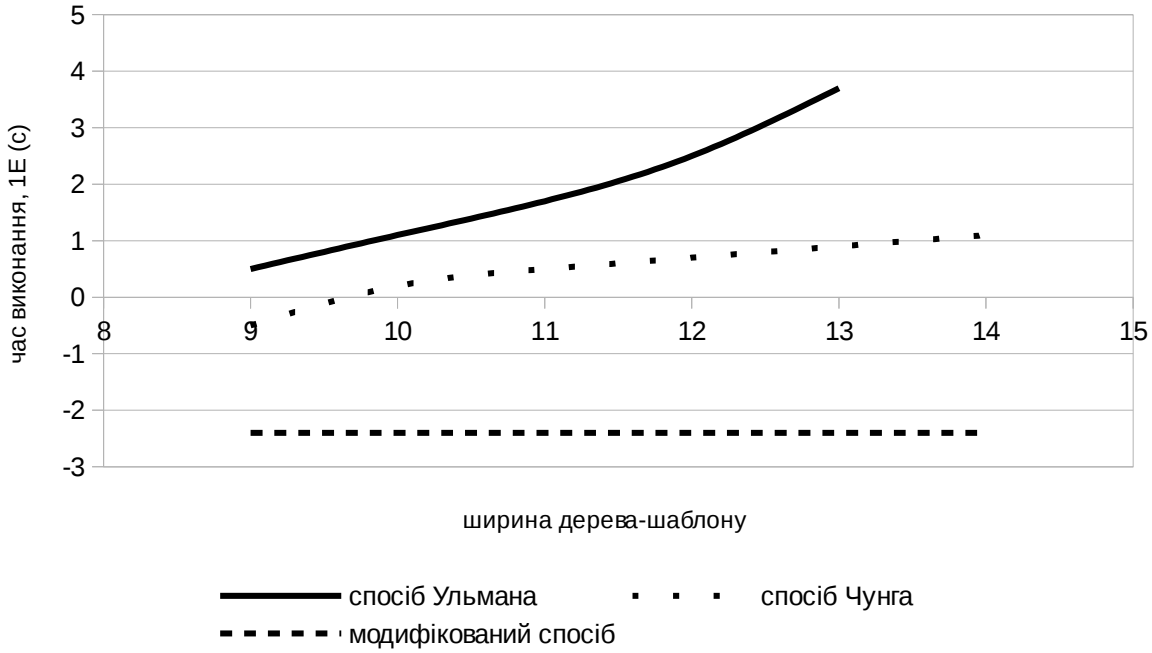


Рисунок 4.9. Залежність часу виконання при збільшені ширини дерева-шаблону для випадку відсутності ізоморфних піддерев

На представленому рис. 4.9 можна побачити, що час виконання модифікованого способу є лінійний за рахунок перевірок на можливі наявність ізоморфних за параметром ширини. Оскільки потужність множини вершин-дітей у дереві, в якому відбувається пошук ізоморфних піддерев завжди менша за потужність аналогічної множини у дереві з яким відбувається процес визначення ізоморфності, модифікований спосіб виявлення подібності не буде виконувати додаткових операцій за виключенням обходу дерева у якому знаходять ізоморфні піддерева. Загалом поведінка лінії графіку залежності часу виконання від параметру ширини схожа на лінію графіку, що зображена на рис. 4.7, для способі Чунга. Обидва випадки були досягненні за рахунок додаткових перевірок для різних характеристик дерев. Спосіб Чунга продовжує показувати лінійний розвиток, що був відображений на рис. 4.8. В свою чергу спосіб Ульмана витрачає багато часу для виконання операцій над матрицями, що

зображують зв'язок між вершинами, оскільки у загальних випадках, граф не має обмежень, що характерні для часткового випадку дерева.

Наступним етапом аналізу є порівняння часу роботи способів при запуску системи для виявлення шаблонів проектування у мові програмування Java з використанням систем з відкритим вихідним кодом.

Першим проектом для перевірки був навчальний проект, який має у собі низький рівень спадкування та гарантовану наявність шуканих твірних шаблонів проектування. Час роботи розглянутих трьох способів відображено у таблиці 4.1. Варто зазначити, що для усіх подальших замірів часу були виключені етапи синтаксичного аналізу, побудови абстрактного синтаксичного дерева та деревоподібної структури даних, що відображає зв'язки діаграми класів, оскільки кожен з етапів не впливає на час роботи кожного з трьох способів.

Таблиця 4.1 складається з трьох колонок, що показують час роботи кожного з трьох способів у секундах, що були розглянути у частинах 2.2 та 2.3. Назви проекту відповідають проектам, що були представлені у частині 4.1.

Назва проекту	спосіб Ульмана	спосіб Чунга	модифікований спосіб
навчальний	235	184	170
Java swing	739	561	632
java.io	458	386	395
java.net	593	435	419

Таблиця 4.1. Результати роботи трьох способів на реальних проектах з відкритим вихідним кодом

Час виконання розпізнавання наявності твірних шаблонів проектування відрізняється від проекту оскільки кожен з них має різну

складність за рівнем спадкування, використання зв'язку реалізації у вигляді інтерфейсів та загального зв'язку залежності, що відображає залежність між використанням, створенням одного класу іншим.

4.3. Порівняння часу роботи для синтетичних та реальних вхідних даних

Час виконання розпізнавання ізоморфних дерев на синтетичних та реальних результатах відрізняється за рахунок того, що реальні структури даних, що були отримані після процесу синтаксичного аналізу та побудови деревоподібних структур даних, відрізняються від синтетично згенерованих. Розглянемо більш детально чим відрізняються структури даних, що були згенеровані та структурами даних, що були отримані після виконання перших двох модулів системи, що були описані у частинах 3.2 та 3.3 відповідно, та узагальнимо поведінку кожного з трьох способів.

При виконанні аналізу вихідного коду мовою програмування Java було отримано велику кількість дерев з коренями, що не поєднані між собою, тобто є лісом. Кожен вид зв'язку був представлений самостійним деревом у вхідному шаблоні проектуванні та окремими лісами у системі з відкритим вихідним кодом, що була проаналізована. Іншими словами, в ході роботи системи було отримано велику кількість дерев з невеликою кількістю вершин порівняно з синтетичними результатами. Для синтетичних результатів збільшувалась кількість вершин за одним з двох параметрів для відображення поведінки кожного з трьох способів для різних структур дерев.

Саме завдяки роздрібленості дерев на яких був запущений процес розпізнавання ізоморфних дерев, спосіб Ульмана завершився за час у секундах, який не перевищив трьох порядків. Варто відзначити, що у разі подання на вхід специфічної системи з відкритим вихідним кодом, яка має

у собі класи та інтерфейси, що продукують деревоподібну структуру даних з великою кількістю вершин та зв'язків, процес розпізнавання може бути завершений за значно більший проміжок часу. Такий висновок було зроблено на основі графіків, що зображені на рис. 4.3 та рис 4.5. При цьому не так важливо за рахунок якого параметру буде відбуватись процес збільшення кількості вершин. В незалежності від збільшення за глибиною чи за шириною, визначення ізоморфних підграфів для розпізнавання використання шаблонів проектування у вихідному коді може займати значення часу з четвертим або навіть вищим порядком.

Спосіб Чунга проявив себе найбільш стабільно серед розглянутих чотирьох проектів. Таку поведінку можна пояснити тим, що спосіб Чунга показав відсутність різкого зростання часу виконання процесу розпізнавання при збільшенні кількості вершин та стабільний час виконання в незалежності від параметру за яким відбувалось збільшення кількості вершин у дереві в якому відбувався пошук ізоморфних піддерев. Найменший час виконання способу Чунга спостерігається, коли кількість вершин у дереві-шаблоні є доволі низькою порівняно з кількістю вершин дерева в якому відбувається виявлення ізоморфізму піддерев. Іншими словами, час роботи способу Чунга буде набагато нижчим порівняно зі способом Ульмана чи модифікованим способом, якщо проект з вихідним кодом представлений множиною деревоподібних структур даних, кількість вершин в яких може перевищувати значення 20.

Представлений модифікований спосіб розпізнавання подібності дерев показує збільшення часу виконання аналогічно до способу Ульману, однак з таблиці 4.1 можна побачити, що для деяких випадків модифікований спосіб проявляє себе краще за спосіб Ульмана та спосіб Чунга. Це можна пояснити тим, що розроблений спосіб показав найменший час виконання серед трьох розглянутих у випадках, коли ширина дерева в якому відбувається пошук ізоморфних піддерев відрізняється від ширини дерева-

шаблону в межах 5 вершин. Також модифікований спосіб проявляє малий час виконання для випадків, коли відсутні ізоморфні піддерева. У випадку реальних проектів, відсутність шаблону є більш розповсюдженим варіантом.

З результатів таблиці 4.1 для порівняння часу виконання розглянутих трьох способів не можна зробити однозначні висновки, оскільки відсутня детальна інформація про структуру дерев, що були подані на вхід для процесу розпізнавання ізоморфних піддерев. Кожен проект генерує специфічні структури даних, для порівняння яких необхідно використовувати вже існуючі інструменти для збору характеристик або створювати нові. Синтетичні результати на відміну від реальних мають суттєву перевагу у передбачуваності їх структури, через це більшу увагу у порівняльному аналізі було приділено саме штучно згенерованим результатам за якими були зроблені висновки щодо переваг та недоліків використання кожного з розглянутих способів.

ВИСНОВКИ

В результаті виконання даної роботи був запропонований модифікований спосіб повного перебору для рішення задачі розпізнавання ізоморфних графів, а також була реалізована система для розпізнавання використання породжувальних шаблонів проектування у вихідному коді мови програмування Java, що працює на основі цього способу. Задача виявлення шаблонів проектування у вихідному коді представлена процесом визначення ізоморфних дерев. Вхідні деревоподібні структури даних, між якими визначалась подібність у якості вершин мають класи або інтерфейси, що представлені у вихідному коді та шаблонах проектування, а у якості ребер використано зв'язки діаграми класів.

Проведено порівняльний аналіз використання модифікованого способу повного перебору з двома іншими способами, що вирішують аналогічну задачу: способом Ульмана, що дозволяє вирішити задачу визначення ізоморфних підграфів у загальному випадку, та способом Чунга, що вирішує таку ж задачу для дерев у якості вхідних даних. У четвертому розділі розглянуто детально випадки, при яких кожен зі способів показує найкращий та найгірший час виконання процесу розпізнавання та зроблено висновки про вплив структури вхідних дерев на час визначення ізоморфних піддерев.

Створена програмна система може бути використана як самостійний продукт або як модуль для проектів аналізу вихідного коду на наявність певних структурних шаблонів.

В майбутньому проект може бути розширений для підтримки інших типів шаблонів проектування. Також можливе створення нових модулів системи для операцій над шаблонами проектування: приведення одного шаблону проектування до іншого, генерація вихідного коду на основі шаблонів проектування, доповнення вихідного коду для точного відображення шаблону.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Фаулер М. Рефакторинг: улучшение существующего кода. — Пер. з англ. — Символ-Плюс, 2003. — 432 с
2. Скотт В. Эмблер. Рефакторинг баз данных: эволюционное проектирование / Скотт В. Эмблер, Прамодкумар Дж. Садаладж. — «Вильямс», 2007. — 368 с.
3. Joshua Kerievsky. Refactoring to pattern / J. Kerievsky - Addison-Wesley, 2004 - 134 p.
4. “GoF”. Design Patterns: Elements of Reusable Object-Oriented Software [text] / The "Gang of Four": Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides — Addison-Wesley, 1994 — p. 164.
5. Леоненков Александр. Самоучитель UML — БХВ-Петербург, 2007 — 237 с.
6. Алан Шаллоуей, Джеймс Р. Тротт. Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию — Пер. с англ. Design Patterns Explained: A New Perspective on Object-Oriented Design. — Вильямс, 2002. — 288 с.
7. Reverse Engineering of Design Patterns from Java Source Code // Nija Shi. Doctoral dissertation. University of California, Davis. August 2007.
8. Юрий Карпов. Теория и технология программирования. Основы построения трансляторов — ВНУ, 2005 — 137 с.
9. S. V. N. Vishwanathan. Kernel Methods: Fast Algorithms and Real Life Applications. — PhD thesis, Indian Institute of Science, Bangalore, India, November 2002.
10. Зыков, А. А. Основы теории графов — Наука, ГРФМЛ, 1987 — 384 с.
11. J.R. Ullmann, An Algorithm for Subgraph Isomorphism — National Physical Laboratory, England, 1976/
12. Ron Shamir, Dekel TsurFaster. Faster Subtree Isomorphism — Theory of

Computing and Systems, 1997 — Proceedings of the Fifth Israeli Symposium.

13. Г. Майерс. Искусство тестирования программ / Гленфорд Майерс, Том Баджетт, Кори Сандлер, 3-е издание — М.: «Диалектика», 2012 — 272 с.
14. Марченко О.І., Лиман Д.М. Модифікований спосіб повного перебору для визначення відповідності дерев. Комп'ютерно-інтегровані технології: освіта, наука, виробництво. – 2017. – № 28-29. – с.72-76.
15. Kumar Rajnish, Arbind Kumar Choudhary, Anand Mohan Agrawal. Inheritance metrics for object-oriented design — International Journal of Computer Science & Information Technology (IJCSIT), Vol 2, No 6, 2010

Додаток А. Алгоритм Ульмана без процедури оптимізації

step1:

$M^1 \leftarrow M^0$

$d \leftarrow 1$

$H_1 \leftarrow 0$

$i \leftarrow 0$

for each $i <= p_\alpha$ **do**

$F_i \leftarrow 0$

end for

step2:

if $\nexists j$ для якого $m_{dj}=1$ та $F_j=0$

goto step7

end if

$M_d \leftarrow M^1$

if d = 1 **then**

$k \leftarrow H_1$

else

k ← 0

end if

step3:

$k \leftarrow k + 1$

if $m_{dk} = 0$ or $F_k = 1$ **then**

goto step3

end if

for all $j \neq k$

$m_{dj} \leftarrow 0$

end for

step4:

if d < p_α **then**

goto step6

else

if (умова (2.1)) **then**

output “Ізоморфізм знайдено”

end if

end if

step5:

if $\nexists j>k$ такі що $m_{dj}=1$ та $F_j=0$ **then**

goto step7

end if

$M^1 \leftarrow M_d$

goto step3

step6:

$M^1 \leftarrow k$

$F_k \leftarrow 1$

$d \leftarrow d+1$

goto step2

step7:

if d =1 **then**

exit

end if

$F_k \leftarrow 0$

$d \leftarrow d-1$

$M^1 \leftarrow M_d$

$k \leftarrow H_d$

Додаток Б. Порівняльна таблиця існуючих інструментів для розпізнавання використання шаблонів проектування

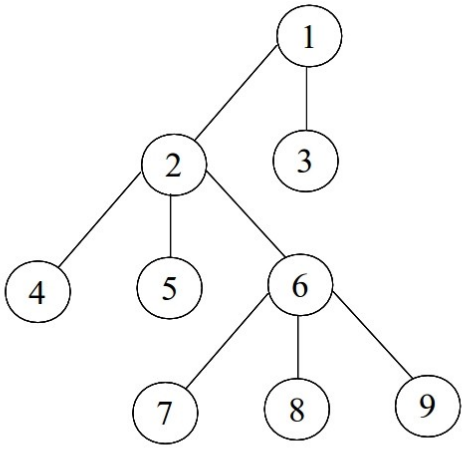
Порівняльна таблиця існуючих інструментів для розпізнавання використання шаблонів проектування

Назва інструменту	Переваги	Недоліки
PINOT	Результативність підтверджена тестами на проектах з відкритим вихідним кодом мовою Java: Java AWT 1.3, JHotDraw 6.0b1, Java Swing 1.4, java.io 1.4.2, java.net 1.4.2, javac 1.4.2, Apache Ant 1.6.2, ArgoUML 0.18.1. Наявність розширень для збільшення можливостей та покращення якості роботи.	Відсутність підтримки, останні зміни були внесені 27 листопада 2007 року. Жорстка залежність від компілятора через прив'язку до абстрактного синтаксичного дерева (АСД), таблиці символів (ТС), що генеруються.
MUSCAT	Візуалізація шаблонів проектування. Можливість модифікації аспектів шаблонів проектування.	Необхідність вивчення створеної мови для налаштування. Відсутність можливості додавання нових шаблонів проектування. Залежність від програмного забезпечення, підтримка якого вже закінчилась.
FUJABA	Використання потужного апарату Unified Modeling Language (UML) діаграм класів та зв'язків між класами. Широка міжнародна підтримка даного проекту.	Зміщення акценту з виявлення шаблонів проектування на підтримку моделе-орієнтованої методики розробки. Нижчі показники виявлення шаблонів проектування порівняно із аналогічною утилітою PINOT. Зміна вектору підтримки, перехід до реалізації будови діаграм історій.
WOP	Мово-незалежна формалізація шаблонів проектування. Можливість оцінювати та брати участь у створенні опису до шаблонів проектування усіма учасниками проекту та онлайн підтримка.	Нижчі показники виявлення шаблонів проектування порівняно з аналогічною утилітою PINOT. Припинення підтримки, останнє оновлення проекту датується 18 березням 2007 року.

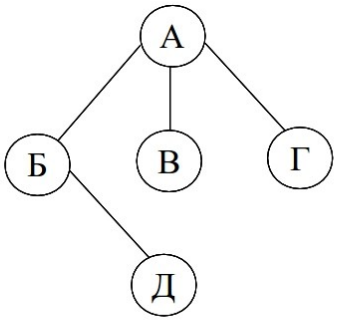
Додаток В. Схема використання структур даних у розробленому модифікованому способі

СИСТЕМА РОЗПІЗНАВАННЯ ВИКОРИСТАННЯ ШАБЛОНІВ ПРОЕКТУВАННЯ В ПРОГРАМАХ

Схема використання структур даних у розробленому модифікованому способі



Дерево пошуку



Дерево-шаблон

А	2
Б	6
В	5
Г	4
Д	7

А	2
Б	6
В	4
Г	5
Д	8

А	2
Б	6
В	5
Г	4
Д	9

Частина множини словників відповідності

А	1
:error	true

А	2
Б	4
В	5
Г	6
:error	true

А	2
Б	5
В	4
Г	6
:error	true

Частина множини словників часткової відповідності

НТУУ "КПІ ім. Ігоря Сікорського", ФПМ, група КВ-62м
Лиман Дмитро

Додаток Г. Псевдокод модифікованого способу

СИСТЕМА РОЗПІЗНАВАННЯ ВИКОРИСТАННЯ ШАБЛОНІВ ПРОЕКТУВАННЯ В ПРОГРАМАХ

Псевдокод модифікованого способу

```
function possible-correspondence-layer (design-pattern-layer model-layer-combination)
possible-corespondences := пустий контейнер словників;
begin
  for  кожен контейнер combination з model-layer-combination ;
    possible-correspondence := пустий словник;
    for кожна вершина model-vertex з контейнеру combination
        кожна вершина pattern-vertex з контейнеру design-pattern-layer;
       Possible-correspondence := fill-correspondence-map (pattern-
       vertex,  model-vertex, possible-correspondence) ;
   Додати словник possible-correspondence до possible-correspondences;
  return possible-correspondences;
end
```

Псевдокод побудови можливих відповідностей між ярусами двох дерев

```
function fill-correspondence-map (design-pattern-vertex model-combination-vertex correspondence-map)
begin
  if э накладання ключів або значень між відповідністю design-pattern-vertex та model-combination-vertex у correspondence-map then
    begin
      Додавання до correspondence-map ключу error зі значенням true;
    end
  else
    begin
      Додавання до correspondence-map ключу design-pattern-vertex та значення model-combination-vertex;
    end
  return correspondence-map;
end
```

Псевдокод заповнення словника відповідності

НТУУ “КПІ ім. Ігоря Сіікорського”, ФПМ, група КВ-62м
Лиман Дмитро

Додаток Д. Графік залежності ширини дерева пошуку подібних дерев від часу виконання для модифікованого способу перебору, способу Ульмана та способу Чунга

Додаток Е. Графік залежності глибини дерева шаблону подібного дерева від часу виконання для модифікованого способу перебору, способу Ульмана та способу Чунга

Додаток Є. Графік залежності ширини дерева шаблону подібного дерева від часу виконання за умови відсутності подібності для модифікованого способу перебору, способу Ульмана та способу Чунга