МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА СПЕЦІАЛІСТА

на тему: **«РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ОПТИМІЗАЦІЇ РОЗТАШУВАННЯ ЕЛЕМЕНТІВ UML ДІАГРАМ»**

Виконав: студент 1 курсу, групи 7.1226-з
спеціальності
122 Комп'ютерні науки та інформаційні технології
(шифр і назва спеціальності)
В.М. Печерський
(ініціали та прізвище)

Керівник доцент кафедри комп'ютерних наук, доцент, к.ф.-м.н. Єрмолаєв В.А.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри програмної інженерії, к.т.н. Чопоров С.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2017

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

РЕФЕРАТ

Кваліфікаційна робота спеціаліста «Розробка програмного забезпечення для оптимізації розташування елементів UML діаграм»: 54 с., 18 рис., 23 джерела, 2 додатки.

Об’єкт дослідження – алгоритми для розташування графів, та їх застосування на UML діаграмах.

Мета роботи – розробка алгоритму для розташування UML діаграм з подальшим впровадженням цього алгоритму у редакторі ArgoUML.

Метод дослідження – описовий, порівняльний, алгоритмізація, програмування.

Апаратура – персональний комп’ютер, програмне забезпечення.

Результат дипломного проекту – алгоритм у якості розширення ArgoUML, для автоматичного розташування елементів у діаграмах класів UML.

UML, XMI, ARGOUML, LAYOUT, ГРАФ, МІШАНИЙ ГРАФ, СПРЯМОВАНИЙ ГРАФ.

ABSTRACT

Specialist's qualifying paper «The Development of Software for Optimizing UML Diagram Layouts»: 54 pages, 18 figures, 23 references, 2 supplements.

The object of the study – algorithms for graph layouting and their application on UML diagrams.

The aim of the study – development of algorithm for UML diagram layouting with further implementation in ArgoUML editor.

The methods of research – descriptive, comparative, algorithmic, programming.

The equipment – personal computer, software.

Result of graduation project – algorithm as an extension for the ArgoUML which allows automatic layouting of elements in the UML class diagrams.

UML, XMI, ARGOUML, LAYOUT, GRAPH, MIXED GRAPH, DIRECTED GRAPH.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

UML	Unified Modeling Language, Уніфікована Мова Моделювання
G	Graph, граф
V	Vertex, вузол
E	Edge, ребро
ПЗ	Програмне Забезпечення
CPU	Central Processing Unit, Центральний процесор, ЦП
XMI	XML Metadata Interchange, XML Обмін метаданими
API	Application programming interface, Прикладний програмний інтерфейс

ВСТУП

У зв’язку з швидким розвитком Інтернет-технологій та при проектуванні комп’ютерних систем досить важливо швидко отримувати та відображати інформацію наочно. Найчастіше застосовуються такі форми візуалізації результатів як: графи для відображення зв’язків між елементами системи, алгоритми роботи для опису функціонування системи, діаграми для відображення вхідних та вихідних даних. Тому питання розміщення елементів завжди залишається актуальною темою.

Метою роботи є розроблення алгоритму автоматичного розміщення елементів для подальшого використання у UML діаграмах.

Предметом дослідження є моделі структурних зв’язків між елементами UML діаграм та методи їх перетворення і відображення.

Об’єктом — структурні зв’язки між елементами UML діаграми та результат роботи алгоритму візуалізації графу.

Методи дослідження. При розв’язанні задач використовувалися методи теорії графів, які забезпечили вибір внутрішнього представлення графових моделей та зв’язків між елементами.

Практичне значення одержаних результатів. Сформульовано критерії оцінювання функціональних можливостей основних модулів алгоритмів візуалізації графових моделей, за допомогою яких можна здійснити порівняльний аналіз та вибір оптимального рішення комп’ютерного відображення. А також побудований алгоритм відображення UML діаграм на основі існуючих алгоритмів з проведенням його подальшої оптимізації враховуючі поточні вимоги.

Основними матеріалами для виконання цього кваліфікаційного проекту є документація з аналізу і розробки графів, візуалізації та алгоритмів їх побудови. А також документація з описання архітектурних рішень

програмного забезпечення ArgoUML для коректної імплементації розробленого алгоритму.

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ РОЗМІЩЕННЯ ЕЛЕМЕНТІВ UML ДІАГРАМ

У цьому розділі будуть розглянуті основні відомості про діаграми UML, а також розкриті основні поняття графів та чому графи безпосередньо зв’язані з UML.

Також я хотів би оглянути такі поняття як графи та алгоритми їх реалізації. Представити детальний розгляд різних типів алгоритмів візуалізації графів. А саме порівняти їх основні характеристики та виявити проблеми зв'язані з їх розробкою.

1.1 Діаграми в UML

UML (Unified Modeling Language) — уніфікована мова моделювання, використовується у парадигмі об'єктно-орієнтованого програмування. Є невід'ємною частиною уніфікованого процесу розробки програмного забезпечення. А також є мовою широкого профілю, це відкритий стандарт, що використовує графічні позначення для створення абстрактної моделі системи, що називається UML-моделлю. Ця мова увібрала в себе найкращі якості і досвід методів програмної інженерії, котрі з успіхом використовувались впродовж останніх років при моделюванні великих та складних систем.

UML був створений для визначення, візуалізації, проектування і документування в основному програмних систем. Він не є мовою програмування, але в засобах виконання UML-моделей як інтерпретованого коду можлива кодогенерація [1]. UML є простим і потужним засобом моделювання, який може бути ефективно використаний для побудови концептуальних, логічних та графічних моделей складних систем самого

різного цільового призначення. UML має велику кількість типів діаграм і частота їх використання залежить від області застосування та конкретного розробника.

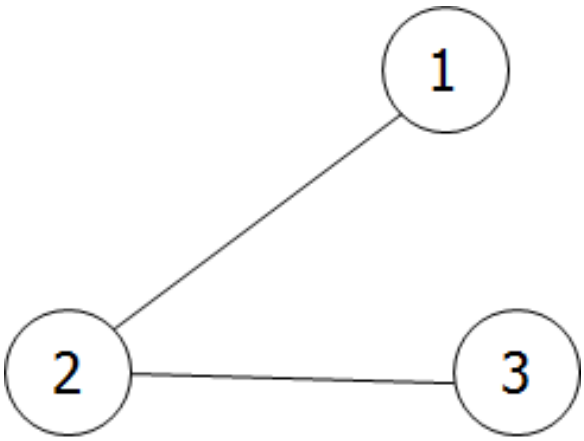
1.1.1 Графи як основа представлення UML діаграм

Діаграми UML можна уявити як мішаний граф вершин сутностей та зв'язків (ребер, дуг) [2]. Наприклад в якості ребер виступають такі зв’язки як асоціація, генералізація, наслідування, композиція та агрегація, а у якості вершин виступають такі сутності як клас, інтерфейс та інші.

Для реалізації кваліфікаційної роботи я буду використовувати графи як основну структуру даних. Вони дозволяють мені гнучко керувати розміщенням блоків UML діаграм та їх залежностями. В якості блоків ми будемо використовувати вершини, в якості залежностей ребра. Таким чином граф дозволить нам відобразити UML діаграму в необхідній формі. Далі ми розглянемо графи більш детально.

1.1.2 Огляд структури графів

Граф являє собою сукупність об'єктів і зв'язків між ними. *G* — це впорядкована пара G := (V, E), для якої виконуються наступні умови: V — множина вершин або вузлів, E — множина пар вершин з V, які називають ребрами. V (і так само E) зазвичай вважаються скінченими множинами [3].



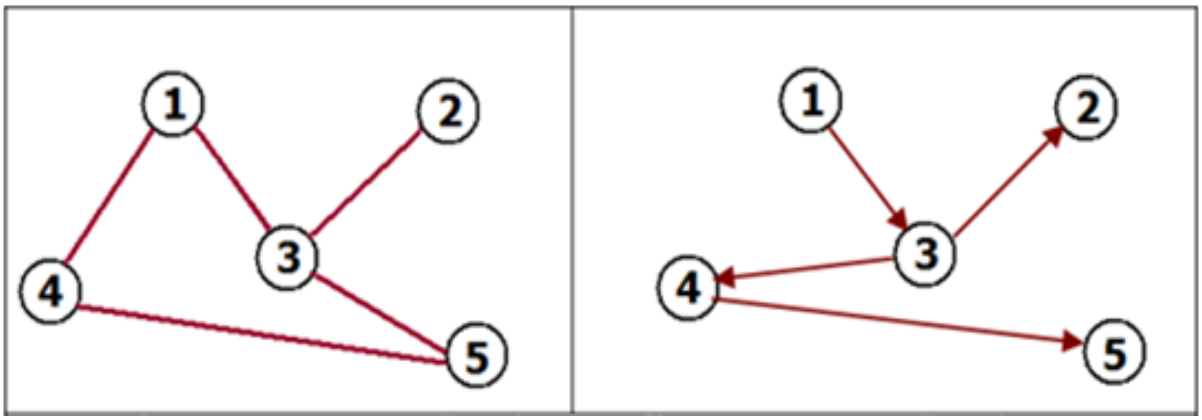
1,2,3 - V (вершина) , 1→2 і 2→3 - E (ребра)

Рисунок 1.1 – Структура графа

Об'єкти розглядаються як вершини, або вузли графу, а зв'язки – як дуги, або ребра це можна побачити на прикладі (рис. 1.1). Для різних областей використовуються різні види графів, які можуть відрізнятися орієнтованістю, обмеженнями на кількість зв'язків і додатковими даними про вершини або ребра.

1.1.3 Орієнтований і неорієнтований та змішаний графи

Граф може бути орієнтований або неорієнтований. В орієнтованому графі, зв'язки є спрямованими. Тобто пара (1, 3) і (3, 1) – це два різних зв’язка. Наприклад на (рис. 1.2б) існує тільки зв’язок (1, 3). У свою чергу в неорієнтованому графі, зв'язки не спрямовані, це означає: якщо існує зв'язок (1, 3), то є і зворотній зв'язок (3, 1) (рис 1. 2а).



а) б)

Рисунок 1.2а – Неорієнтований граф Рисунок 1.2б – Орієнтований граф

Наприкладі, неорієнтований граф: Сусідство (в житті). Якщо (1) сусід (3), то (3) сусід (1) (рис. 1.2а).

Орієнтований граф: Посилання. Сайт (1) може посилатися на сайт (3), але зовсім не обов’язково (хоча можливо) що сайт (3) може посилатися на сайт (1) (рис. 1. 2б) [4].

Змішаний граф G — це граф, в якому деякі ребра можуть бути орієнтованими, а деякі — неорієнтованими. Записується впорядкованою трійкою G := (V, E, A), де V — множина вершин або вузлів, E — множина пар вершин з V, які називають ребрами і A— множина (впорядкованих) пар різних вершин, які називаються дугами або орієнтованими ребрами [5]. Орієнтований і неорієнтований графи є окремими випадками змішаного — граф, що містить як орієнтовані, так і неорієнтовані ребра.

1.1.4 Будова UML діаграми. Класи і відношення

Діаграма компонентів UML це графічне представлення деякої частини графа моделі. Вона використовується для перегляду і редагування згенерованої структури класів та їх залежностей. Діаграма компонентів UML може застосовувати різні моделі, бібліотеки, файли, пакети, виконувані файли і ще безліч інших елементів.

Діаграма класів включає в себе різні класи, які безпосередньо використовуються в програмному коді. Вона являє собою статичну структурну діаграму, призначену для опису структури системи, а також демонстрації атрибутів, методів і залежностей між декількома різними класами. Класи відображаються у вигляді прямокутників, а відношення між ними малюються у вигляді пунктирних ліній, прямих ліній, ліній з ромбом та стрілочок.

Варто відзначити той факт, що є декілька точок зору на побудову таких діаграм в залежності від того яким чином вони будуть використовуватись.

Концептуальна у даному випадку діаграма класів UML здійснює опис моделі певної предметної області, і в ній передбачаються тільки класи прикладних об'єктів.

Специфічна діаграма використовується в процесі проектування різних інформаційних систем.

Реалізаційна діаграма класів включає в себе різноманітні класи, які безпосередньо використовуються в програмному коді [6].

Розробка цього кваліфікаційного проекту полягає в наданні функціоналу автоматично розміщувати UML елементи діаграми. Розмір блоків класів буде визначатися від довжини імені, кількості атрибутів і залежностей у кожного елемента. Розташовування елементів буде формуватися враховуючі поточні зв’язки між елементами.

1.2 Огляд існуючих алгоритмів для впорядкування графів

1.2.1 Алгоритми

Алгоритм — це точне розпорядження яке однозначно визначає обчислювальний процес, що веде від варіаційних початкових даних до бажаного результату. Для комп'ютерних програм алгоритм є списком деталізованих інструкцій, що реалізує процес обчислення, який, починаючи з початкового стану, відбувається через послідовність логічних станів, і завершується кінцевим станом [7].

Існує безліч алгоритмів візуалізації графів, розглянемо алгоритми, що є найбільш поширеними на сьогодні. Враховуючі той факт, що для розробки моєї кваліфікаційної роботи потрібно використовувати алгоритми розташування, серед великої їх кількості ми звернемо увагу на такі алгоритми як: Circle Layout, Compact Tree Layout, Radial Layout, Organic Layout, Partition Layout, Orthogonal Layout, Sugiyama [8].

1.2.2 Проблеми в розробці існуючих алгоритмів

Кожен алгоритм передбачає існування початкових (вхідних) даних та в результаті роботи призводить до отримання певного результату. Робота кожного алгоритму відбувається шляхом виконання послідовності деяких елементарних дій. Ці дії називають кроками, а процес їхнього виконання називають алгоритмічним процесом.

Перед тим як виділити основні проблемі розробки алгоритму треба зрозуміти якими вхідними даними він буде оперувати.

Вхідною інформацією при вирішенні завдань розміщення є:

а) дані про конфігурацію UML діаграми;

б) кількість і геометричні розміри UML елементів, що підлягають розміщенню;

в) схема з’єднань;

г) перелік обмежень на взаємне розташування окремих елементів, що враховують особливості поточної діаграми.

Завдання алгоритму розміщення зводиться до пошуку позицій для кожного елемента таким чином що елементи діаграми будуть розташовані з мінімальною кількістю перетинів.

Отже однією з найважливіших проблем є затрати машинного часу при виконанні алгоритму.

Другою проблемою можна назвати якість виконання алгоритму розміщення і відсоток вірогідності отримання похибки. Тобто, якщо ви хочете отримати найвищий результат якості роботи алгоритму, вам потрібно пожертвувати часом його виконання. А як наслідок - втратити багато часу на оптимізацію внутрішніх процесів алгоритму.

Третьою, не менш важливою частиною, є врахування виділеної пам’яті на обчислювання даного алгоритму. Якщо ви плануєте виконувати роботу розташування на комп’ютерах зі слабкими характеристиками компонентів таких як процесор, пам’ять, тощо, вам треба детально продумати кожен крок його реалізації і можливі шляхи оптимізації.

Тому, на даний момент, всі застосовані алгоритми розміщення використовують проміжні критерії, які можуть сприяти вирішенню основного завдання. До таких критеріїв належать:

а) мінімум сумарної зваженої довжини з'єднань;

б) універсальна довжина ребер: мінімізація відмінностей в довжинах ребер;

в) мінімальне число сполук, довжина яких більше заданої;

г) мінімальна кількість перетину зв’язків;

д) мінімальна довжина дуг;

е) мінімальний розмір елементів;

є) обмеження по широті і висоті площі на якій розміщуються елементи;

ж) симетрія.

Взагалі не існує, ідеальних алгоритмів для розміщення елементів у графах. Однак, маючи приблизні характеристики вхідних даних і визначений перелік необхідних критеріїв, можна підібрати метод, який буде працювати оптимальним чином.

1.3 Візуалізація графів

Візуалізація графів стосується геометричного представлення графів та мереж і використовується у тому ПЗ, де вирішальне значення має візуалізація інформації у вигляді графів.

Задача візуалізації графових структур є надзвичайно актуальною, важливим аспектом якої є подолання розриву між теоретичними досягненнями і реалізованими рішеннями. Адже із розповсюдженням Інтренет-технологій, розвитком нових напрямків у науці та впровадженням новітніх підходів до організації освітнього і управлінського процесів, все частіше і частіше виникає потреба у графічному представленні графових структур даних різного походження. Особливо актуальною ця задача стає при потребі візуалізації структур значних розмірів — зокрема при вирішенні різнорідних проблем наукового, корпоративного та навіть державного характеру. Актуальність даної теми додатково підсилюється тим, що вона не є вузько спеціалізованою, тобто не орієнтована для застосування лише у певній галузі або сфері, а навпаки є легко застосовною в багатьох галузях людської діяльності. Програмне забезпечення для візуалізації графових структур створюється з метою вирішення цих проблем [9].

1.3.1 Способи візуалізації графів і доступні засоби для впорядкування

Візуалізація або відображення графів, як відгалуження теорії графів, що відноситься до топології і геометрії - двовимірне подання графа. В основному, це графічне представлення укладання графа на площину (як правило, допускається перетин ребер), спрямоване, зазвичай, на зручне відображення деяких властивостей графа, або об'єкта, що моделюється.

Можна виділити наступні способи відображення:

а) довільне;

б) прямолінійне – ребра представлені відрізками;

в) сіткове – припускає, що всі вершини, а також всі точки пересічення і згини ребер мають цілочисельні координати, а саме перебувають у вузлах координатної сітки, утвореної прямими, паралельними координатним вісями і пересікаючими їх в точках з цілочисельними координатами;

г) полігональне або полілінійное – для відображення ребер використовуються ламані;

д) ортогональне – ребра видаються ламаними, відрізки яких – вертикальні або горизонтальні лінії;

е) планарне або плоске – зображення передбачає відсутність точок перетину у ліній, що зображують ребра;

ж) висхідний або спадна – висхідний (відповідно спадний) зображення має сенс по відношенню до ациклічного орграфу і передбачає, що кожне ребро орграфа відображається кривою, яка монотонно не убуває (відповідно не зростає) у вертикальному напрямку [10].

Алгоритми візуалізації графів можна умовно розділити на такі групи:

а) алгоритми компоновки;

б) алгоритми розміщення;

в) алгоритми трасування.

Алгоритми компонування:

а) алгоритми, що використають методи цілочисельного програмування (використовуються нечасто);

б) послідовні алгоритми – алгоритми при роботі яких спочатку вибирається вершина графа, а потім здійснюється послідовний вибір вершин (із числа нерозподілених) і приєднання їх до формованої частини графа;

в) ітераційні алгоритми. В ітераційних алгоритмах початкове розрізання графа на шматки виконують довільно; оптимізація компонування досягається парними або груповими перестановками вершин графа з різних шматків;

г) змішані алгоритми. У змішаних алгоритмах компонування для одержання початкового варіанта “розрізу” використовується алгоритм послідовного формування частин.

Алгоритми розміщення діляться на такі групи:

а) силові алгоритми розміщення – процес розміщення елементів представляється як рух до стану рівноваги системи елементів, на кожну з яких діють сили тяжіння і відштовхування, інтерпретують зв'язки між розміщеними елементами;

б) послідовні алгоритми засновані на припущенні, що для одержання оптимального розміщення необхідно в сусідніх позиціях розташовувати елементи, максимально зв'язані один з одним;

в) ітераційні алгоритми розміщення мають структуру, аналогічну ітераційним алгоритмам компонування.

Трасування з'єднань є, як правило, заключним етапом відображення і полягає у визначенні ліній, що з’єднують елементи [11].

1.3.2 Критерії вибору відповідного алгоритму

При розробці алгоритмів дуже важливо мати можливість оцінити ресурси, необхідні для проведення обчислень, результатом оцінки буде функція складності. Оцінюваним ресурсом найчастіше є процесорний час і пам'ять. Оцінка дозволяє передбачити час виконання і порівнювати ефективність алгоритмів.

Також одним з важливих факторів є порівняння складності роботи алгоритмів або нотація «великого О».

Складність алгоритму – це те, що грунтується на порівнянні двох алгоритмів на ідеальному рівні, ігноруючи низькорівневі деталі на зразок реалізації мови програмування, «заліза», на якому запущена програма, або набору команд в CPU.

Аналіз складності також дозволяє нам пояснити, як буде вести себе алгоритм при зростанні вхідного потоку даних. Якщо алгоритм виконується одну секунду при 1000 елементах на вході, то як він себе поведе, якщо ми подвоїмо це значення? Працюватиме також швидко, в півтора рази швидше або в чотири рази повільніше? Саме це ми і зможемо визначити знаючи його показник складності.

Нижче на рисунку 1.3 наведений перелік класів функцій, які часто зустрічаються в аналізі алгоритмів. Функції, які зростають повільніше, наведені першими c – константа, $n \to \infty$ [12].

нотація	назва
$O(1)$	константа
$O(\log n)$	логарифмічне
$O([\log n]^c)$	полілогарифмічне
$O(n)$	лінійне
$O(n \cdot \log n)$	лінеарітмічне
$O(n^2)$	квадратичне
$O(n^c)$	поліноміальне
$O(c^n)$	експоненціальне
$O(n!)$	факторіальне

Рисунок 1.3 – Перелік функцій нотації «великого О»

Зазвичай кожен розроблений алгоритм вже має свою оцінку складності, тому вибираючи алгоритм можна наперед передбачити поведінку його роботи. Але завжди треба пам’ятати – жоден алгоритм не є універсальним і вирішує тільки відповідні цілі поставленої задачі.

1.3.3 Таблиця порівнянь існуючих алгоритмів розміщення

Нижче у таблиці 1.1 я хотів би порівняти декілька відомих алгоритмів розміщення, визначити його загальний вигляд та сферу застосування.

Таблиця 1.1 — Порівняння характеристик алгоритмів розміщення

Назва	Відображення	Сфера застосування
Circle Layout	Створює вузли та розділи, аналізуючи структуру зв'язності мережі, а також організовує розділи як окремі кола.	Соціальні мережі, управління мережею WWW, візуалізація, електронна комерція
Compact Tree Layout	Створює компактні ортогональні креслення дерева	Це особливо корисно, коли граф повинен чітко відповідати заданому розміру.
Radial Layout	Вузли графа розташовані на концентричних колах	Для візуалізації направлених графів і дерев структур.
Organic Layout	Отримані результати часто виявляють притаманну симетричну і кластерну структуру графа, вони показують добре збалансований розподіл вузлів і мають кілька ребер перетину.	Підходить для візуалізації магістральних областей з прикріпленими периферійними кільцями або структурами у вигляді зірок
Partition Layout	Підтримує організацію заданих частин діаграми користувачем, так званих часткових елементів	Підходить для додаткових сценаріїв, в яких нові додані частини повинні бути розташовані таким чином, щоб вони відповідали найкращим в даній схемі без застосування будь-яких змін в існуючому розміщенні.
Orthogonal Layout	заснований на topology-shape-metrics підході	представлення складних мереж
Sugiyama	розміщує вузли на горизонтальних рівнях, так що всі зв’язки спрямовані однаково	малювання класів, що беруть участь в спадкових відносинах.

1.4 Висновки

У данному розділі були отриманні результати з основних теоретичних понять UML діаграм та їх характеристик. Результати стосовно алгоритмів і особливостей їх розробки та основних критеріїв вибору підходящого алгоритму. Розглянуто тему графів та існуючі методи їх візуалізації, отримано порівняння найпоширеніших з них враховуючи особливості їх відображення та сфери застосування.

Дані результати будуть використанні у наступних розділах у вигляді теоретичної бази для створення алгоритму авто розміщення.

2 РЕАЛІЗАЦІЯ АВТОМАТИЧНОГО РОЗМІЩЕННЯ UML ДІАГРАМ

2.1 Особливості реалізації з використанням редактора ArgoUML

2.1.1 Дослідження існуючого в ArgoUML алгоритму та методів взаємодії з полотном

У процесі дослідження вбудованого алгоритму розміщення було визначено, що в його основі лежить комбінація поширених алгоритмів для відображення інформації - Orthogonal Layout та Sugiyama. Orthogonal Layout дозволяє оперувати великими об’ємами інформації, а Sugiyama - розташувати їх використовуючи ієрархічні взаємовідношення між елементами.

На етапі детального дослідження коду алгоритму було знайдене наступне:

а) клас ClassdiagramLayouter відповідає за процес розміщення UML елементів на діаграмі, який імплементує методи класу Layouter [13];

б) за побудову меню відповідає клас ActionLayout який є нащадком UndoableAction.

Основним методом алгоритму називається layout, який виконує наступний порядок дій:

а) спочатку виконується установка з’язків між елементами;

б) потім алгоритм порівнює rank і weight кожного вузла та переміщує його у потрібний рядок. Rank і weight відповідають за те наскільки даний вузол взаємодіє з іншими елементами. Наприклад вузол з великою кількістю зв’язків буде розташований ближче до центру;

в) далі проводиться розміщення вузлів враховуючі висоту рядка, розмір самого вузла та вертикальний і горизонтальний відступи;

г) останнім етапом є розміщення ребер де перевіряються залежності наслідування та виконується розміщення враховуючи особливості батьківських вузлів.

2.1.2 Пошук недоліків і методів їх усунення

Не зважаючи на те що алгоритм має досить гарну реалізацію та добре-структуровану логіку роботи, на практиці виявилось що не у всіх випадках він коректно виконує розміщення елементів та їх залежностей. А саме:

а) недосконала система розташування класів з однаковою weight та rank;

б) у деяких випадках елементи діаграми досить близько розташовувались один до одного що значно ускладнює розуміння діаграми;

в) некоректне відображення зв’язків між елементами таким чином що два зв’язки могли сходитись в одну лінію;

г) для складних UML діаграм виконання алгоритму могло бути проігноровано.

2.1.3 Пошук відповідного алгоритму з уже існуючих для розміщення діаграми класів

Треба одразу зробити наголос на тому що я використовую результати кваліфікаційної роботи [14] результатом якої є діаграма класів, тому у кваліфікаційній роботі алгоритм авто-розміщення використовується саме для діаграм класів. В майбутньому я планую виконати його розширення для інших видів діаграм (детально описано у розділі 3).

У зв’язку з тим що Orthogonal Layout та Sugiyama ідеально підходять для реалізації поставленої задачі – рішення використовувати їх як основу для створюємого алгоритму у поточного дипломного проекту є раціональним.

2.2. Опис обраних алгоритмів розміщення

2.2.1 Алгоритм Orthogonal Layout і Sugiyama

Один з алгоритмів, який буде застосовуватись у моїй кваліфікаційній роботі, являє собою поєднання добре відомого алгоритму Sugiyama з деякими ортогональними методами малювання.

Orthogonal Layout – є основним для виконання розміщення неорієнтованих графів. Він допомагає побудувати чіткі уявлення складних мереж і особливо підходить для застосування у таких областях як:

а) розробка програмного забезпечення;

б) схема бази даних;

в) управління системою;

г) уявлення знань.

Алгоритм ортогонального розміщення заснований на підході topology–shape–metrics і складається з трьох етапів. На першому етапі обчислюється перехрещення. Друга фаза обчислює перегини на кресленні, а на третьому етапі визначаються остаточні координати.

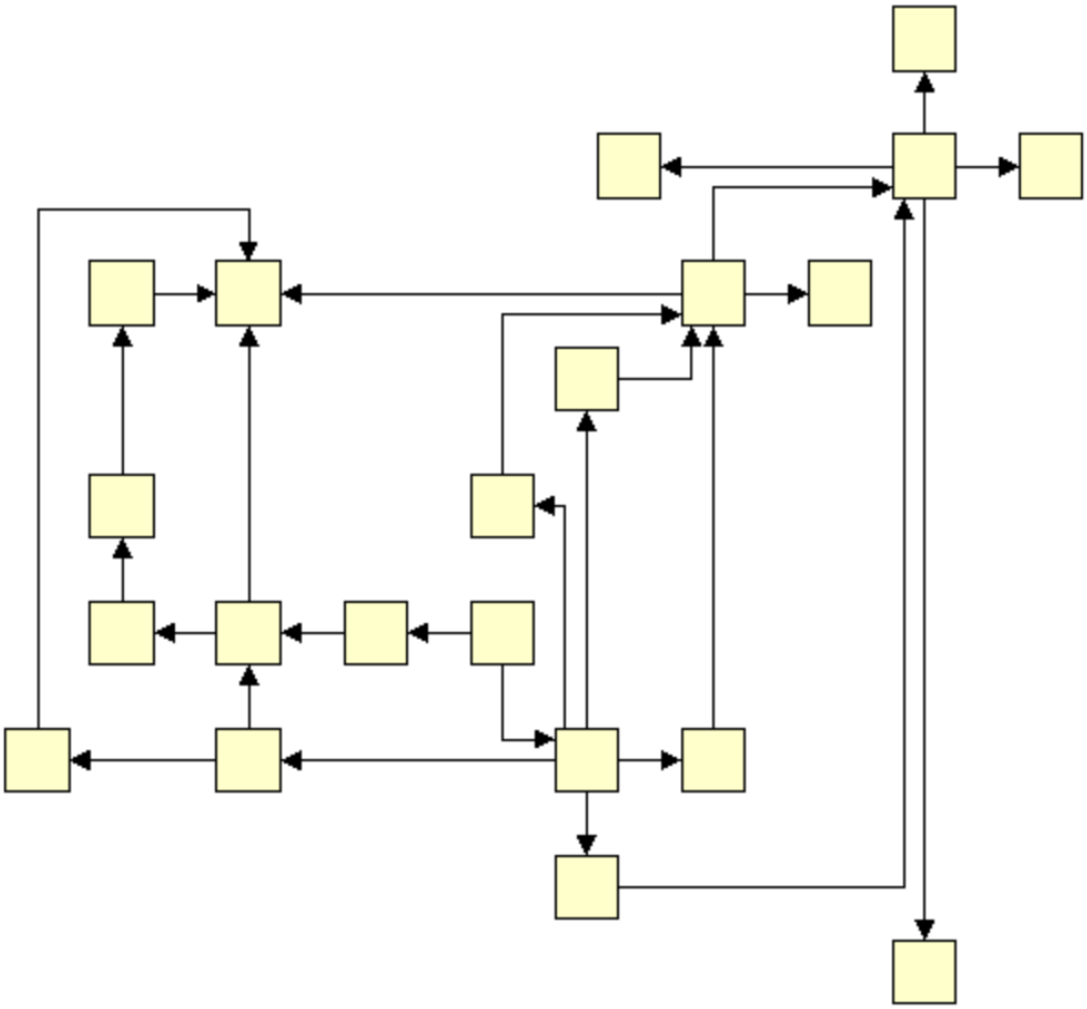


Рисунок 2.1 – Приклад роботи алгоритму Orthogonal Layout

Алгоритм Sugiyama використовується для малювання класів, що беруть участь в спадкових відносинах. На рисунку 2.1 представлений приклад роботи алгоритму Orthogonal Layout.

Алгоритм має чотири кроки:

а) тимчасово видаляє будь-які спрямовані цикли, перевернувши невелику кількість зв’язків;

б) розміщує вузли на горизонтальних рівнях, так що всі зв’язки спрямовані однаково;

в) переставляє вузли на кожному рівні, мінімізуючи кількість перехрещень зв’язків і врівноважує розміщення;

г) додає зв’язки асоціації зберігаючи базову структуру.

На рисунку 2.2 відображена схема алгоритму Sugiyama і приклад роботи алгоритму Sugiyama на рисунку 2.3.

Рисунок 2.2 - Схема алгоритму Sugiyama

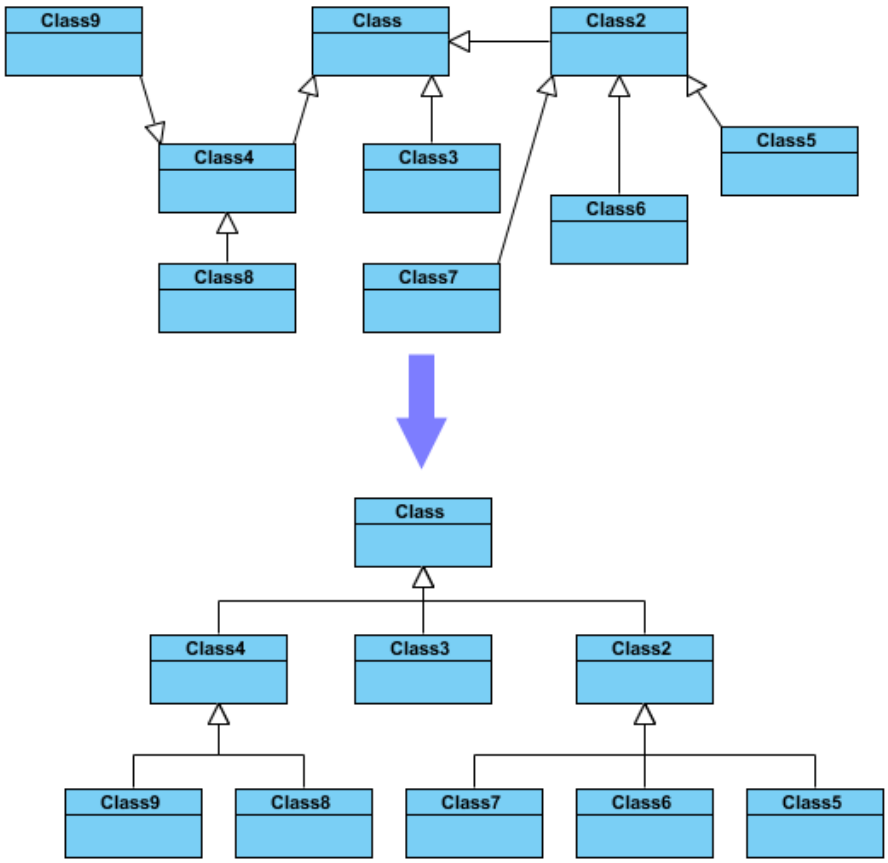


Рисунок 2.3 – Приклад роботи алгоритму Sugiyama

2.2.2 Переваги і недоліки обраних алгоритмів

Таблиця 2.1 – Порівняння алгоритмів

Переваги	Недоліки
Можливість розташовувати великі об’єми інформації.	Orthogonal Layout потребує більшого часу виконання для великої кількості ребер та вузлів, але є можливість провести оптимізацію шляхом відключення деяких параметрів.
Можливість розташовувати спадкові відносини зберігаючи ієрархію	Може потребувати додаткові ресурси пам’яті
Досить добре підходить для оптимізації візуальних критеріїв, таких як зведення до мінімуму перетинів між ребрами і їх довжин	
Можливість зберегти кількість фіктивних вершин і ребер лінії без збільшення числа перетинів	

2.3 Реалізація

2.3.1 Підготовка коду ArgoUML для впровадження класів реалізації алгоритму

Діаграма класів UML має додаткові вимоги, які не обробляються за допомогою загальних алгоритмів графіків компонування, наприклад, вузли не мають фіксований розмір і посилання, асоціації позначені не дуже наочно.

Для впровадження нового алгоритму авто-розміщення, дотримуючись поточної архітектури ArgoUML будуть створені два класи з назвами ClassdiagramSmartLayouter - повинен імплементувати інтерфейс Layouter та ActionSmartLayout - повинен наслідувати клас UnableAction [15].

ActionSmartLayout буде відповідати за процес обробки натиску кнопки панелі меню, отримувати поточну діаграму та передавати у клас ClassdiagramSmartLayouter для подальшої обробки інформації.

У свою чергу ClassdiagramSmartLayouter, отримавши поточну діаграму, повинен розкласти її на слої та елементи для проведення наступного калькулювання та розміщення елементів UML моделі.

2.3.2 Програмна реалізація алгоритму в ArgoUML

Розглядаючи програмну реалізацію на більш детальному рівні, спершу треба приділити увагу отриманню даних з діаграми, що виконується безпосередньо у конструкторі класу ClassdiagramSmartLayouter.

```
public ClassdiagramSmartLayouter (ArgoDiagram diagramToLayout) {
    diagram = diagramToLayout;
    for (Fig fig : diagram.getLayer ().getContents()) {
```

```
            if (fig.getEnclosingFig() == null) {
        add(ClassdiagramModelElementFactory.SINGLETON.getInstance(fig));
            }
        }
}
```

Наступним фрагментом коду є робота з rank і weight. Як можна побачити у додатку А, ранжування вузлів залежить від їх позиції в ієрархії елементів, а присвоєння ваги проводиться з ціллю досягнення непослідовного авто-розміщення.

Треба зазначити, що робота цієї функції ще не є досконалою враховуючи складності визначення weight (ваги) у елементів в різних групах залежностей.

Наступним розглянемо метод setupLinks() додаток Б, який встановлює висхідні та низхідні зв'язки між вузлами поточної діаграми. В основі реалізації цієї функції лежить процес аналізу поточних типів зв'язків та вузлів до яких вони приєднуються. Також хочеться сказати що у подальшій розробці цього проекту потрібно буде провести удосконалення функціоналу розміщення різних типів зв'язків, враховуючи всі виключні випадки.

І наприкінці треба коротко зазначити те, що метод layoutNodes встановлює координати кожному вузлу враховуючи його належність до складових груп. А метод layoutEdges виконує розміщення ребер використовуючи edge-type алгоритм, тобто базується на порівнянні їх типів.

2.4 Висновки

У данному розділі були отриманні наступні результати: досліджено проект ArgoUML та можливі шляхи його розширення, виявлені особливості реалізації розробки алгоритму авто-розміщення та його впровадження у проект ArgoUML, описана реалізації цього алгоритму на рівні коду.

Даний результат буде використаний у наступному розділі в ході проведення експериментів над розміщенням UML діаграм.

3 ЕКСПЕРЕМЕНТАЛЬНА ПЕРЕВІРКА РЕЗУЛЬТАТІВ РОЗМІЩЕННЯ

3.1 Кроки проведення експерименту

3.1.1 Отримати неупорядковану діаграму класів як результат парсингу та подальшого редагування UML моделі в ArgoUML

Розглянемо неупорядковану діаграму класів зображену на рисунку 3.1, як результат завантаження створеної UML діаграми для куплету з віршу [17]. Дана діаграма є результатом роботи парсера текстів природною мовою з кваліфікаційної роботи [14].

Рисунок 3.1 – UML діаграма представлена без використання алгоритму розташовування, куплету з віршу [17]

Поточний вигляд діаграми можна отримати після імпортування XMI файлу у програму ArgoUML, так як UML діаграми у форматі XMI не містять інформацію про розташовування елементів на діаграмі, такий неупорядкований вид є закономірним результатом.

3.1.2 Завантаження даної діаграми на полотно ArgoUML

Для проведення експерименту, нам необхідно мати можливість завантажувати діаграми з XMI файлів, які були отримані за допомогою програми парсера з кваліфікаційної роботи [14].

На рисунку 3.2 ви можете побачити, як саме це відбувається у ArgoUML.

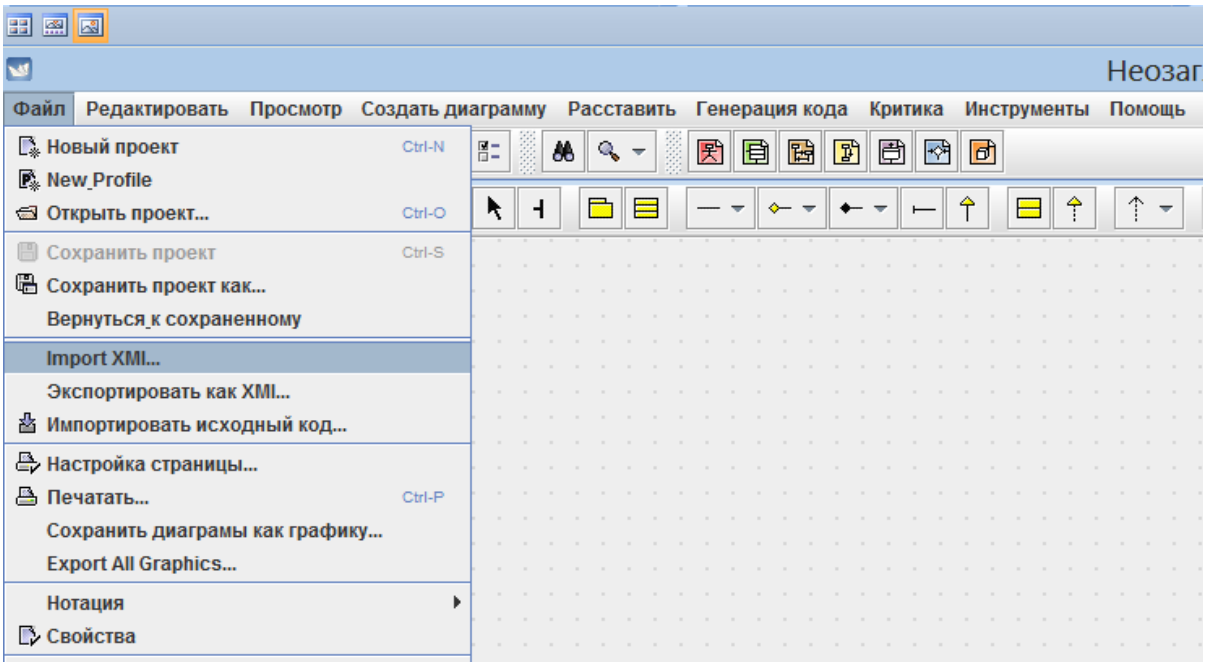


Рисунок 3.2 – Імпорт XMI файлу у програмі ArgoUML

Треба зазначити що не всі XMI файли можуть бути імпортовані, так як ArgoUML не підтримує деякі властивості з версії XMI 2.

3.1.3 Використання алгоритму розміщення і отримання візуального представлення

Далі буде розглянуто використання алгоритму розміщення для чотирьох неупорядкованих UML діаграм, та представлені результати роботи.

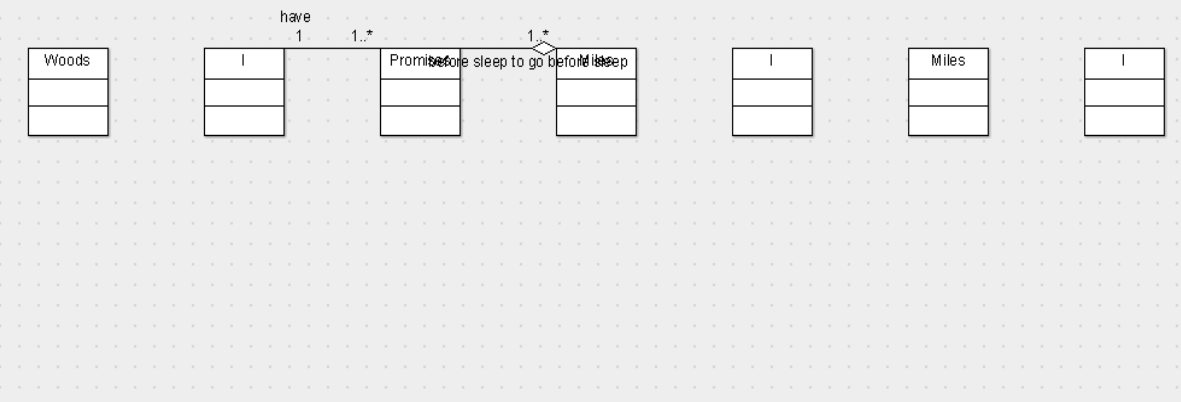


Рисунок 3.3 – UML діаграма після застосування алгоритму розташовування, для куплету з віршу [17]

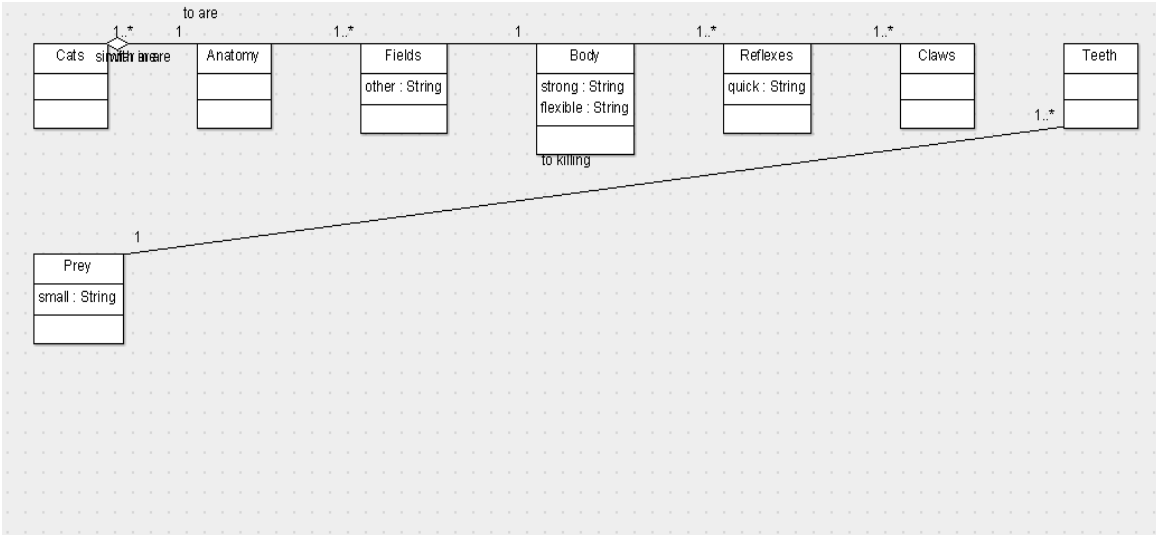


Рисунок 3.4 – UML діаграма опису котів як хижих тварин

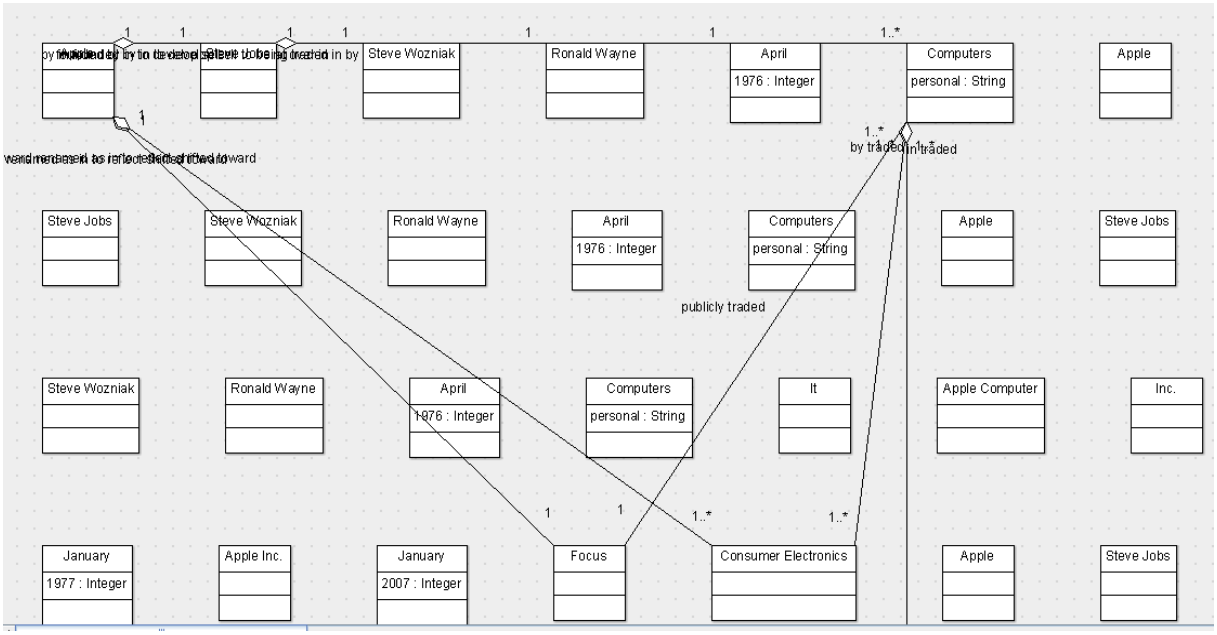


Рисунок 3.5 – UML діаграма створена з тексту про компанію Apple

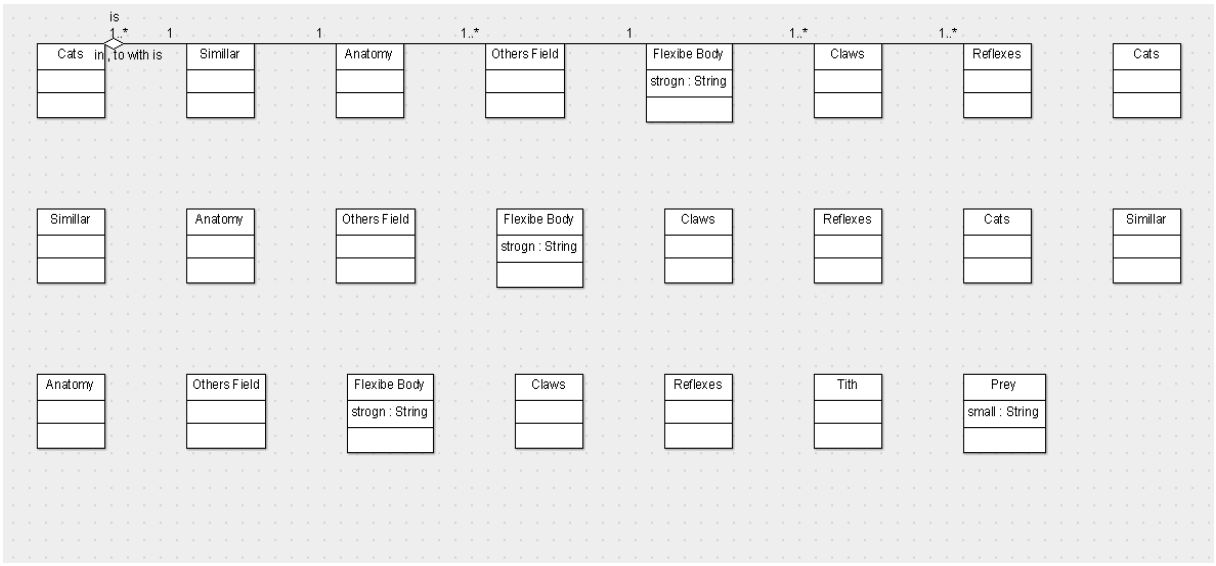


Рисунок 3.6 – UML діаграма опису котів як хижих тварин отримана з тексту який містить орфографічні та синтаксичні помилки

Як можна бачити на рисунках 3.3-3.6, не всі елементи діаграм розташовані у зручному вигляді, особливо імена для зв’язків між класами, ці сутності поки що не враховуються при використанні алгоритму

розташовування. У разі подальшого розвитку даного проекту, розташовування додаткових елементів буде важливим покращенням.

Також потрібно зазначити що якість використання алгоритму розташовування падає, при наявності великої кількості зв’язків та елементів у UML діаграмі, це можна побачити на рисунку 3.5 коли зв’язки між класами починають перетинатися.

3.2 Перевірка на практиці відображення і зручності розміщення UML діаграми

3.2.1 Впевнитися в правильності розміщення вершин графів

У якості вершин графу які впорядковано алгоритмом розташовування, можемо розглядати класи діаграми на рисунку 3.3. Алгоритм розташовування впорався з коректним розміщенням вершин графів (класів). Якщо розглянути решту рисунків 3.4-3.6, то можна побачити що і на цих діаграмах UML елементи розташовані у прийнятній послідовності, але на рисунку 3.5 можна бачити що з’єднані вершини (класи) знаходяться на великій відстані одна від одної. Це спричиняє схрещення деяких зв’язків, що вказує на не ідеальну роботу алгоритму.

3.2.2 Впевнитися в правильності розміщення ребер графів

Як було зазначено у розділі 3.2.1 некоректне розташовування вершин графу, призводить до некоректного відображення ребер цього графу, а саме до їх перехрещення. Це можна бачити на рисунку 3.7.

Що стосується рисунка 3.3, рисунка 3.4 та рисунка 3.6 на них можна побачити коректне розташовування ребер (зв’язків) між вершинами графу,

що свідчить про ще одну особливість поточного алгоритму яка вже була описана у пункті 3.1.3, її можна розглядати як пряму залежність якості роботи алгоритму від кількості вершин (класів) та ребер (зв'язків) у UML діаграмі.

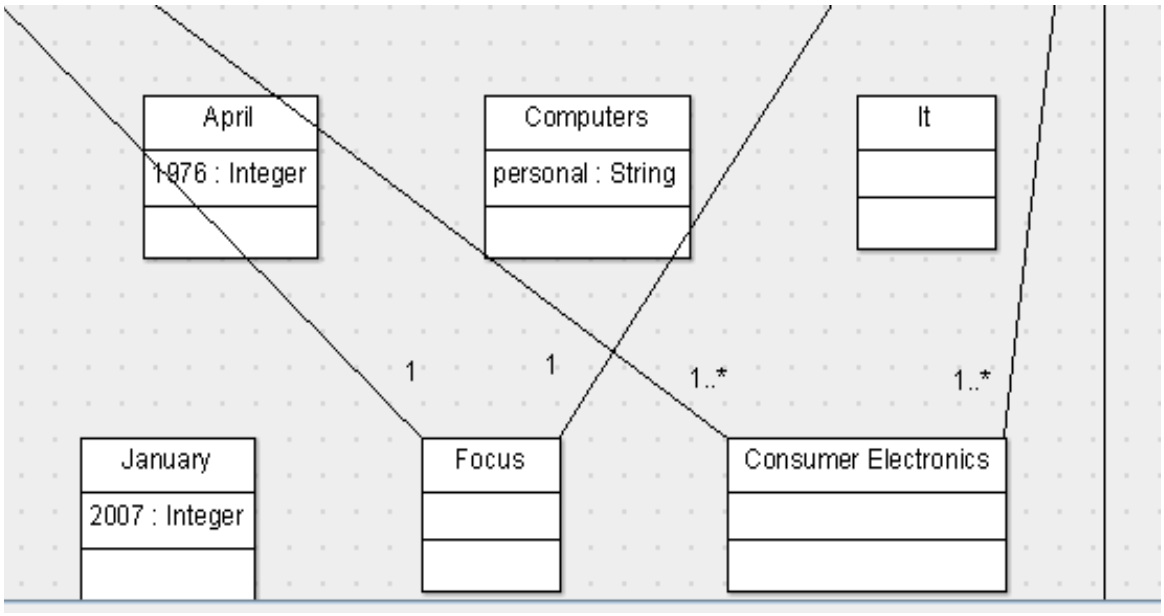


Рисунок 3.7 – Частина UML діаграми на якій можна побачити пересічення зв'язків між класами (вершинами)

3.2.3 Впевнитися в правильності загального розміщення UML діаграми класів

Розглянемо отримані діаграми рисунки 3.3-3.4, після використання алгоритму розташовування. Ці діаграми розміщені правильно, за виключенням імен зв'язків, які поки що не враховуються. Але якщо розглядати ці діаграми з точки зору користувача і спробувати уявити як можна ідеально розташувати UML елементи, то в результаті могла б вийти наступна діаграма як на рисунку 3.8.

Нажаль такого результату неможливо досягти, використовуючи поточний алгоритм розташовування елементів. Але цей алгоритм може бути

використаний для надання користувачу, деякої бази, з якої можна розпочати роботу з діаграмою. Також цей вид розташовування буде корисний для надання загального уявлення про діаграму, яку буде використано у якості стартової точки для початку роботи.

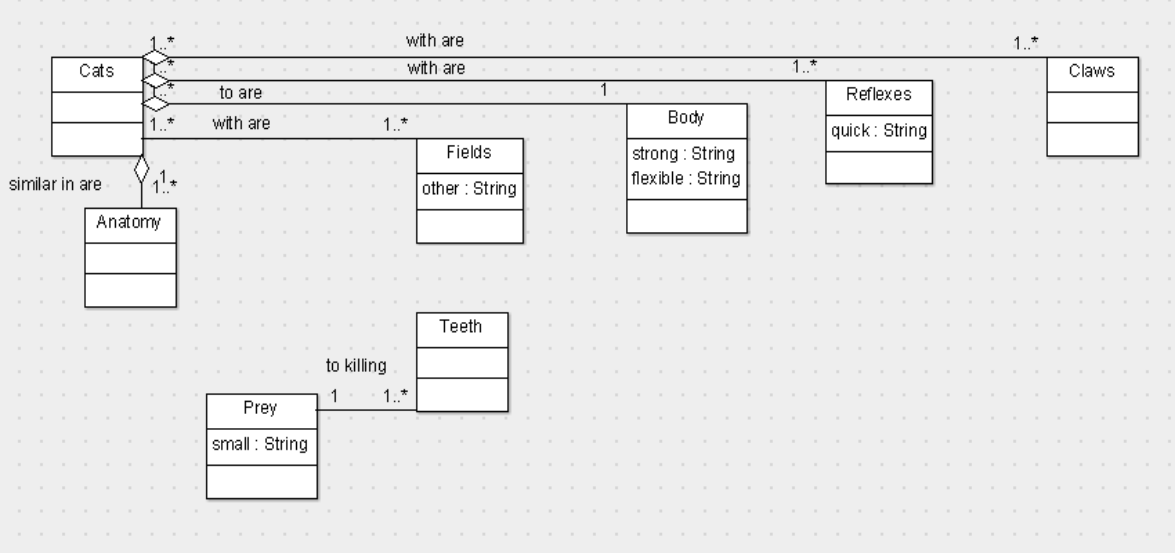


Рисунок 3.8 – UML діаграма, елементи якої були розташовані користувачем

3.2.4 Практично перевірити відображення і зручність розміщення складних діаграм, діаграм без зв'язків і діаграм з двома і менше елементами

Розглянемо отриману складну діаграму на рисунку 3.9 після використання алгоритму розташовування. Як можна побачити, що при наявності великої кількості зв'язків та елементів у UML діаграмі, якість використання алгоритму розташовування падає. Зокрема це проявляється в тому, що зв'язки між класами починають перетинатися і втрачається наочність відображення у UML діаграмі

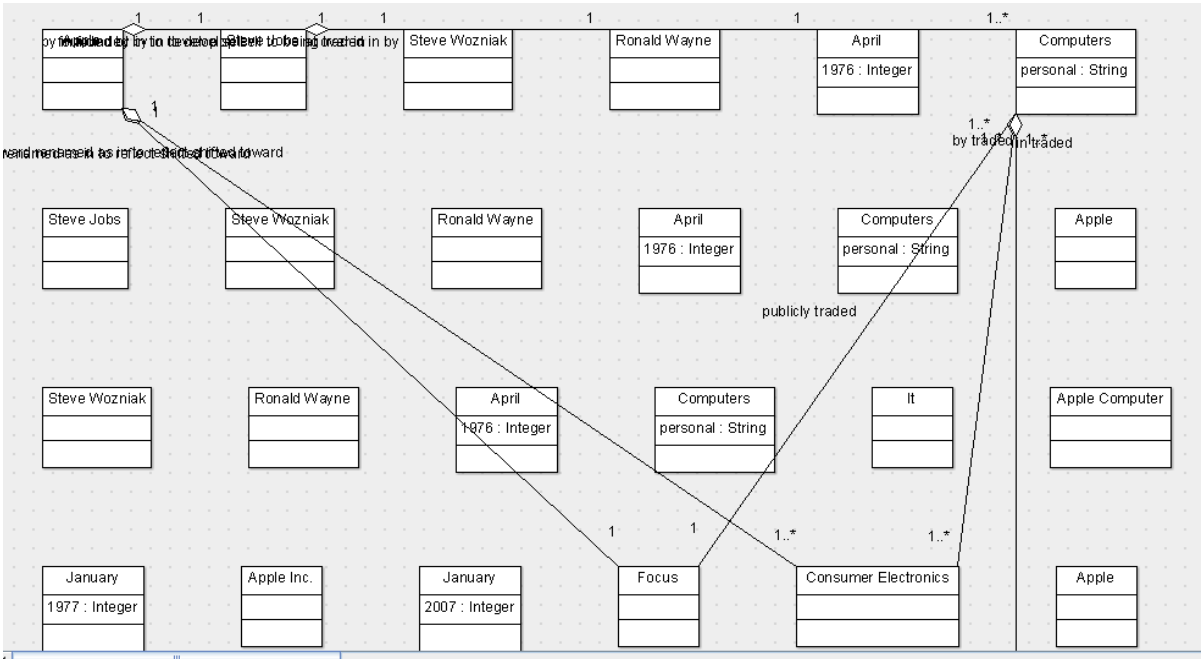


Рисунок 3.9 – Фрагмент UML діаграми з тексту про компанію Apple

Алгоритм впорався з коректним розташовування вершин графів (класів) діаграми без зв’язків, і ми можемо в цьому переконатися на рисунку 3.10. Елементи у UML діаграмі відображені компактно та логічно що в свою чергу, зручно для сприйняття користувачами.

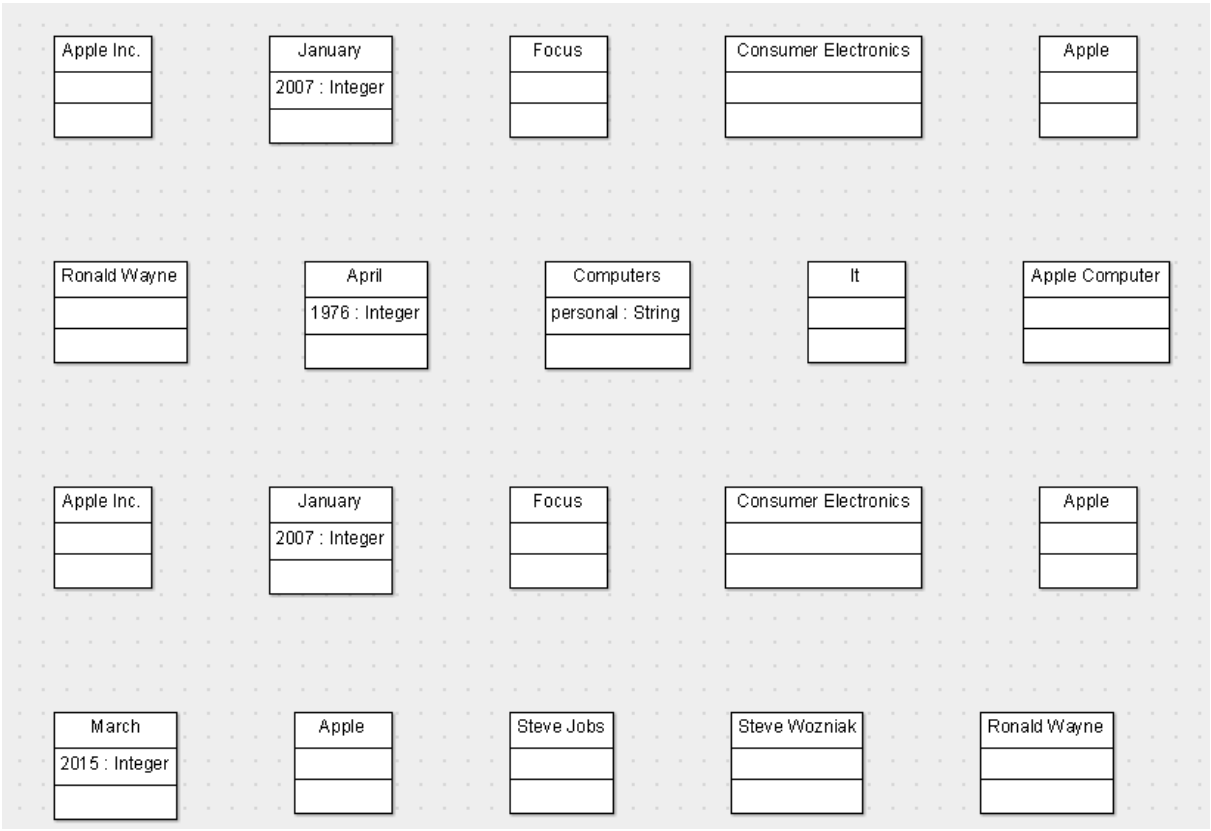


Рисунок 3.10 – UML діаграма без зв’язків з тексту про компанію Apple

При розміщенні двох або менше елементів, адекватність результатів близька до 100%, що є логічним для такої кількості UML елементів. Результати можна бачити на рисунку 3.11.

Рисунок 3.11– UML діаграма з двох елементів

3.3 Аналіз і оцінка

3.3.1 Аналіз візуального представлення UML діаграми після застосування алгоритму розміщення

Загальний результат після застосування алгоритму розміщення позитивний. Найкращі результати були отримані якщо алгоритм був використаний для діаграми з невеликою кількістю елементів. Також проаналізувавши вихідні діаграми можна сказати наступне, що результат розміщення поки не є ідеальним. Діаграми після розміщення елементів потребують додаткового втручання користувача для доопрацювання поточного виду.

Можна стверджувати що після авто-розміщення UML елементів, користувач отримає діаграму у певному візуальному стані, який можна використати як базовий, для проведення подальших дій по оптимізації відображення. Всі наступні дії користувач буде виконувати власноруч.

Основною задачею на сьогодні є скорочення об’єму таких дій до мінімально необхідного.

3.3.2 Оцінка зручності відображення скомпонованої діаграми для користувача

Розглянувши приклади скомпонованих діаграм на рисунку 3.3, рисунку 3.4 та рисунку 3.6 можна стверджувати, що ці діаграми не є зручними для того щоб користувач одразу почав аналіз та редагування. Як було зауважено у розділі 3.3.1 користувач отримає стартове (не ідеальне) відображення, з якого можливо розпочати роботу по дорозміщенню UML елементів.

Якщо ж розглядати діаграму після застосування авто-розміщення як базову, можна стверджувати що UML класи були розміщені правильно,

залежності між класами інколи розташовані невдало, в деяких випадках вони перехрещуються. Додаткові елементи, такі як імена зв’язків, ніяк не позиціонуються у поточній версії програмного продукту.

У ArgoUML відсутня можливість надавати програмне API для згинання ліній зв’язку. Це можливо зробити лише переписавши декілька базових класів для роботи з діаграмами та їх відображенням. Тому на даний час не можливо наочно відображати залежності (ребра) після застосування авто-розміщення. Виправлення цього недоліку заплановано в наступних версіях програмного продукту. Приклад потенційної діаграми з зігнутими лініями зв’язку можна побачити на рисунку 3.12.

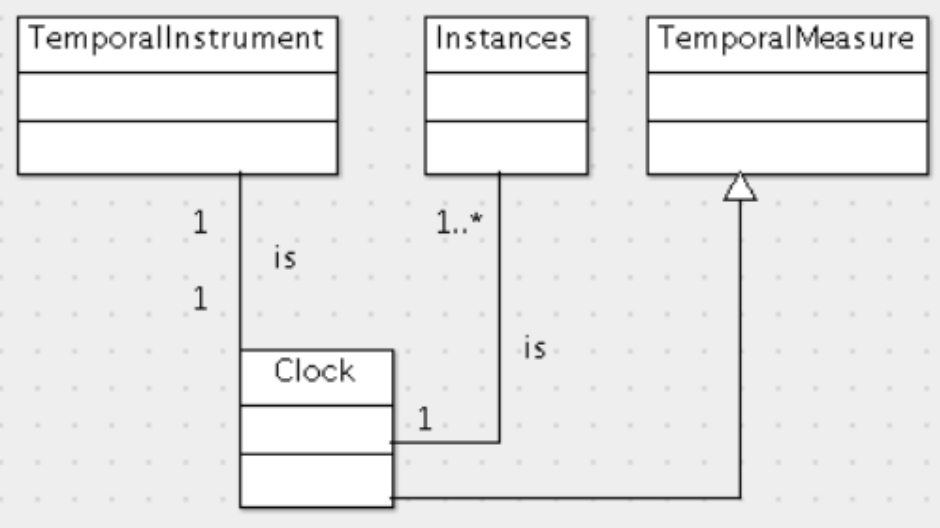


Рисунок 3.12 – Приклад UML діаграми з зігнутими лініями

Наприкінці можна зауважити, що зручність для користувача була досягнута не в повній мірі, особливо з використанням діаграм з великою кількістю елементів, але також треба відзначити що на сьогодні не існує ідеальних рішень для вирішення цієї задачі. Також питання зручності не може бути виражено точною формулою чи коефіцієнтом, інколи це напряму залежить від вподобань конкретного користувача та контексту використання UML діаграми.

3.3.3 Визначення недоліків та помилок розміщення, перелік потенційних покращень

Розглянемо рисунок 3.3. На цьому малюнку можна побачити деякі недоліки які з’явилися при використанні алгоритму авто-розміщення:

а) елементи UML діаграми розміщені в рядок що не є інформативним, та потребує додаткових зусиль для загального розуміння діаграми;

б) назви елементів розташовані дуже близько один біля одного, для них поки що не застосовується алгоритм розміщення, цей недолік буде виправлений у наступних версіях;

в) якщо класи в UML діаграмі розташовані у рядок та між ними є декілька зв’язків як на рисунку 3.3. Стає дуже складно відрізняти до якого саме класу належить зв’язок конкретного типу, для цього потрібно розташовувати класи з невеликим зміщенням відносно один одного, у разі, якщо між сусідніми класами існують зв’язки будь якого типу.

Ці недоліки потрібно виправити в першу чергу тому, що це значно поліпшить розуміння діаграм користувачем після застосування алгоритму авто-розміщення, а також тому, що ці зміни можуть бути запроваджені у ArgoUML без додаткового опору програмного середовища.

Розглянемо список необхідних вдосконалень та розширень які можуть допомогти у покращені розуміння користувачем діаграм:

а) можливість використання різних алгоритмів розташування, для різних частин діаграми. Користувач може відмічати потрібні UML елементи, а потім застосовувати до них різні алгоритми розташування. Це може значно поліпшити вихідні результати, а також додасть гнучкості у процес їх отримання;

б) одним з найскладніших розширень для ArgoUML може бути масштабування (зум), це є дуже корисним для діаграм з великою кількістю

елементів. Також цей тип розширення майже не вплине на логіку роботи алгоритмів розташування;

в) також дуже зручним може бути надання користувачу графічного інтерфейсу для конфігурування деяких параметрів алгоритму. Наприклад: мінімальні відступи від UML елементів, встановлення ваги за замовчуванням для деяких типів елементів, відступи від зв’язків, максимальна довжина рядків для розташування класів, та інші;

г) для зручності користувача може буде надана можливість згинати лінії зв’язку між елементами UML діаграм, це потрібно для більш наочного представлення діаграми та для більш гнучких видів розташування елементів.

ВИСНОВКИ

У першій частині були отриманні результати з основних теоретичних понять про UML діаграми та їх властивості. Результати стосовно алгоритмів і особливостей їх розробки та основних критеріїв вибору підходящого алгоритму. Розглянуто тему графів та існуючі методи їх візуалізації, отримано порівняння найпоширеніших з них, враховуючи особливості їх відображення та сфери застосування.

У другій частині роботи були розглянуті особливості програмної реалізації алгоритму для автоматичного розташування елементів UML діаграм. На першому етапі були проведені дослідження існуючого алгоритму ArgoUML. На другому і третьому етапах були розглянуті алгоритми Orthogonal Layout і Sugiyama, їх особливості та короткий опис принципів роботи. На заключному етапі були розглянуті такі пункти, як впровадження створеного алгоритму у редактор ArgoUML, підключення створених класів та використання у робочій області проекту для автоматичного розташування елементів UML. Також важливою інформацією яка була розглянута є те, що при додаванні класів які розширюють або створюють нові функції у ArgoUML, потрібно дотримуватися існуючої архітектури, а також необхідно провести попередній аналіз коду цього редактору для розуміння схеми взаємодії між класами.

Третя частина цієї роботи містить результати експериментальних досліджень алгоритму для автоматичного розміщення елементів UML діаграм. У результаті проведених експериментів були визначені сильні та слабкі сторони алгоритму та його реалізації у ArgoUML. Також був наданий список потенційних вдосконалень для алгоритму та список розширень для редактору ArgoUML. Грунтуючись на результатах експериментальної частини можна зазначити, що алгоритм справляється з функцією розміщення UML елементів діаграми класів, але має можливості для вдосконалення.

Дивлячись на кінцевий результат можна стверджувати, що мета даної кваліфікаційної роботи була досягнута, але цей результат не повністю вирішує всі проблеми. Подальшим розвитком цього ПЗ, буде надання можливості для користувачів обирати різні типи алгоритмів розташування, як для всієї діаграми так і для окремих елементів. Також обов’язковим покращенням буде надання можливості розташовувати додаткові елементи на діаграмах, такі як імена зв’язків, коментарі та інші. Обов’язковим покращенням буде надання можливості згинати лінії зв’язку між елементами UML діаграм, це потрібно для більш наочного представлення діаграми та для більш гнучких видів розташування елементів.

ПЕРЕЛІК ПОСИЛАНЬ

1. Unified Modeling Language [Електронний ресурс] Режим доступу: https://uk.wikipedia.org/wiki/Unified_Modeling_Language

2. Моделирование на UML [Електронний ресурс] Режим доступу: http://book.uml3.ru/sec_1_4

3. Граф (математика) [Електронний ресурс] Режим доступу: https://uk.wikipedia.org/wiki/Граф_(математика)

4. Алгоритмы на графах - Часть 0: Базовые понятия [Електронний ресурс] Режим доступу: https://habrahabr.ru/post/65367/

5. Граф (математика) [Електронний ресурс] Режим доступу: https://ru.wikipedia.org/wiki/Граф_(математика)

6. UML [Електронний ресурс] Режим доступу: https://ru.wikipedia.org/wiki/UML

7. Алгоритм [Електронний ресурс] Режим доступу: https://uk.wikipedia.org/wiki/Алгоритм

8. Major Layout Algorithms [Електронний ресурс] Режим доступу: http://docs.yworks.com/yfiles/doc/developers-guide/major_layouters.html

9. Візуалізація графів [Електронний ресурс] Режим доступу: https://www.science-community.org/ru/node/5582

10. Проблема визуализации графа [Електронний ресурс] Режим доступу: https://habrahabr.ru/sandbox/30334/

11. Алгоритмы и методы компоновки, размещения и трассировки радиоэлектронной аппаратуры [Електронний ресурс] Режим доступу: http://www.radioland.net.ua/contentid-13-page2.html

12. Нотація Ландау [Електронний ресурс] Режим доступу: https://uk.wikipedia.org/wiki/Нотація _Ландау

13. ArgoUML homepage [Електронний ресурс] Режим доступу: http://argouml.tigris.org/

14. Моісеєнко С.А. Розробка програмного забезпечення для парсингу текстів і генерації UML моделей [Електронний ресурс] Режим доступу: https://github.com/alexnodejs/onto-diploma

15. Шилдт Герберт. Java 8. Руководство для начинающих 6-е издание. Язык русский. Москва: Издательский дом «Вильямс», 2015. 712 с.

16. Дж. Макконелл Анализ алгоритмов. Активный обучающий подход. Москава: Техносфера, 2009. 416с.

17. Robert Frost, Stopping by Woods on a Snowy Evening, [Електронний ресурс] Режим доступу: https://www.poetryfoundation.org/poems-and-poets/poems/detail/42891.

18. Ermolayev V., Keberle N., Jentasch E., Sohnius R.: Performance Simulation Initiative. Upper-Level Ontology v.2.3 Reference Specification, 2009, VCAD EMEA Cadence Design Systems GmbH.

19. Ermolayev V., Copylov A., Keberle N., Jentasch E., Matzke W.-E.: Using Contexts in Ontology Structural Change Analysis, Zaporozhye National University, Cadence Design Systems GmbH.

20. Ermolayev V., Keberle N., Harth A.: Part I: Refining Temporal Representations using OntoElect. In: Extended Semantic Web Conference, 2016 Heraklion, Crete, May 29, 2016.

21. Ermolayev V.: Gravitation and Fitness in Ontology Dynamics. In: KIT, AIFB: Kolloquium Angewandte Informatik Jan. 26, 2016, Karlsruhe, Germany

22. Vadim Ermolayev V.: Toward a Syndicated Ontology of Time for the Semantic. In: KIT, AIFB February, 2016, Karlsruhe, Germany

23. Гради Буч, Джеймс Рамбо, Ивар Якобсон Язык UML. Руководство пользователя. 2-е изд.: Пер. с англ. Мухин Н. Москва: ДМК Пресс, 2006. 496 с.: ил.

Додаток А

```
private void rankAndWeightNodes() {
      List<ClassdiagramNode> comments =
                  new ArrayList<ClassdiagramNode>();
      nodeRows.clear();

      TreeSet<ClassdiagramNode> nodeTree =
                  new TreeSet<ClassdiagramNode>(layoutedClassNodes);

      for (ClassdiagramNode node : nodeTree) {
            if (node.isComment()) {
                  comments.add(node);
            } else {
                  int rowNum = node.getRank();

                  for (int i = nodeRows.size(); i <= rowNum; i++) {
                        nodeRows.add(new Node(rowNum));
                  }


                  nodeRows.get(rowNum).addNode(node);
            }
      }

      for (ClassdiagramNode node : comments) {
            int rowInd = node.getUpNodes().isEmpty() ?
                        0 : ((node.getUpNodes().get(0)).getRank());

            if (nodeRows.size() == 0) {
                  nodeRows.add(new Node(0));
            }
            nodeRows.get(rowInd).addNode(node);
      }

      for (int row = 0; row < nodeRows.size();) {
            Node diaRow = nodeRows.get(row);
            diaRow.setRowNumber(row++);
            diaRow = diaRow
                        .doSplit(MAX_ROW_WIDTH, HORIZONTAL_NODE_GAP);

            if (diaRow != null) {
                  nodeRows.add(row, diaRow);
            }
      }
}
```

Додаток Б

```
private void setupLinks() {
           figNodes.clear();
           HashMap<Fig, List<ClassdiagramInheritanceEdge>>
figParentEdges =
     new HashMap<Fig, List<ClassdiagramInheritanceEdge>>();

           for (ClassdiagramNode node : layoutedClassNodes) {
                node.getUpNodes().clear();
                node.getDownNodes().clear();
                figNodes.put(node.getFigure(), node);
           }

           for (ClassdiagramEdge edge : layoutedEdges) {
                Fig parentFig = edge.getDestFigNode();
                ClassdiagramNode child =
                     figNodes.get(edge.getSourceFigNode());
                ClassdiagramNode parent = figNodes.get(parentFig);

                if (edge instanceof ClassdiagramInheritanceEdge) {
                     if (parent != null && child != null) {
                     parent.addDownlink(child);
                     child.addUplink(parent);
                     List<ClassdiagramInheritanceEdge> edgeList =
figParentEdges.get(parentFig);

                     if (edgeList == null) {
                          edgeList =
                     new ArrayList<ClassdiagramInheritanceEdge>();
                          figParentEdges.put(parentFig, edgeList);
                     }

                     edgeList.add((ClassdiagramInheritanceEdge)
edge);
                     } else {
                     LOG.log(Level.SEVERE, "Edge with missing end(s):
" + edge);
                     }
                } else if (edge instanceof ClassdiagramNoteEdge) {
                     if (parent.isComment()) {
                          parent.addUplink(child);
                     } else if (child.isComment()) {
                          child.addUplink(parent);
                     } else {
                          LOG.log(Level.SEVERE, "Unexpected
parent/child constellation for edge: " + edge);
                     }
                } else if (edge instanceof
ClassdiagramAssociationEdge)
                     // TODO: Create appropriate ClassdiagramEdge
                } else {
                     LOG.log(Level.SEVERE, "Unsupported edge type");
```