МАГІСТЕРСЬКА РОБОТА

МР.ІПм - 06.00.00.000- ПЗ

Група ІПм-24-1

Василик Віктор

2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Василик Віктор Михайлович

(прізвище, ім’я, по батькові)

УДК ____________
(індекс)

МАГІСТЕРСЬКА РОБОТА

Методи візуально орієнтованого модельно-орієнтованого інжинірингу для архітектурного моделювання програмного забезпечення

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

Василик В. М.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Козак Олексій Федорович, канд. тех. н, ст. викл.

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

Допущено до захисту

Завідувач кафедри

доц. Бандура В. В.

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

Нормоконтроль

доц. Вовк Р. Б.

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

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

Івано-Франківськ – 2025

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Завідувач кафедри ІПЗ
доц. Бандура В. В.
„ 04 ” вересня 2025р.

ЗАВДАННЯ
НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Василику Віктору Михайловичу
(прізвище, ім'я, по батькові)

1. Тема магістерської роботи „Методи візуально орієнтованого модельно-орієнтованого інжинірингу для архітектурного моделювання програмного забезпечення”

керівник проєкту (роботи) Козак Олексій Федорович, к.т.н., старший викладач

затверджені наказом вищого навчального закладу від „ 05 ” 11 20 25 р. № 695/7

2. Строк здачі студентом закінченої роботи 15 грудня 2025 р.

3. Вихідні дані до роботи методи інженерії керованої моделями, методи архітектурного моделювання програмного забезпечення, методологія розробки з візуальними мовами програмування, результати наукового дослідження „Теоретичні аспекти візуальних мов програмування”

4. Зміст пояснювальної записки (перелік питань, що їх належить розробити)
 1. Дослідження предметної області та постановка задачі
 2. Концептуальне проєктування інструменту візуального модельно-орієнтованого інжинірингу
 3. Технічна реалізація архітектури та механізмів інструменту
 4. Експериментальна оцінка та приклади використання інструменту

5. Перелік графічного матеріалу (з точним забезпеченням обов'язкових креслень)
 1. Візуальна нотація інструменту для архітектурного моделювання (рис. 3.1, ст. 86)
 2. Візуальна нотація інструменту елементів програмної логіки (рис. 3.2, ст. 92)
 3. Компонентна діаграма архітектури інструменту (рис. 4.1, ст. 98)
 4. Демонстрація функціональних можливостей інструменту (рис. 4.8, ст. 105)

6. Консультанти з роботи із зазначенням розділів роботи, що їх стосуються

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Перевірка на плагіат	доц., к.т.н. Вовк Р. Б.		

7. Дата видачі завдання 4 вересня 2025 р.

Керівник ______________ ____________________
(підпис) (розшифровка підпису)

Завдання прийняв до виконання ______________ ____________________
(підпис) (розшифровка підпису)

КАЛЕНДАРНИЙ ПЛАН

Номер і назва етапів магістерської роботи	Термін виконання етапів роботи	Примітка
1 Підбір і вивчення літератури	03.11.2025 – 05.11.2025	виконано
2 Дослідження предметної області та постановка задачі	06.11.2025 – 16.11.2025	виконано
3 Концептуальне проєктування інструменту візуального модельно-орієнтованого інжинірингу	17.11.2025 – 24.11.2025	виконано
4 Технічна реалізація архітектури та механізмів інструменту	25.11.2025 – 30.11.2025	виконано
5 Експериментальна оцінка та підготовка прикладів використання інструменту	01.12.2025 – 07.12.2025	виконано
6 Оформлення пояснювальної записки згідно з вимогами	06.11.2025 – 14.12.2025	виконано
7 Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2025	виконано

Студент – магістр ________________________________
(підпис) (розшифровка підпису)

Керівник проєкту ________________________________
(підпис) (розшифровка підпису)

АНОТАЦІЯ

Магістерська робота: 119 с., 28 рис., 13 табл., 50 джерел.

Тема: Методи візуально орієнтованого модельно-орієнтованого інжинірингу для архітектурного моделювання програмного забезпечення.

Об'єкт дослідження: процес архітектурного моделювання програмного забезпечення в контексті сучасних методологій розробки.

Мета роботи: розробка методів візуального модельно-орієнтованого інжинірингу для архітектурного моделювання програмного забезпечення, що поєднують сильні сторони модельно-орієнтованої інженерії та візуальних мов програмування з урахуванням когнітивних обмежень розробників.

Предмет дослідження: методи візуального модельно-орієнтованого інжинірингу для підвищення ефективності архітектурного проєктування складних програмних систем.

Результати дослідження: Здійснено аналітичний огляд 26 інструментів архітектурного моделювання та виявлено 14 ключових проблем. Запропоновано нову парадигму «Когнітивно-ергономічна холістична візуальна парадигма розробки» (CEHVDP). Розроблено візуальні нотації для архітектурного моделювання та для програмної логіки. Реалізовано прототип інструменту «BioFusion IDE» з механізмами валідації когнітивних обмежень та генерації структури Java-проєктів.

Висновок: Розроблений прототип підтверджує життєздатність запропонованого підходу та демонструє можливість поєднання візуального редагування архітектури з автоматичною генерацією програмного коду. Результати дослідження можуть бути використані при створенні нового покоління інструментів для архітектурного проєктування програмного забезпечення з урахуванням когнітивних обмежень розробників.

АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ВІЗУАЛЬНЕ МОДЕЛЮВАННЯ, ВІЗУАЛЬНІ МОВИ ПРОГРАМУВАННЯ, КОГНІТИВНА ЕРГОНОМІКА, МОДЕЛЬНО-ОРІЄНТОВАНИЙ ІНЖИНІРИНГ, CEHVDP, VPL.

ABSTRACT

Master's thesis: 119 p., 28 fig., 13 tab., 50 refs.

Topic: Methods of visually oriented model-driven engineering for architectural modeling of software.

Object of research: the process of architectural modeling of software in the context of modern development methodologies.

Purpose of the work: to develop methods of visual model-oriented engineering for architectural modeling of software that combine the strengths of model-oriented engineering and visual programming languages, taking into account the cognitive limitations of developers.

Subject of research: methods of visual model-oriented engineering to improve the efficiency of architectural design of complex software systems.

Research results: An analytical review of 26 architectural modeling tools was conducted and 14 key problems were identified. A new paradigm, the Cognitive-Ergonomic Holistic Visual Development Paradigm (CEHVDP), was proposed. Visual notations for architectural modeling and software logic were developed. A prototype of the «BioFusion IDE» tool was implemented with mechanisms for validating cognitive constraints and generating Java project structures.

Conclusion: The developed prototype confirms the viability of the proposed approach and demonstrates the possibility of combining visual architecture editing with automatic software code generation. The research results can be used to create a new generation of tools for architectural software design, taking into account the cognitive limitations of developers.

CEHVDP, COGNITIVE ERGONOMICS, MODEL-DRIVEN ENGINEERING, SOFTWARE ARCHITECTURE, VISUAL MODELING, VISUAL PROGRAMMING LANGUAGES, VPL.

ЗМІСТ

Стр.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ........8
ВСТУП........9

РОЗДІЛ 1
ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СУМІЖНИХ НАПРЯМІВ........14
1.1 Основи програмної інженерії, керованої моделями, та окреслення дослідницьких напрямів........14
1.2 Класифікація інструментарію та методів розробки програмного забезпечення: історичний огляд та еволюція........23
1.2.1 Сучасний стан інструментарію розробки програмного забезпечення........23
1.2.2 Класифікація мов програмування за поколіннями........26
1.2.3 Хронологія еволюції методологій розробки програмного забезпечення....29
1.2.4 Концепція Software 1.0, 2.0, 3.0........36
1.3 Інтеграційні методи архітектурного проєктування........37
1.3.1 Модельно-орієнтовані підходи: уточнення термінології........37
1.3.2 Візуальні мови програмування та мови моделювання........41
1.4 Інноваційні технологічні методи проєктування........43
1.4.1 Предметно-орієнтоване проєктування: концептуальні засади........43
1.4.2 Low-code та No-code платформи розробки........45
1.4.3 AI-driven розробка: еволюція парного програмування........46
1.5 Висновки до розділу........48

РОЗДІЛ 2
ОГЛЯД ВІЗУАЛЬНИХ МЕТОДІВ МОДЕЛЬНО-ОРІЄНТОВАНОГО ІНЖИНІРИНГУ ТА ПОСТАНОВКА ЗАДАЧІ........49
2.1 Огляд візуальних інструментів та методів для архітектурного моделювання програмного забезпечення........49
2.2.Сучасний стан архітектурного моделювання програмного забезпечення........66

2.3 Тенденції та перспективи розвитку інструментів моделювання........................67
2.4 Підсумки огляду візуальних інструментів та методів...69
2.5 Висновки до розділу..73

РОЗДІЛ 3
КОНЦЕПТУАЛЬНЕ ПРОЄКТУВАННЯ ІНСТРУМЕНТУ ВІЗУАЛЬНОГО МОДЕЛЬНО-ОРІЄНТОВАНОГО ІНЖИНІРИНГУ..75
3.1 Аналіз виявлених проблем, концептуалізація проєктованого рішення.............75
3.2 Проєктування архітектурного представлення...84
3.3 Проєктування представлення програмної логіки...89
3.4 Висновки до розділу..96

РОЗДІЛ 4
РЕАЛІЗАЦІЯ І ТЕСТУВАННЯ ІНСТРУМЕНТУ ВІЗУАЛЬНОГО МОДЕЛЬНО-ОРІЄНТОВАНОГО ІНЖИНІРИНГУ..97
4.1 Технічна реалізація архітектури та механізмів інструменту..............................97
4.2 Експериментальна оцінка та приклади використання інструменту.................102
4.3 Висновки до розділу..108

ВИСНОВКИ...109
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ...111
ДОДАТКИ...117

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

AaC	–	Architecture as Code
API	–	Application Programming Interface
BPMN	–	Business Process Model and Notation
CASE-засоби	–	засоби автоматизованої розробки ПЗ
CI/CD	–	Continuous Integration / Continuous Delivery (Deployment)
DDD	–	Domain-Driven Design
DSL	–	Domain-Specific Language
Ecore	–	метамодель у Eclipse Modeling Framework (EMF)
IDE	–	Integrated Development Environment
MBE	–	Model-Based Engineering
MBSE	–	Model-Based Systems Engineering
MDA	–	Model-Driven Architecture
MDD	–	Model-Driven Development
MDE	–	Model-Driven Engineering
MOF	–	Meta-Object Facility стандарт
OMG	–	Object Management Group консорціум
PMI	–	Platform Independent Model
PSM	–	Platform Specific Model
QVT	–	Query View Transformation (стандарт OMG)
RAD	–	Rapid Application Development
SysML	–	Systems Modeling Language стандарт
TPL	–	Text Programming Language
UML	–	Unified Modeling Language стандарт
VPL	–	Visual Programming Language
WYSIWYG	–	принцип роботи програмних інтерфейсів «Що бачиш, те й отримуєш»
XMI	–	XML Metadata Interchange (стандарт OMG)

ВСТУП

Сутність і стан наукової проблеми. Сучасна інженерія програмного забезпечення характеризується стрімким зростанням складності систем, необхідністю забезпечення їхньої гнучкості, масштабованості та підтримки протягом усього життєвого циклу. У таких умовах традиційні текстові та кодово-орієнтовані підходи до проєктування та документування архітектури програмного забезпечення дедалі частіше виявляються недостатніми, оскільки вони ускладнюють розуміння системи різними зацікавленими сторонами, підвищують ризик помилок при передачі знань і ускладнюють еволюцію архітектури. Як відповідь на ці виклики сформувався та набув широкого визнання модельно-орієнтований інжиніринг програмного забезпечення, основна ідея якого полягає в тому, щоб вважати моделі первинними артефактами розробки, а код та інші технічні деталі є похідними від них. Ця парадигма ставить моделі в центр процесу розробки програмного забезпечення, де замість ручного написання коду розробники створюють абстрактні моделі системи, а код або інші артефакти генеруються автоматично або напівавтоматично з цих моделей. На сучасному етапі розвитку інженерії програмного забезпечення спостерігається чітке розділення внутрішніх підходів для реалізації модельно-орієнтованого інжинірингу. Одні інструменти та методології зосереджуються переважно на моделюванні архітектури високого рівня, але ігнорують або спрощують питання детального алгоритмічного програмування та автоматичної генерації виконуваного коду. З іншого боку, візуальне програмування, блокові середовища, малокодові платформи та сучасні асистенти на основі штучного інтелекту відмінно справляються з швидкою генерацією коду та прототипуванням, але практично не надають засобів контролю та документування архітектури системи загалом. Проведений у межах дослідження аналіз 26 інструментів за 12 категоріями виявив критичну прогалину, жоден розглянутий інструмент не поєднує одночасно: візуальне редагування з низьким когнітивним навантаженням, формальну метамодель для сумісності з модельно-орієнтованим інжинірингом, двонаправлену синхронізацію з програмним кодом та генерацію виконуваних артефактів. Зокрема,

показник двонаправленої синхронізації між архітектурною моделлю та програмним кодом становить 0% для повної синхронізації та лише 23% для часткової, що свідчить про фундаментальну невирішеність цієї проблеми.

Актуальність теми дослідження обумовлена кількома ключовими факторами. Когнітивні обмеження розробників суттєво впливають на продуктивність та якість роботи інженера при проєктуванні складних систем. Фундаментальні дослідження когнітивної психології встановили, що апаратна межа робочої пам'яті людини становить приблизно 4 одиниці інформації, що суттєво нижче за класичне уявлення про «магічне число 7±2» [1]. Це означає, що архітектура, яка змушує утримувати у фокусі 5 і більше незалежних сутностей одночасно, гарантовано призводить до когнітивного перевантаження та помилок. Подальші дослідження продемонстрували, що місткість робочої пам'яті не є фіксованою величиною, а обернено пропорційна складності об'єктів: для простих об'єктів вона становить 4–5 елементів, тоді як для складних об'єктів з багатьма ознаками місткість падає до 1–2 елементів [2]. Це пояснює, чому складний клас із заплутаною логікою сприймається як «важкий» об'єкт, і розробник може оперувати лише двома такими об'єктами одночасно. Сучасні експериментальні дослідження підтверджують ресурсну модель когнітивного навантаження: прості об'єкти з однією ознакою можуть зберігатися у робочій пам'яті у кількості понад 5 елементів, тоді як складні об'єкти з кількома ознаками обмежують місткість приблизно до 2 елементів [3]. Це підтверджує ресурсну модель когнітивного навантаження, де складніші об'єкти споживають більше когнітивних ресурсів, що має принципове значення для проєктування інструментів архітектурного моделювання [4-6]. Робоча пам'ять функціонує як мультимодальна система з об'єктно-орієнтованим, ознаково-орієнтованим та просторовим збереженням, що має принципове значення для інструментів моделювання, де елементи є частинами складної просторової структури. Іншим ключовим фактором є швидкість обробки текстової інформації, значно нижча за візуальну. Візуальні нотації дозволяють знизити когнітивне навантаження завдяки преатентивній обробці [7], принципам гештальт-психології [8] та зовнішньому когнітивному розвантаженню [9]. Водночас статистичний аналіз

демонструє парадокс: хоча 58% інструментів мають формальну метамодель, лише 19% забезпечують повноцінну генерацію коду. Існує обернена кореляція між потужністю інструменту та його когнітивною доступністю. Ця ситуація обґрунтовує необхідність розробки нового інтеграційного підходу з урахуванням ресурсної моделі когнітивного навантаження.

Мета дослідження полягає у розробці методів візуального модельно-орієнтованого інжинірингу для архітектурного моделювання програмного забезпечення, що поєднують сильні сторони модельно-орієнтованої інженерії та візуальних мов програмування з урахуванням когнітивних обмежень розробників.

Для досягнення поставленої мети даної наукової роботи було сформовано наступні **завдання дослідження**:

– здійснити аналітичний огляд сучасного стану інструментарію розробки та окреслити хронологічну еволюцію методологій розробки програмного забезпечення;

– визначити вплив розробки з використанням штучного інтелекту, модельно-орієнтованих підходів та предметно-орієнтованого проєктування на розв'язок окреслених вище проблеми;

– проаналізувати сучасні підходи до архітектурного моделювання програмного забезпечення, тенденції та перспективи розвитку інструментів розробки програмного забезпечення;

– здійснити огляд візуальних інструментів та методів для архітектурного моделювання програмного забезпечення;

– концептуалізувати розв'язок окреслених вище проблем архітектурного моделювання програмного забезпечення в єдине рішення;

– здійснити реалізацію сформованої концепції інструменту візуального модельно-орієнтованого інжинірингу;

– надати експериментальну оцінку реалізованому рішенню.

Об'єктом дослідження є процес архітектурного моделювання програмного забезпечення в контексті сучасних методологій розробки.

Предметом дослідження є методи візуального модельно-орієнтованого інжинірингу для підвищення ефективності архітектурного проєктування складних програмних систем.

Методи дослідження. У роботі застосовано комплекс методів наукового дослідження. Аналітичний огляд використано для систематизації сучасного стану інструментарію розробки програмного забезпечення та виявлення ключових тенденцій розвитку галузі. Порівняльний аналіз застосовано для оцінки 26 інструментів за 12 категоріями з визначенням їх сильних та слабких сторін за критеріями наявності метамоделі, можливостей генерації коду, двонаправленої синхронізації, когнітивної складності та типу візуалізації. Метод систематизації використано для класифікації підходів до модельно-орієнтованого інжинірингу та візуального програмування. Концептуальне моделювання застосовано для формування теоретичних засад інтеграційного підходу, що поєднує переваги модельно-орієнтованої інженерії та візуальних мов програмування. Експериментальні дослідження проведено для верифікації розробленого прототипу інструменту та оцінки його ефективності порівняно з розглянутими рішеннями.

Наукова новизна одержаних результатів полягає в наступному. Покращено інтеграційний підхід візуального модельно-орієнтованого інжинірингу, що поєднує принципи модельно-орієнтованої архітектури з когнітивно-ергономічними практиками візуальних мов програмування. Удосконалено методи архітектурного моделювання шляхом врахування когнітивних обмежень розробників. Запропоновано парадигму «Когнітивно-ергономічна холістична візуальна парадигма розробки», що інтегрує когнітивну ергономіку, холістичне моделювання та візуальні мови програмування. Дістала подальшого розвитку концепція застосування візуальних мов програмування для архітектурного моделювання, що розширює результати попереднього дослідження та адаптує їх для потреб модельно-орієнтованого інжинірингу.

Практичне значення одержаних результатів. Розроблено прототип інструменту візуального модельно-орієнтованого інжинірингу версії 1.0.0, що реалізує сформовану концепцію та демонструє можливість поєднання візуального

редагування архітектури з автоматичною генерацією програмного коду. Сформульовано методичні рекомендації щодо застосування когнітивно-ергономічних принципів при розробці інструментів архітектурного моделювання. Результати дослідження можуть бути використані при створенні нового покоління інструментів для архітектурного проєктування програмного забезпечення, що враховують природні когнітивні обмеження розробника.

Особистий внесок. Усі результати, викладені в магістерській роботі, отримані автором особисто. Автором самостійно проведено аналітичний огляд 26 інструментів архітектурного моделювання, розроблено концепцію інтеграційного підходу візуального модельно-орієнтованого інжинірингу, здійснено реалізацію прототипу інструменту та проведено експериментальну оцінку отриманих результатів. Дана наукова робота має спільний теоретичний фундамент зі статтею «Теоретичні аспекти візуальних мов програмування» [10], де автором було сформульовано архітектуру для розробки середовищ візуальних мов програмування та список принципів розробки програмного забезпечення із застосуванням візуальних мов. Частина попередньо отриманих результатів адаптована й покращення в цій науковій роботі.

Структура магістерської роботи.

Магістерська робота викладена на 119 сторінках друкованого тексту, який складається з вступу, трьох розділів, висновків, списку використаних джерел (50 найменувань). Робота містить 13 таблиць та 28 рисунків.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА СУМІЖНИХ НАПРЯМІВ

1.1 Основи програмної інженерії, керованої моделями, та окреслення дослідницьких напрямів

Стрімке зростання складності програмних систем та обмеженість традиційних кодово-орієнтованих підходів зумовили перехід до модельно-орієнтованого інжинірингу (Model-Driven Engineering, MDE). Ця парадигма зміщує фокус розробки з ручного написання коду на створення абстрактних моделей, які стають первинними артефактами. У рамках MDE саме моделі, а не код, слугують основою для автоматичної генерації системи, її документування та подальшої еволюції, що дозволяє нівелювати розрив між проєктуванням та реалізацією [11].

Методи роботи з парадигмою MDE можуть включати, як побудову моделей завдяки опису їх в текстовому форматі, так і одразу через візуальні об’єкти/конструкції. В даній науковій роботі фокус робиться саме на методи візуального модельно-орієнтованого інжинірингу через бажання досягти унікальних корисних результатів із заздалегідь визначеними характеристиками. Такі методи можна визначити, як сукупність технологій і практик, що використовують графічні (візуальні) нотації та діаграми як основний засіб специфікації, аналізу, проєктування, верифікації та трансформації архітектурних моделей.

Визначені методи дозволяють створювати абстрактні, зрозумілі всім учасникам проєкту представлення архітектури, автоматично генерувати з них код, конфігурації, документацію, а також виконувати перевірку властивостей системи ще на етапі моделювання. Найвідомішими прикладами є мови архітектурного моделювання (ArchiMate[12] у частині IT-систем, частково SysML[13], BPMN[14] у частині архітектури), а також стандартизована OMG-нотація UML 2.5 [15] та інструментальні платформи (Eclipse Modeling Project, Sparx Enterprise Architect, Visual Paradigm, Modelio). Довідка про деякі зі згаданих засобів та інструментів буде наведена при їх подальшому розгляді.

На сучасному етапі розвитку інженерії програмного забезпечення спостерігається чітке розділення внутрішніх підходів для реалізації MDE. Одні інструменти та методології (наприклад, класичні CASE-засоби, Archi, Sparx Enterprise Architect) зосереджуються переважно на моделюванні архітектури високого рівня, але ігнорують або спрощують питання детального алгоритмічного програмування та автоматичної генерації виконуваного коду. З іншої сторони візуальне програмування [10], блокові середовища (Blockly, App Inventor) [16], low-code/no-code платформи (OutSystems, Mendix) [17] та сучасні AI-асистенти (GitHub Copilot, Claude) відмінно справляються зі швидкою генерацією коду та прототипуванням, але практично не надають засобів контролю та документування архітектури системи загалом. У результаті на платформах розробки без коду відповідальність за архітектурну якість та безпеку перекладається на власника, а при використанні ШІ-генерації коду архітектурний аналіз знову повертається до ручного перегляду великих обсягів згенерованого тексту, виявлення вразливостей, оцінки людиноцентричності та усунення прихованих дефектів, тобто інженер змушений виконувати всю «другу половину» роботи, яку автоматизація поки що не здатна гарантовано виконати на відмінному рівні.

Саме тому в магістерській роботі пропонується інтеграційний підхід, що поєднує сильні сторони візуального модельно-орієнтованого інжинірингу та візуального програмування з автоматичною генерацією коду. Для обґрунтування та розробки такого підходу необхідний ґрунтовний порівняльний аналіз двох великих груп методів. Перша група – це методи й інструментів архітектурного моделювання (UML, ArchiMate, SysML, C4 [18], модель 4+1 [19]). Друга – це методи й інструменти візуальної та модельно-орієнтованої генерації коду (Acceleo [20], ATL[21], low-code платформи, середовища візуальних мов програмування). Такий підхід дозволить виявити слабкі сторони обох сторін, запропонувати єдине рішення, визначити точки інтеграції, а надалі створити єдину методологію, яка дозволить одночасно забезпечувати контрольовану, документовану архітектуру та ефективну автоматизовану реалізацію програмних компонентів.

Дана наукова робота має спільний теоретичний фундамент зі статтею «Теоретичні аспекти візуальних мов програмування» [10], і є продовження цього дослідження, переслідує спільні цілі, а також розширює їх до більшого списку. В статті автори поставили собі питання, чи має вищий рівень абстракції візуальних мов програмування (Visual programming languages, VPLs) переваги над текстовими мовами (Textual programming languages, TPLs). Також, з корисних артефактів наведено нову архітектуру за якою мали б розроблятися середовища VPLs та список принципів по розробці ПЗ із застосуванням візуальних мов програмування. Частина попередньо отриманих результатів буде адаптована для цієї наукової роботи.

Отже, тема магістерської роботи «Методи візуального орієнтованого модельно-орієнтованого інжинірингу для архітектурного моделювання програмного забезпечення» має на меті поєднати сильні сторони MDE та VPLs із врахуванням когнітивно-ергономічного дизайну. Вважається, це дасть наступних гіпотетичних шість переваг.

По-перше, когнітивні обмеження впливають на процес розробки, серед них: людська пам’ять, концентрація, увага, навігація, швидкість мислення тощо. При роботі з програмним кодом чи текстом мозок змушений обробляти інформацію як послідовність символів. Це робить процес сприйняття повільнішим і вразливим до помилок, адже навіть невелике порушення послідовності призводить до синтаксичних збоїв. Натомість візуальні елементи мозок сприймає як цілісні образи. Завдяки цьому вони обробляються швидше й ефективніше, бо не потребують покрокового аналізу кожного символу. Інструменти MDE та їх інтегровані середовища розробки (Integrated Development Environment, IDE) повинні враховувати ергономічність для сприяння зручності при роботі з ними, а також особливості роботи мозку людини, пам’яті в тому числі. Отже, враховуються переваги візуального підходу з позиції когнітивної ергономіки та обмежень людського сприйняття. Людський мозок має чітко підтверджені наукою когнітивні обмеження, які суттєво впливають на продуктивність та якість роботи інженера при проєктуванні складних систем:

– об’єм короткочасної (робочої) пам’яті за сучаснішими дослідженнями становить 4-5 нескладних об’єктів одночасно [1-2] та приблизно 2 складних [3];

– швидкість обробки текстової інформації значно нижча, ніж візуальної, мозок розпізнає та інтерпретує зображення за 13–100 мс [22], інше дослідження стверджує значення 95–100 мс [23], тоді як читання та розуміння текстового коду вимагає сотень мілісекунд на одне слово;

– увага та концентрація швидко виснажуються при необхідності утримувати в голові велику кількість зв’язків між сутностями, особливо якщо ці зв’язки представлені лише текстом або розкидані по багатьох файлах, а також переключення між контекстами потребує часу;

– навігація по тисячам рядків коду або десяткам текстових описів архітектури призводить до значного когнітивного навантаження, зростання кількості помилок і втрати контексту.

Візуальні нотації та діаграми дозволяють радикально знизити це навантаження завдяки таким ефектам:

1. Преатентивна обробка – мозок миттєво сприймає форму, колір, положення, розмір, групування без свідомих зусиль. Також сюди включають здатність мозку передбачувати майбутні події або інформацію на основі попереднього досвіду та шаблонів у пам’яті, що забезпечує швидше й ефективніше реагування на подразники середовища розробки.

2. Закон близькості та закон спільної долі (принципи Гештальт-психології), де перше стверджує, що елементи, які розташовані близько один до одного, сприймаються як єдине ціле або група. А друге, описує явище, коли об’єкти, що рухаються або змінюються в однаковому напрямку чи з однаковою швидкістю, сприймаються як частина одного цілого.

3. Зовнішнє когнітивне розвантаження – це стратегія, при якій людина зменшує навантаження на власну робочу пам'ять, перекладаючи частину когнітивної обробки на зовнішні носії, в даному випадку, моделі, діаграми та схеми. Замість зберігання та маніпулювання інформацією в голові (внутрішня робоча пам'ять), інформація фіксується або організується у зовнішньому середовищі. Просторова

пам'ять та навігаційні схеми (ментально або фізично організовані простори, карти, об'єкти в області) можуть виступати зовнішніми «сховищами» для когнітивних процесів мозку.

Тому сучасні інструменти візуального модельно-орієнтованого інжинірингу мають свідомо враховувати ці обмеження і застосовувати найкращі ергономічні практики, зокрема:

– мінімалістичний інтерфейс (принцип «менше є добре»);

– чітке розділення робочих просторів, окреме вікно/вкладка для структури системи, окреме для поведінки, інше для детального вмісту вибраного елемента;

– суворе логічне групування та доменне розшарування (програмні пакети розглядаються, як простір зв'язаних об'єктів);

– ієрархічна декомпозиція з можливістю «згортання» підсистем (просування до деталей вглиб тільки за потребою);

– заборона одночасного відображення більше одного рівня ієрархії (наприклад, не показувати одночасно батьківські класи, поточний клас і всі його дочірні на одній діаграмі);

– використання кольорового кодування лише за строгими правилами (не більше 5–6 кольорів);

– автоматичне приховування другорядних деталей до моменту наведення або явного запиту.

Використання саме таких принципів когнітивно-ергономічного дизайну дає змогу інженеру утримувати в робочій пам'яті лише 4-5 активних елементів замість десятків або сотень, різко знижує ймовірність помилок і підвищує швидкість проєктування архітектури складних систем у рази.

По-друге, робота у справді візуальному, а не гібридному (комбінація тексту та діаграм) середовищі суттєво прискорює більшість повсякденних операцій розробки та архітектурного проєктування завдяки прямій маніпуляції цілісними об'єктами замість посимвольного редагування тексту. Конкретні ефекти, які очікуються від поєднання сильних сторін інженерії керованої моделями та сучасного покоління візуальних мов програмування:

– операції переміщення, копіювання, клонування, групування/розгрупування, зміна ієрархії чи пакетів виконуються одним-двома натисканнями або «drag-and-drop» замість багатоступеневих текстових команд (виділення → копіювати → перейти в інший файл → вставити → виправити імпорти/шляхи → виправити простір імен);

– автоматичне налагодження при переміщенні об'єктів (оновлення всіх залежностей, імпортів, просторів імен, маршрутів у мікросервісах тощо) виконується інструментом у фоновому режимі без участі розробника;

– створення зв'язків між компонентами (асоціації, залежності, реалізація інтерфейсу, виклик сервісу) відбувається простим з'єднанням лінією або притягуванням одного об'єкта до іншого, а не пошуком та ручним прописуванням сигнатур та типу з'єднання;

– зміна типу зв'язку, напрямленості чи стереотипу є вибором з контекстного меню, а не редагуванням тексту анотацій чи профілів;

– миттєве оновлення всіх пов'язаних представлень (наприклад, діаграма класів ↔ діаграма компонентів ↔ діаграма розгортання) без необхідності синхронізувати їх вручну;

– значно швидше сприйняття та пошук потрібного елемента в системі з тисячами класів/сервісів завдяки просторово-візуальним пошукам замість текстової команди «пошук у файлі»;

– візуальні мови програмування можуть включати функції моделювання архітектури як семантичної мережі, де вузли (класи, компоненти) та ребра (зв'язки, залежності) роблять навігацію й аналіз більш інтуїтивними;

– замість традиційної лінійної структури каталогів і файлів, де навігація обмежена ієрархічним списком, пропонується узагальнений граф. Він дозволяє вільно рухатися у двох вимірах, масштабувати (приблизити/віддалити) та оглядати всю систему з висоти, що значно полегшує орієнтацію у складних залежностях і пришвидшує пошук ключових вузлів архітектури.

Внаслідок цього загальний час на типові структурні зміни архітектури (виділення нового мікросервісу, винесення спільної бібліотеки, перехід від моноліту

до модульного моноліту чи розподіленої системи) може скоротитися в декілька разів порівняно з середовищами, що працюють з кодом. Водночас якість архітектури зростає, оскільки розробник частіше проводить рефакторинг і експериментує з різними варіантами структури, бо операції стають практично безболісними та миттєвими. Застосування більшості принципів візуальних мов програмування, викладених у статті «Теоретичні аспекти візуальних мов програмування» [10], робить архітектуру програмного забезпечення гнучкою, дозволяючи легко масштабувати систему від прототипу до об’ємного рішення без потреби налагодження несуттєвого коду. Беручи до уваги теоретичний напрям згаданої статті, архітектурна гнучкість проявляється в автоматичній адаптації залежностей і інтерфейсів при еволюційних змінах, що мінімізує ймовірності того, що зміна (або еволюція) архітектури призведе до появи нових помилок у вже існуючій, раніше коректно працюючій функціональності системи і полегшує інтеграцію з новими технологіями, забезпечуючи довготривалу життєздатність програмного забезпечення у динамічному бізнес-середовищі. Таким чином, інтеграція глибокого модельно-орієнтованого інжинірингу з принципами сучасного візуального програмування дає змогу не лише підвищити швидкість розробки, а й суттєво знизити психологічний бар’єр перед масштабними архітектурними змінами, що в кінцевому підсумку сприяє створенню більш гнучких, підтримувальних та еволюційно стійких програмних систем.

По-третє, утворюється вищий рівень абстракції. Візуальні мови програмування по суті є надбудовою над текстовими мовами програмування, оскільки вони не створюють абсолютно нову мову, а радше використовують існуючі компілятори для лінкування та виконання програм. Це підводить до ключового питання, чи існує краща методологія розробки програмного забезпечення, ніж та, що застосовується зараз? Відповідь полягає в тому, що підвищення рівня абстракції може надати значні переваги, зокрема спрощення розуміння складних структур, зменшення помилок на етапі проєктування та полегшення співпраці між розробниками з різним досвідом. Наприклад, замість роботи з низькорівневим кодом, розробники можуть оперувати візуальними моделями, що абстрагують деталі

реалізації. Однак, як і будь-який підхід, це має потенційні недоліки, такі як надмірна складність для простих систем, де традиційний код може бути ефективнішим. У контексті цієї магістерської роботи фокус робиться саме на покращенні ситуації для великих, складних систем, де вищий рівень абстракції дозволяє ефективніше керувати архітектурою ПЗ, зменшуючи час на розробку та підтримку, а також полегшуючи масштабованість.

По-четверте, очікується покращення методів MDE. Дослідження візуального модельно-орієнтованого інжинірингу в сфері MDE передбачає активний пошук нових ідей, застосування вже існуючих практик та синтез інноваційних підходів. Це включає адаптацію практик з суміжних сфер, створення нових метамоделей для візуалізації, або розробку нових алгоритмів для перевірки цілісності моделей у візуальному середовищі.

По-п'яте, застосування можливостей візуальних мов програмування. Як було описано раніше, VPLs пропонують унікальні можливості, такі як графічне представлення логіки, блоків та потоків даних, що полегшує візуалізацію архітектури ПЗ. У контексті цієї роботи їх застосування полягає в інтеграції з MDE для створення інструментів, де розробники можуть «малювати» моделі замість написання коду, використовуючи «drag-and-drop» інтерфейси, автоматичну генерацію коду та валідацію моделей в режимі реального часу. Це дозволяє використати сильні сторони VPLs, наприклад, зменшення когнітивного навантаження, швидке прототипування та кращу колаборацію в командах. У великих системах це може проявлятися в створенні візуальних діаграм програмної логіки, що автоматично з'єднуються з елементами архітектурних моделей ПЗ, забезпечуючи узгодженість і полегшуючи виявлення помилок на ранніх етапах.

По-шосте, відмова від будь-якого тексту. Цей аспект передбачає максимальну візуалізацію процесів моделювання, де текст використовується лише в необхідних випадках, таких як форми вводу даних, вивід іменування об'єктів, опис артефактів, їх призначення чи частини документації. Усі інші елементи, такі як: логіка, потоки, залежності та архітектура, представляються виключно візуально, через діаграми, іконки, графіки чи інтерактивні елементи. Це гіпотетично підвищує доступність для

нетехнічних спеціалістів, зменшує бар'єри для входу в розробку та мінімізує помилки, пов'язані з синтаксисом тексту. Однак, для великих систем це може вимагати розробки спеціалізованих інструментів, що підтримують повну візуальну екосистему. У підсумку, така відмова від тексту сприяє фокусу на концептуальному рівні, роблячи моделювання більш інтуїтивним і ефективним.

Представлений аналіз демонструє, що поєднання візуального та модельно-орієнтованого підходів не є просто технічною оптимізацією практик розробки програмного забезпечення, а являє собою фундаментальний парадигматичний зсув у способі мислення про проєктування складних програмних систем. Шість виділених переваг, від урахування когнітивних обмежень людського сприйняття до повної візуалізації процесів моделювання, формують цілісну картину трансформації інженерної практики, де центром уваги стає не код як текстовий артефакт, а архітектура як візуально-просторова структура. Ця трансформація відображає глибшу тенденцію в еволюції програмної інженерії, поступовий перехід від машиноцентричних до людиноцентричних підходів, де інструменти адаптуються під природні когнітивні механізми розробника, а не навпаки. Візуальне модельно-орієнтоване середовище стає своєрідним «когнітивним протезом», що розширює обмежені можливості людської робочої пам'яті та уваги, дозволяючи оперувати системами надвисокої складності без втрати контролю над архітектурною цілісністю. Водночас запропонований підхід не є революційним розривом з попередніми практиками, а скоріше еволюційним синтезом десятиліть розвитку програмної інженерії, де кожен новий рівень абстракції будується на фундаменті попередніх досягнень. Візуальні моделі не заміщують код, а надбудовуються над ним, створюючи багаторівневу систему представлень, де кожен рівень оптимізований для вирішення специфічних завдань, від високорівневого архітектурного проєктування до низькорівневої оптимізації продуктивності.

Однак, щоб повною мірою оцінити потенціал та обмеження запропонованого візуального модельно-орієнтованого підходу, необхідно розглянути його в контексті історичної еволюції інструментарію та методологій розробки програмного

забезпечення, тому підрозділ 1.2 присвячений огляду й класифікації підходів та методів розробки.

1.2 Класифікація інструментарію та методів розробки програмного забезпечення: історичний огляд та еволюція

1.2.1 Сучасний стан інструментарію розробки програмного забезпечення

Сучасна розробка характеризується безпрецедентним зростанням складності систем. Операційні системи, хмарні платформи та корпоративні застосунки містять мільйони рядків коду, інтегруються з десятками зовнішніх сервісів і мають задовольняти вимоги високої доступності, безпеки та масштабованості. Це супроводжується переходом від монолітних застосунків до розподілених мікросервісних архітектур, інтеграцією з хмарними сервісами та необхідністю забезпечення безперервного доставлення. Відповідно, інструментарій розробника трансформувався від примітивних текстових редакторів до комплексних екосистем, що охоплюють увесь життєвий цикл програмного продукту.

Історія інструментів розробки починається з епохи безпосереднього програмування в машинних кодах (1940–1950-ті), коли програмісти вводили бінарні інструкції безпосередньо в пам’ять машин. Поява мов асемблера дещо спростила процес, проте він залишався трудомістким. Створення мов високого рівня (FORTRAN, 1957; COBOL, 1959) зумовило потребу в компіляторах та перших текстових редакторах, але розробка залишалася фрагментованою, коли редагування, компіляція та налагодження програм виконувалися окремими утилітами через командний рядок.

Концепція IDE виникла у 1980-х як відповідь на потребу об’єднати розрізнені інструменти в єдиний інтерфейс. Turbo Pascal (Borland, 1983) став одним із перших комерційно успішних середовищ, що поєднував редактор, компілятор та налагоджувач. Подальший розвиток IDE у 1990–2000-х роках приніс синтаксичне підсвічування, автодоповнення (IntelliSense у Visual Studio, 1997), рефакторинг,

інтеграцію з системами контролю версій та статичний аналіз коду. Сучасні IDE (IntelliJ IDEA, Visual Studio Code, Eclipse) – це розширювані платформи з сотнями плагінів, які охоплюють специфічні потреби різних технологічних стеків. З 2020-х років IDE дедалі більше переходять у хмарні середовища (Cloud IDEs) та перетворюються на екосистеми, що підтримують розробника на всіх етапах, від генерації шаблонного коду за допомогою великих мовних моделей (Large Language Models, LLM) [24] до автоматичного розгортання інфраструктури.

Паралельно з розвитком IDE у 1980–1990-х набули поширення CASE-засоби (Computer-Aided Software Engineering), які автоматизували процеси аналізу, проєктування та документування. Хоча розвиваток CASE-засобів не виправдав очікувань щодо повної автоматизації, він заклав фундамент для модельно-орієнтованих підходів. У сучасній практиці інженерії програмного забезпечення, моделі використовуються у двох способах. Модель – це документація, як от UML-діаграми та архітектурні схеми для комунікації, які швидко втрачають актуальність. Модель – це первинний артефакт, як основа, з якої автоматично генерується код, конфігурації та документація. Сьогодні зростання складності систем актуалізує CASE-інструменти нового покоління та модельно-орієнтовану інженерію, що дозволяють абстрагуватися від деталей реалізації та управляти складністю на концептуальному рівні.

Останнім значущим етапом стала інтеграція великих мовних моделей безпосередньо в середовища розробки. Випуск GitHub Copilot (2021) ознаменував появу нового класу інструментів AI-асистентів програмування. Вони здатні генерувати код на основі контексту та коментарів природною мовою, пропонувати автодоповнення на рівні функцій і пояснювати важкі для розуміння елементи в програмному коді. За даними GitHub, використання Copilot підвищує продуктивність розробників на 55% для типових завдань [25]. Водночас ця технологія породжує нові виклики щодо можливості перегляду великої кількості коду, забезпечення якості та безпеки, етики використання, інтелектуальної власності.

Описаний досвід висвітлений в хронологічній таблиці 1.1.

Таблиця 1.1

Хронологія еволюції інструментарію розробки програмного забезпечення

Період	Основний інструмент	Ключова характеристика	Вплив на продуктивність
1940–1950-ті	Машинний код	Пряме програмування апаратури; відсутність абстракцій	Базовий рівень; сотні інструкцій на день
1950–1960-ті	Асемблери	Символьної мнемоніки; залежність від архітектури	Підвищення у 2–5 разів порівняно з машинним кодом
1960–1970-ті	Компілятори мов високого рівня + текстові редактори	Абстрагування від апаратури; роздільні інструменти	Значне підвищення; можливість створення складніших програм
1980–1990-ті	Перші IDE	Інтеграція редактора, компілятора, налагоджувача	Скорочення циклу редагування-компіляції-виконання
1990–2000-ті	IDE з IntelliSense, CASE-засоби	Автодоповнення, рефакторинг, візуальне моделювання	Зменшення рутинних помилок; підтримка великих проєктів
2000–2010-ті	IDE з інтеграцією контролю версій, CI/CD	Git-інтеграція, безперервна інтеграція, плагіни	Командна розробка; автоматизація збірки та тестування
2010–2020-ті	Хмарні IDE, контейнеризація	Розробка у браузері; DevOps-інтеграція	Уніфікація середовищ; швидке розгортання
2020-ті – дотепер	AI-асистенти	Генерація коду на основі LLM; контекстні підказки	За оцінками, +55% для типових завдань [25]

Кожен наступний етап не заміщував попередній повністю, а надбудовувався над ним, формуючи багаторівневий стек інструментів. Сучасний розробник оперує одночасно текстовим кодом, візуальними моделями, декларативними конфігураціями та промптами для AI-асистентів, що актуалізує потребу в уніфікованих підходах до інженерії програмного забезпечення.

Отже, модельно-орієнтована інженерія (MDE) виникла як логічний розвиток CASE-засобів і сьогодні набуває нової актуальності через безпрецедентне зростання складності програмних систем. MDE дозволяє абстрагуватися від деталей реалізації, використовуючи модель як первинний артефакт, з якого автоматично генерується код, конфігурації та документація, що забезпечує ефективне управління складністю на концептуальному рівні. Таким чином, еволюція інструментарію розробки демонструє стійку тенденцію до підвищення рівня абстракції та автоматизації рутинних операцій, від машинних кодів через мови високого рівня до моделей і AI-

асистентів, що підтверджує перспективність модельно-орієнтованих підходів в умовах сучасних розподілених архітектур та хмарних середовищ.

1.2.2 Класифікація мов програмування за поколіннями

Класифікація мов програмування за поколіннями (Generation Languages, GL) є однією з усталених таксономій в історії комп'ютерних наук. Її основна цінність полягає у відображенні ступеня абстрагування від апаратного рівня та зміни парадигми взаємодії програміста з обчислювальною системою. Попри появу альтернативних класифікацій за парадигмами (імперативні, декларативні, функціональні, об'єктно-орієнтовані), поколіннєва таксономія залишається релевантною, оскільки відображає фундаментальну тенденцію розвитку програмування, поступове підвищення рівня абстракції та перенесення когнітивного навантаження з програміста на інструментальні засоби.

Перше покоління (1GL). Машинний код складається з бінарних інструкцій, які процесор виконує безпосередньо. Програмування здійснювалося введенням послідовностей нулів та одиниць, що відповідали конкретним операціям. Це забезпечувало максимальну швидкість виконання, але вимагало від програміста значних когнітивних зусиль. Машинний код повністю залежав від архітектури процесора, тому програми не можна було переносити між різними платформами. Попри складність, він залишається базовим рівнем, до якого зрештою транслюється будь-яка мова програмування.

Друге покоління (2GL). Мови асемблера стали символьним представленням машинного коду, де інструкції записувалися у вигляді мнемонік. Це значно спростило процес написання та пошуку помилок у програмах. Асемблерні програми залишалися залежними від конкретної архітектури процесора, оскільки кожна інструкція відповідала машинним командам. Сьогодні асемблер використовується у драйверах, вбудованих системах та для оптимізації продуктивності.

Третє покоління (3GL). Мови високого рівня, такі як FORTRAN, COBOL та ALGOL, відкрили нову епоху програмування. Вони використовували англомовні ключові слова, що зробило код зрозумілішим для людини. Програми потребували

компіляції або інтерпретації, але забезпечували переносимість між різними платформами. Згодом з'явилися C, Pascal, Java, Python та інші мови, які стали основою сучасної розробки. Їхня популярність пояснюється балансом між простотою написання та ефективністю виконання. Порівняльна характеристика поколінь наведена в табл. 1.2.

Таблиця 1.2

Порівняльна характеристика поколінь мов програмування

Покоління / Рівень абстракції	**Приклади мов**	**Історич-ний початок**	**Сучасний статус**	**Переваги та недоліки**
1GL / Нульова абстракція	Машинний код (x86, ARM)	1940-ті	Цільовий формат компіляції; реверс-інжиніринг.	*Переваги*: найвища швидкість, немає транслятора *Недоліки*: надзвичайно складно читати й модифікувати, непереносимість
2GL / Символьної мнемоніки	NASM, MASM, GAS	1950-ті	Драйвери, вбудовані системи, оптимізація	*Переваги*: легше читати й модифікувати ніж 1GL *Недоліки*: машинно-залежна, потребує асемблера
3GL / Незалежність від апаратури	FORTRAN, C, Java, Python, Go, C++	1957	Домінуючий, покоління для розробки застосунків	*Переваги*: зрозумілі англомовні конструкції, портованість, відносно короткий код *Недоліки*: потрібен компілятор або інтерпретатор
4GL / Орієнтація на домен	SQL, R, MATLAB	1970-ті	Бази даних, бізнес-аналітика, наукові обчислення	*Переваги*: висока продуктивність у предметній області, мінімум коду, низька помилковість *Недоліки*: слабкий контроль над апаратурою, менша гнучкість
5GL / Високий: логіка та обмеження	Prolog, Lisp, OPS5	1972	Нішеве: експертні системи, обробка природної мови, верифікація	*Переваги*: автоматичний пошук рішень, мінімальні зусилля програміста *Недоліки*: складний і довгий код, високе споживання ресурсів
6GL (обговорюється) / Природна мова	Промптинг LLM	2020-ті	Дискусійний статус; експериментальне застосування	*Переваги*: найвищий рівень абстракції, інтуїтивність, низький поріг входу *Недоліки*: код генерується у 3GL/4GL, залишаються їхні обмеження, залежність від LLM

Четверте покоління (4GL). Мови четвертого покоління орієнтовані на опис результату, а не алгоритму його досягнення. Класичним прикладом є SQL, що дозволяє працювати з базами даних у декларативній формі. Вони значно скорочують час розробки та зменшують кількість помилок завдяки високому рівню абстракції.

Недоліком є високе споживання ресурсів та обмежена гнучкість у порівнянні з мовами третього покоління. Такі мови активно застосовуються у бізнес-аналітиці, наукових обчисленнях та системах управління даними.

П'яте покоління (5GL). Мови п'ятого покоління базуються на логіці та концепціях штучного інтелекту. Вони дозволяють системі самостійно знаходити рішення на основі заданих правил та обмежень. Найвідомішими прикладами є Prolog та Lisp, які використовуються для символьних обчислень та експертних систем. Перевагою є автоматизація пошуку рішень і зменшення зусиль програміста. Недоліком є складність коду та значні вимоги до обчислювальних ресурсів, що обмежує їхнє застосування у масовій розробці.

Шосте покоління (6GL, концепція). Шосте покоління розглядається як програмування природною мовою, де інструкції формулюються у вигляді звичайних текстових запитів. Прикладом є використання великих мовних моделей, які інтерпретують промпти та генерують код. Це робить програмування інтуїтивним і доступним навіть для користувачів без технічної освіти. Концепція ще перебуває на стадії формування, але вже демонструє потенціал у зниженні порогу входу. Її розвиток може радикально змінити уявлення про взаємодію людини з комп'ютером.

Під мовами програмування шостого покоління (6GL) у сучасній літературі зазвичай розуміють системи, де основним способом специфікації програми виступає природна мова або високорівневий намір, а процеси генерації, підтримки та еволюції коду делегуються великим мовним моделям чи AI-агентам. З появою LLM ця концепція отримала практичне втілення: сучасні AI-асистенти здатні створювати працездатний код на основі текстових описів. Водночас дискусія щодо того, чи можна промптинг-взаємодію з LLM через текстові запити вважати новим поколінням мов програмування, залишається відкритою. Моделі не виконують програму безпосередньо, а лише генерують код мовами третього та четвертого покоління, тому промптинг радше виступає інтерфейсом до генератора коду, ніж самостійною мовою. Станом на 2025 рік поняття 6GL має переважно маркетинговий або спекулятивний характер, адже академічна спільнота не досягла формального консенсусу щодо виділення нового покоління після 5GL.

Класифікація мов програмування за поколіннями залишається актуальною у 2025 році, оскільки вона наочно демонструє послідовне підвищення рівня абстракції та приховування деталей реалізації. Вона також допомагає зрозуміти співіснування різних рівнів у сучасних системах, де вебзастосунок може бути написаний мовою третього покоління, використовувати четверте для роботи з базою даних, а критичні бібліотеки містити оптимізації на асемблері. Водночас ця таксономія має обмеження, вона є лінійною та спрощеною, тоді як реальний простір мов багатовимірний і включає різні парадигми, типізацію та моделі виконання. Межі між поколіннями залишаються розмитими, а віднесення конкретної мови часто викликає дискусії. Попри це, як евристична модель для розуміння історичної еволюції та загальних тенденцій, класифікація зберігає свою пояснювальну цінність.

1.2.3 Хронологія еволюції методологій розробки програмного забезпечення

Еволюцію методологій та інструментарію розробки програмного забезпечення не можна розглядати як лінійний процес, де кожна наступна технологія повністю витісняє попередню. Натомість це процес накопичення абстракцій та методів, де кожен новий етап пропонує інструменти для розв’язання специфічних проблем, що виникали в попередніх парадигмах. Історичний огляд дозволяє простежити, як інженерна думка рухалася від низькорівневого кодування до маніпулювання візуальними моделями та декларативними описами, намагаючись подолати прірву між людським мисленням та машинним кодом.

Початок систематизації процесів розробки програмного забезпечення пов'язаний із виникненням структурного підходу в 1970-х роках. Ця парадигма стала відповіддю на проблему заплутаного коду – неструктурованих програм із хаотичними переходами управління, що робили код практично неможливим для розуміння та супроводу. Ключова ідея структурного аналізу та проєктування полягала в декомпозиції складних систем на функціональні процеси та потоки даних із використанням чітких контрольних структур. Методології, розроблені Едвардом Йордоном та Томом Демарко, формалізували цей процес через діаграми потоків даних (Data Flow Diagrams, DFD) [26], що дозволяли візуалізувати рух інформації в

системі. Також важливим внеском у розвиток структурного підходу стала методологія SADT (Structured Analysis and Design Technique) [27], розроблена Дугласом Россом наприкінці 1960-х на початку 1970-х років. SADT запропонувала ієрархічну декомпозицію системи через взаємопов'язані діаграми активностей та даних, що згодом стало основою для стандарту IDEF0 [28]. Структурний підхід запровадив дисципліну в процес розробки та створив основу для подальшої автоматизації, проте його лінійність та орієнтація виключно на процеси виявилися недостатніми для моделювання складних систем зі станами та поведінкою об'єктів.

Логічним продовженням структурного підходу стала поява інструментів CASE (Computer-Aided Software Engineering) у 1980-х роках. Ці системи були спрямовані на автоматизацію рутинних задач аналізу та проєктування, зокрема побудови діаграм потоків даних, генерації схем баз даних та підтримки репозиторіїв проєктних артефактів. Часто CASE-інструменти використовувалися у поєднанні з методологією SSADM (Structured Systems Analysis and Design Method) [29], яка виникла як відповідь на проблему відсутності єдиного стандартизованого підходу до розробки великих інформаційних систем у 1980-х, що призводило до непередбачуваності результатів та складнощів у координації команд. SSADM запровадив чітку послідовність етапів життєвого циклу системи та інтегрував три погляди, що доповнюють один одного: функціональний (діаграми потоків даних), структуру даних (логічне моделювання даних) та поведінковий (історії життя сутностей). Це забезпечило передбачуваність процесів розробки, покращило якість документування та полегшило верифікацію й аудит складних інформаційних систем. CASE-засоби обіцяли суттєве прискорення розробки та зменшення кількості помилок завдяки автоматичній генерації коду. Однак на практиці CASE-засоби часто залишалися переважно інструментами документування. Згенерований код виявлявся неефективним, а механізми синхронізації моделі та коду (round-trip engineering) працювали недосконало. Крім того, монолітність цих систем, високі витрати на впровадження та слабка інтеграція між продуктами різних розробників суттєво обмежили їхнє поширення. Залежність від дорогих платних інструментів та

складність їх інтеграції в команди, що вже існують, стали додатковими бар'єрами для широкого промислового впровадження.

Паралельно з розвитком академічних підходів до моделювання, індустрія вимагала скорочення часу виходу продуктів на ринок. Концепція швидкої розробки програмного забезпечення (Rapid Application Development, RAD) [30], популяризована Джеймсом Мартіном на початку 1990-х років, запропонувала принципово інший підхід, швидке ітеративне прототипування як спосіб скорочення циклу зворотного зв'язку з користувачем. Інструменти на кшталт Visual Basic та Delphi впровадили компонентний підхід та візуальні дизайнери інтерфейсів, що стало першим масовим успіхом візуального програмування. Розробник отримав можливість буквально «намалювати» частину програми, що суттєво прискорювало створення користувацьких інтерфейсів. RAD-підхід дозволяв швидше реагувати на зміни вимог користувачів, проте часто призводив до накопичення технічного боргу через пріоритизацію швидкості над архітектурною якістю. Проблеми з масштабованістю та недостатнім документуванням обмежували застосування RAD у великих корпоративних проєктах.

Зростання складності програмних систем вимагало нових підходів до структурування коду, що призвело до розквіту об'єктно-орієнтованого програмування (Object-Oriented Programming, OOP) [31]. Об'єктна парадигма запропонувала об'єднання даних та поведінки в єдині сутності – об'єкти, що покращило модульність та створило можливості для повторного використання коду. У першій половині 1990-х років паралельно існувало кілька нотацій об'єктно-орієнтованого моделювання (ООП), вони відрізнялися синтаксисом діаграм та розставляли різні акценти в процесі моделювання, що ускладнювало взаємодію між командами та обмін проєктними артефактами. У 1997 році автори незалежних методів розробили UML (Unified Modeling Language) – уніфіковану мову моделювання, яка об'єднала елементи попередніх нотацій. UML надав стандартний набір діаграм для опису статичної структури (діаграми класів, компонентів) та динамічної поведінки систем (діаграми послідовностей, станів). Це спростило комунікацію між аналітиками, архітекторами та розробниками. В контексті OOP,

розвинулася ідея, об'єктно-орієнтованого проєктування (Object-Oriented Design, OOD), що сформувалася як дисципліна, що визначає принципи структурування програмних систем на основі об'єктів, їхніх відповідальностей та взаємодій до етапу написання коду. OOD запровадив систематичні методи декомпозиції системи на класи та об'єкти, визначення їхніх інтерфейсів та залежностей, що забезпечило основу для повторного використання компонентів та покращило супровід програмного забезпечення. Водночас проблема розриву між моделлю та кодом залишалася актуальною, після початку імплементації моделі часто не оновлювалися синхронно з кодом і втрачали практичну цінність як актуальна документація.

Паралельно з розвитком професійних методологій, починаючи з 1980-х років, формувався напрямок розробки для кінцевих користувачів (End-User Development, EUD) [32], метою якого було надати експертам предметних областей інструменти для створення програмного забезпечення без глибоких знань програмування. Найяскравішим прикладом EUD стали електронні таблиці, які фактично являють собою форму реактивного програмування з декларативним описом залежностей між комірками. Цей напрямок продемонстрував критичну важливість наближення мови розробки до мови предметної області, що стало фундаментальним принципом для подальшого розвитку MDE-підходів з їх фокусом на доменно-специфічних мовах моделювання (DSL).

Поворотним моментом для модельно-орієнтованого підходу стала ініціатива консорціуму OMG (Object Management Group) у 2001 році, створення MDA (Model-Driven Architecture) [11]. Головна ідея полягала у чіткому розділенні бізнес-логіки, описаної в платформонезалежній моделі (Platform Independent Model, PIM), та деталей реалізації, специфічних для конкретної платформи (Platform Specific Model, PSM), а модель предметної області (Computation Independent Model, CIM) описує систему концептуально. MDA передбачала автоматизовані трансформації між рівнями абстракції, що мало розв’язати проблему залежності попередніх методів від платформ. Це заклало фундамент для ширшої дисципліни MDE, де модель перестала бути просто документацією, а стала первинним артефактом, з якого генеруються інші артефакти розробки, включаючи програмний код. MDE розширила ідеї MDA на

доменні моделі, покращивши можливості абстракції та автоматичної генерації. Проте практичне впровадження цих підходів зіткнулося зі значними труднощами. Складність створення точних метамоделей та коректних трансформацій, а також проблема підтримки синхронізації в обидва боки між моделями та кодом обмежили промислове поширення. Твердження про можливість повної автоматичної генерації коду з моделей виявилися надто оптимістичними, на практиці трансформації часто вимагали значного ручного доопрацювання.

З появою хмарних технологій виникла потреба в управлінні інфраструктурою як програмним забезпеченням. Підхід «інфраструктура як код» (англ. Infrastructure as Code, IaC) [33] використовує декларативні або імперативні описи для розгортання та конфігурування інфраструктурних ресурсів. Інструменти на кшталт Terraform, Pulumi та маніфести Kubernetes дозволяють описувати топологію та конфігурацію системи у вигляді коду, що зберігається під контролем версій, може тестуватися та автоматично застосовуватися. Цей підхід змінив операційну практику, інфраструктура перестала бути ручними налаштуваннями в консолі й стала відтворюваним, виконуваним кодом, що безпосередньо визначає стан ресурсів. IaC розв’язав проблему ручного конфігурування та забезпечив відтворюваність середовищ, водночас ввівши нові ризики, пов’язані з помилками в інфраструктурному коді та необхідністю опанування нових декларативних мов. Архітектура як код (англ. Architecture as Code, AaC), з іншого боку, працює на вищому рівні. Вона передбачає визначення загальної архітектури системи, включаючи компоненти, їх взаємодію та залежності, у кодифікованому форматі. AaC акцентує на дизайні та структурі системи, дозволяючи командам створювати версії, перевіряти та автоматизувати архітектурні рішення. AaC робить архітектуру машинозчитуваною та піддатною автоматичній валідації, дозволяючи перевіряти сумісність компонентів, застосовувати архітектурні політики та автоматизувати перегляд архітектурних змін. Інструменти та підходи AaC включають описові DSL (наприклад, PlantUML, Mermaid, Structurizr), моделі архітектури в коді та механізми для автоматичної генерації діаграм і перевірок, це допомагає уникати розриву між документацією та реальним дизайном системи.

Сучасний етап характеризується об’єднання попередніх ідей у платформах Low-code та No-code, які об’єднують концепції візуального моделювання та End-User Development. Ці платформи дозволяють так званим «цивільним розробникам» (англ. citizen developers) – спеціалістам без глибокої технічної підготовки, створювати корпоративні застосунки переважно візуальними засобами. Платформи малокодової розробки суттєво знижують поріг входження в розробку та прискорюють створення типових бізнес-застосунків, розв’язують проблему дефіциту кваліфікованих кадрів. Водночас вони створюють ризики залежності від конкретного постачальника послуг (англ. vendor lock-in), обмеженої можливості тонкого налаштування та розширюваності, а також потенційних проблем із продуктивністю та масштабованістю для нетипових сценаріїв використання.

Починаючи з 2020-х років, еволюція методологій перейшла у фазу розробки використовуючи потужності штучного інтелекту (AI-driven/augmented development), де великі мовні моделі виступають у ролі інтелектуальних асистентів розробника [34]. Ці системи здатні генерувати код, тести та навіть архітектурні описи на основі специфікацій природною мовою. AI-augmented development започатковує принципово нову еру, де специфікація природною мовою потенційно може стати первинним артефактом розробки. Роль програміста при цьому зміщується від безпосереднього написання коду до його верифікації, керування та інтеграції. Цей підхід розв’язує проблему рутинного кодування типових конструкцій, проте породжує нові виклики [35], пов'язані з точністю та надійністю згенерованого коду, питаннями інтелектуальної власності та етичними аспектами використання штучного інтелекту (ШІ) у критичних системах. Цей етап лише розпочинається, і його довгострокові наслідки для інженерії програмного забезпечення ще потребують систематичного дослідження. Аналіз хронології розвитку методологій дозволяє виявити кілька наскрізних тенденцій.

Кожен новий етап виникав як спроба подолати розрив між концептуальним розумінням системи та її технічною реалізацією. CASE-засоби з'явилися як відповідь на неможливість керувати складністю структурного коду вручну. UML забезпечив стандартизацію архітектурних рішень та покращив комунікацію в

команді. MDE намагався розв'язав проблему синхронізації між моделлю та кодом, перетворивши модель на першоджерело істини. Low-code платформи та AI-асистенти продовжують цю тенденцію, ще більше підвищуючи рівень абстракції.

В табл. 1.3 закріплено погляд на вищенаведені підходи, парадигми, інструменти й методології.

Таблиця 1.3

Хронологія еволюції методологій розробки програмного забезпечення

Період	Метод	Ключова ідея	Основний артефакт	Ступінь автоматизації генерації коду
1970–80-ті	Структурний підхід	Декомпозиція системи на процеси та дані	DFD, SADT, структурні схеми	Низький – скелет коду, схеми БД
1980-ті	CASE-інструменти	Автоматизована підтримка аналізу, дизайну та розробки через інтегровані інструменти	Репозиторії метаданих, діаграми	Середній – часткова генерація коду з моделей, неповний круговий інжиніринг
1990-ті	RAD	Швидкість розробки, візуальне конструювання UI, ітеративність	Прототипи, візуальні форми, компоненти	Високий – для UI та типових операцій
1990-ті	OOP та OOD	Моделювання реального світу через об'єкти	UML-діаграми	Середній – генерація класів, заглушок методів
2000-ті	End-User Development	Усунення посередників-програмістів	Електронні таблиці, макроси, скрипти	Повний – у межах обмеженого домену
2001+	Model Driven Architecture	Розділення логіки (PIM) та платформи (PSM)	Формальні моделі (UML, MOF, Ecore)	Високий – трансформація PIM → PSM → код
Середина 2000-х	Model-Driven Engineering	Розширення MDA на доменні моделі; генерація артефактів (код, документація) з абстрактних представлень	Доменні моделі, метамоделі, інструменти для трансформацій	Високий – алгоритмічна генерація з моделей, інтеграція з CASE для повного циклу
2010-ті	Architecture as Code (AaC)	Архітектура системи описується як код; компоненти, їх взаємодії та залежності кодифікуються	Описові моделі; архітектурні шаблони; діаграми в коді	Середній – автоматична генерація діаграм і валідація архітектурних правил.
2015+	Low-code / No-code	Мінімізація ручного кодування	Візуальні моделі бізнес-процесів, UI-конструктори	Дуже високий – часто інтерпретація моделі в режимі реального часу
2020+	AI-driven Development	Генеративний AI як партнер в розробці (Pair Programming)	Промпти, контекст, підібрані шаблони	Гібридний – генерація, рефакторинг, пояснення

Водночас підвищення рівня абстракції неминуче створювало нові виклики. Автоматизація рутинних задач часто призводила до втрати гнучкості та повного контролю над продуктивністю системи. Перехід від написання коду до

конфігурування моделей лише змістив складність з рівня синтаксису мови програмування на рівень метамоделей та інструментарію. Таким чином, історія розвитку методологій демонструє постійний пошук балансу між зручністю візуального сприйняття, швидкістю розробки та необхідністю точного контролю над архітектурою програмного забезпечення.

1.2.4 Концепція Software 1.0, 2.0, 3.0

Концепцію еволюції програмного забезпечення через парадигми Software 1.0, 2.0 та 3.0 запропонував дослідник у галузі ШІ Андрій Карпаті (Andrej Karpathy) у своїй програмній публікації «Software 2.0» [36]. Карпаті сформулював тезу про фундаментальний зсув в способі створення програмного забезпечення, від явного програмування до навчання на даних. Подальший розвиток великих мовних моделей дозволив розширити цю концепцію до Software 3.0, де природна мова стає інтерфейсом програмування.

Software 1.0 представляє класичний підхід до розробки програмного забезпечення, що домінував протягом усієї історії комп'ютерних наук. У цій парадигмі програміст явно специфікує алгоритми через детерміновані інструкції мовами програмування (Python, C++, Java). Програма є прямим відображенням логіки розробника, де кожен рядок коду має чітке призначення та передбачувану поведінку. Верифікація здійснюється через формальні методи, тестування та перегляду коду.

Software 2.0 радикально змінює парадигму створення програм. Замість явного написання інструкцій, розробник формує архітектуру нейронної мережі та надає навчальні дані. Кодом у цій парадигмі виступають ваги моделі, які оптимізуються автоматично через алгоритми навчання. В результаті це створює принципово новий артефакт, програму, логіка якої є розподіленою та непрозорою. Software 2.0 вимагає нових підходів (MDE4ML) [37] до забезпечення якості та тестування, оскільки поведінка системи стає ймовірнісною, а не детермінованою. Це створює проблему «чорної скриньки», де архітектура системи стає менш зрозумілою для людини, що вимагає нових методів абстрагування, зокрема в контексті MDE.

Software 3.0 виникає з появою великих мовних моделей та їх здатністю до навчання за контекстом. У цій парадигмі природна мова стає інтерфейсом програмування, а промпт – специфікацією поведінки системи. Дослідження [38] показує, що інтеграція таких компонентів у традиційні системи вимагає нових підходів до модельно-орієнтованого інжинірингу (MDE4AI), оскільки традиційні методи моделювання не враховують стохастичну природу та «галюцинації» генеративних моделей.

Співіснування трьох парадигм створює унікальні виклики для сучасної інженерії програмного забезпечення. Сучасні системи поєднують каркас Software 1.0, модулі Software 2.0 та генеративні можливості Software 3.0, що формує складні гібридні системи. У такому середовищі роль інженера змінюється. Він переходить від «кодера» до «архітектора» і «верифікатора», адже штучний інтелект здатний генерувати великі обсяги коду, а ключовим стає контроль архітектури на високому рівні абстракції. Тому виникає потреба у візуалізації, щоб уникнути хаотичних і неструктурованих систем та забезпечити контроль над програмною архітектурою.

1.3 Інтеграційні методи архітектурного проєктування

1.3.1 Модельно-орієнтовані підходи: уточнення термінології

Історичний розвиток інженерії програмного забезпечення призвів до появи широкого спектра акронімів, що описують використання моделей у процесі розробки. Метою цього пункту підрозділу є систематизація термінології та встановлення чітких меж між поняттями. Терміни Model-Driven Development (MDD), Model-Driven Architecture (MDA), Model-Driven Engineering (MDE) та Model-Based Engineering (MBE) описують різні рівні абстракції, ступені автоматизації та сфери застосування [11].

Model-Based Systems Engineering (MBSE) представляє застосування модельно-орієнтованого підходу в системній інженерії. MBSE виходить за межі програмного забезпечення, охоплюючи проєктування комплексних систем

(апаратура, механіка, електроніка, ПЗ). Фокус зміщений на взаємодію апаратного та програмного забезпечення. Основною мовою MBSE є SysML (Systems Modeling Language) – розширення UML для системної інженерії. MBSE широко застосовується у критичних галузях: аерокосмічна промисловість, автомобілебудування, оборонна промисловість, медичного обладнання.

Model-Based Engineering (MBE) представляє найбільш «м'який» підхід до використання моделей у інженерії. У парадигмі MBE моделі виконують допоміжну функцію, вони використовуються для візуалізації архітектури, комунікації між стейкхолдерами, документування рішень та раннього аналізу системи. Принципова відмінність MBE полягає в тому, що моделі не обов'язково є джерелом для автоматичної генерації коду. Розробники можуть створювати код вручну, використовуючи моделі як проєктний план. MBE охоплює широкий спектр практик, від неформальних ескізів на дошці до структурованих UML-діаграм, що не підлягають автоматичній обробці. Такий підхід залишається найпоширенішим у індустрії завдяки низькому бар'єру входження та гнучкості. Рис. 1.1 демонструє відношення всіх термінів між собою.

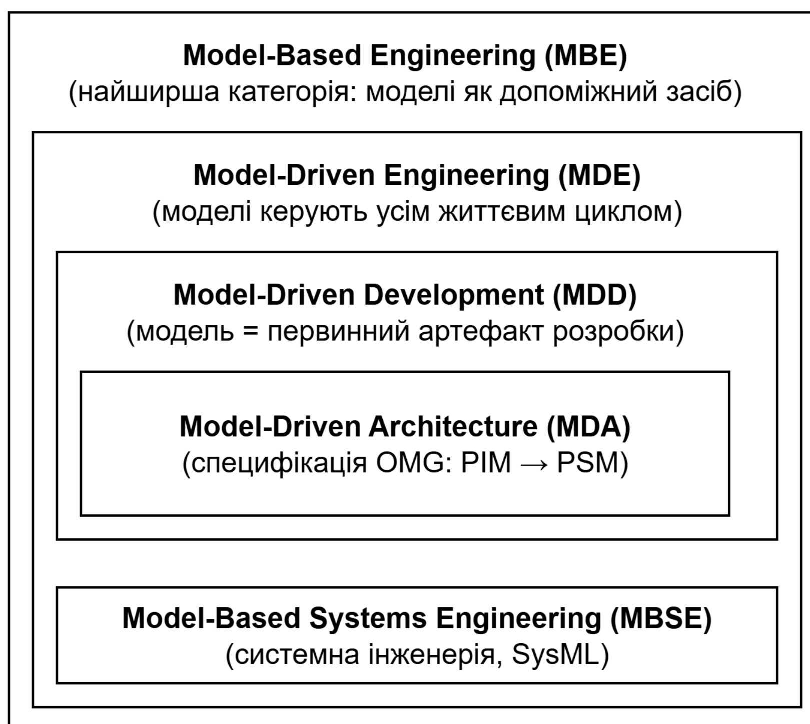

Рис. 1.1. Підмножини підходів, що пов’язані моделями

Model-Driven Development (MDD) – це парадигма розробки програмного забезпечення, де модель є первинним артефактом процесу розробки. На відміну від традиційного підходу, де код є центральним артефактом, у MDD виконуваний код генерується автоматично на основі формально визначених моделей. MDD знайшов широке застосування у розробці вбудованих систем, де генератори коду (такі як Simulink Coder, Embedded Coder) перетворюють візуальні моделі на оптимізований C-код для мікроконтролерів. Ключові характеристики MDD:

- моделі створюються на вищому рівні абстракції, ніж код;
- трансформації моделей є автоматизованими;
- зміни вносяться на рівні моделі, а не коду;
- підтримується концепція «моделі як код».

Model-Driven Architecture (MDA) – це конкретна специфікація, розроблена Object Management Group. MDA є реалізацією принципів MDD з чітко визначеною архітектурою трансформацій. Центральною ідеєю MDA є розділення бізнес-логіки від технологічної платформи через три рівні моделей:

- модель предметної області (CIM), незалежна від обчислювальних аспектів;
- платформо-незалежна модель системи (PIM);
- платформо-специфічна модель (PSM), наприклад Java, .NET тощо.

Трансформація PIM в PSM дозволяє переносити бізнес-логіку між платформами. MDA базується на стеку стандартів OMG: UML (мова моделювання), MOF (Meta Object Facility для метамоделей), XMI (обмін моделями), QVT (для трансформації моделей).

Model-Driven Engineering (MDE) є розширеною концепції MDD, що охоплює не лише розробку, а й весь життєвий цикл програмної системи: аналіз вимог, проєктування, реалізацію, тестування, розгортання та супровід. MDE включає:

- традиційні підходи MDD та MDA;
- предметно-орієнтовані мови (Domain-Specific Languages, DSL);
- Low-code/No-code платформи як сучасну еволюцію підходу;
- генерація моделей за допомогою ШІ (AI-augmented моделювання);
- генерація тестів з моделей (Model-Based Testing, MBT).

Сучасне розуміння MDE 2.0 передбачає інтеграцію з практиками DevOps та застосування ШІ [39] для автоматизації трансформацій моделей. Таблиця 1.4 демонструє характеристику наведених вище модельно-орієнтовані підходів. Далі за текстом наведено маловідомі терміни, що наведені в таблиці.

Таблиця 1.4

Порівняльна характеристика модельно-орієнтованих підходів

Критерій	**MBE**	**MDD**	**MDA**	**MDE**
Рік появи / Ініціатор	~1990-ті / Інженерна спільнота	~2000 / Академічна спільнота	2001 / OMG	2005+ / Розширення MDD
Роль моделі	Допоміжний артефакт	Первинний артефакт розробки	Первинний артефакт з формальними трансформаціями	Центральний артефакт усього життєвого циклу
Ступінь автоматизації	Низький (ручна інтерпретація)	Високий (генерація коду)	Високий (PIM→PSM трансформації)	Варіативний (від часткової до повної)
Основний фокус	Візуалізація, аналіз, комунікація	Генерація виконуваного коду	Незалежність від платформи, портабельність	Весь життєвий цикл: розробка, тестування, супровід
Стандарти	Немає обов'язкових	UML, DSL	UML, MOF, XMI, QVT	UML, DSL, SysML, MOF, low-code платформи
Сучасне використання	Широке (ескізи, прототипи)	Вбудовані системи, кодогенератори	Legacy-системи, корпоративні рішення	DevOps, AI-augmented моделювання, Digital Twins

В таблиці згадуються наступні терміни. DSL (Domain-Specific Language) – спеціалізована мова програмування для розв'язання задач у конкретній предметній області, що спрощує написання зрозумілого коду. Digital Twins – цифрові моделі реальних систем для їх аналізу та оптимізації у віртуальному середовищі; в інженерії ПЗ застосовуються для симуляції життєвого циклу програм. Meta Object Facility (MOF) [40] є стандартом OMG для визначення метамоделей. У стандарті MOF виділяють чотири рівні абстракції: M0 – реальні об'єкти та дані; M1 – моделі, що їх описують (наприклад, UML-діаграми); M2 – метамоделі, що задають правила побудови моделей; M3 – мета-метамодель MOF, що встановлює основу для метамоделей. XMI – стандарт OMG для обміну метаданими на основі XML, що

забезпечує серіалізацію моделей та інтероперабельність інструментів моделювання. QVT – стандарт OMG для опису запитів та трансформацій моделей, що базується на MOF і забезпечує автоматизацію перетворень в MDE. Також важливими є Object Constraint Language (OCL) для формалізації правил у UML-моделях та Eclipse Modeling Framework (EMF) для створення моделей даних з автоматичною генерацією Java-коду.

Концептуальною основою MDE слугує чотирирівнева архітектура метамоделювання з MOF на вершині (рівень M3). У поєднанні зі стандартами QVT, OCL та реалізаціями на кшталт EMF ця архітектура робить MDE гнучким підходом для адаптації до різноманітних технологічних контекстів.

1.3.2 Візуальні мови програмування та мови моделювання

У контексті еволюції методів розробки програмного забезпечення особливе місце посідають візуальні мови програмування (Visual Programming Languages, VPLs), які пропонують альтернативу текстовому кодуванню. Візуальна мова програмування (VPL) визначається як формальна мова, що дозволяє створювати програми шляхом маніпуляції графічними елементами (блоками, іконками, діаграмами) замість написання текстових інструкцій. Основна ідея VPLs полягає у використанні просторового розташування візуальних об'єктів для визначення семантики програми, що дозволяє розробникам фокусуватися на логіці та потоці даних, мінімізуючи синтаксичні помилки, притаманні текстовим мовам. Візуальні мови програмування за своєю суттю не створюють нових обчислювальних стандартів, а виступають візуальною абстракцією над текстовим кодом, що автоматично трансформує графічні примітиви у конструкції текстових мов програмування. Водночас вони відіграють ключову роль у формуванні новітньої парадигми розробки, докорінно змінюючи взаємодію інженера з кодом через перенесення уваги з синтаксичних деталей на семантику та архітектурну структуру рішення. Важливо розмежовувати поняття візуальних мов програмування та візуальних мов моделювання (Visual Modeling Languages, VML), таких як UML або ArchiMate. Хоча обидва підходи використовують графічну нотацію, їхні цілі

відрізняються. VML фокусуються на абстрактному описі структури та поведінки системи, проєктуванні. Моделі часто потребують подальшої ручної імплементації або складної трансформації в код.

Дослідження [41] показує, що VPLs вже давно вийшли за межі освітніх цілей (як Scratch) і активно застосовуються у професійних сферах. У світі Інтернет речей (Internet of Things , IoT), наприклад Node-RED широко використовується для швидкого прототипування та інтеграції пристроїв. У ігровій індустрії широко використовуються візуальні скриптові системи (Unreal Blueprints, Unity Visual Scripting) для створення ігрової логіки. А в наукових та інженерних обчисленнях LabVIEW та Simulink забезпечують моделювання складних систем і обробку сигналів. Аналіз сучасних інструментальних засобів дозволяє класифікувати VPL за домінуючою парадигмою візуалізації. Найпоширенішими є блочні, потокові та діаграмні підходи. У табл. 1.5 наведено порівняльну характеристику цих типів VPLs із зазначенням їхніх переваг та обмежень.

Таблиця 1.5

Типи візуальних мов програмування та їх характеристики

Тип VPL	Парадигма та приклади	Переваги	Обмеження
Блокові (Block-based)	*Парадигма*: Імперативна, структурована. *Приклади*: Scratch, Google Blockly, MIT App Inventor.	• Повне усунення синтаксичних помилок (syntax-free) • Низький поріг входу для новачків • Інтуїтивне розуміння вкладеності (scope)	• Низька щільність інформації (verbose) • Складність редагування великих алгоритмів • Повільна швидкість розробки для досвідчених програмістів
Потокові (Node-based / Dataflow)	*Парадигма*: Реактивна, потокова. *Приклади*: Node-RED (IoT), Unreal Blueprints (GameDev), LabVIEW, Dynamo (Building Information Modeling).	• Інтуїтивна візуалізація потоку даних • Природне представлення паралелізму • Легке відстеження залежностей	• Ризик виникнення заплутаності через масштаб візуалізації • Обмежені можливості рефакторингу • Труднощі з циклами та рекурсією
Діаграмні (Diagram-based)	*Парадигма*: Автоматна, поведінкова. *Приклади*: Yakindu (Statecharts), MATLAB Simulink, BPMN	• Високий рівень абстракції (відповідає предметній області) • Пряма верифікація логіки	• Вимагає спеціальних знань домену • Залежність від інструментів

Блочні мови використовують метафору пазлів, де програмні конструкції (цикли, умови) представлені як блоки, що можуть з'єднуватися лише синтаксично правильним способом. Це повністю усуває проблему синтаксичних помилок.

Потокові мови базуються на графі, де вузли виконують операції, а ребра передають дані між ними.

Діаграмні мови використовують існуючі нотації, такі як діаграми станів або блок-схеми, перетворюючи їх на виконувані специфікації.

Основними перевагами VPLs є зниження порогу входу в програмування, наочна візуалізація потоків даних і керування, мінімізація синтаксичних помилок, прискорене прототипування та полегшення співпраці між технічними й нетехнічними учасниками. Водночас існують суттєві обмеження: проблеми масштабованості у великих проєктах, обмежена виразність порівняно з текстовими мовами, а також складність версійного контролю. У модельно-орієнтованому інжинірингу ключовою перевагою VPL є зменшення когнітивного навантаження на розробника, оскільки візуалізація сприяє швидкій ідентифікації архітектурних компонентів і зв’язків, що особливо важливо для складних систем. Проте головним чинником, який обмежує масштабне застосування VPL, залишається проблема підтримуваності та читабельності коду, коли надмірна кількість візуальних зв’язків ускладнює розуміння та рефакторинг логіки. Подальше вдосконалення інструментів і методологій дозволить поступово усунути обмеження та зробити цей підхід ефективним навіть для масштабних і складних проєктів.

1.4 Інноваційні технологічні методи проєктування

1.4.1 Предметно-орієнтоване проєктування: концептуальні засади

Предметно-орієнтоване проєктування (Domain-Driven Design, DDD) – це методологія проєктування програмного забезпечення, яка фокусується на створенні програмних моделей, що точно відображають предметну область та бізнес-логіку системи [42]. В основі DDD лежить принцип, що структура та мова програмного

коду повинні відповідати бізнес-домену через тісну співпрацю між технічними та доменними експертами для створення спільної мови – Ubiquitous Language. Поява DDD (Ерік Еванс, 2003) заклала концептуальну основу для сучасних візуально орієнтованих підходів, систематизувавши принципи створення моделей через єдину мову спілкування.

Концептуально DDD поділяється на стратегічне та тактичне проєктування. Стратегічне проєктування зосереджується на високорівневих рішеннях щодо декомпозиції великих систем. Ключовим поняттям є обмежений контекст (Bounded Context) – явна межа, всередині якої модель має суворе значення. Піддомени класифікують частини системи: ядро (Core Domain) представляє найважливішу частину бізнесу, допоміжний піддомен підтримує основний, а загальний піддомен охоплює типові задачі. Поєднання контекстів (Context Mapping) візуалізує відносини між контекстами через типи зв'язків: Partnership, Shared Kernel, Customer-Supplier, Conformist, Anticorruption Layer. Інструменти, такі як Context Mapper [43], доводять, що формалізація цих зв'язків дозволяє генерувати специфікацію API та валідувати інтеграційну архітектуру ще до написання коду [44].

Тактичне проєктування надає патерни для реалізації моделі: Entity (об'єкт з унікальною ідентичністю), Value Object (об'єкт без ідентичності), Aggregate (кластер пов'язаних об'єктів з Aggregate Root), Domain Service, Domain Event, Repository та Factory. У візуальних моделях Aggregate представляється як композитні елементи з чітко визначеними інтерфейсами.

Зростання популярності DDD з розвитком мікросервісної архітектури створило потребу у візуальних інструментах. Bounded Context природно застосовується на мікросервісах, а Context Maps стали стандартом для документування мікросервісних ландшафтів. У контексті візуального MDE, DDD виступає як семантичний фундамент: DDD надає семантику (що моделюємо?), а MDE – інструментарій (як трансформуємо в код?). Як зазначають автори дослідження [44], використання DDD-специфічних мов (завдяки DSL) дозволяє «швидко прототипувати» архітектуру, перетворюючи моделі контекстів та агрегатів у шаблонний код сервісів.

Це підтверджується дослідженнями [45], які демонструють методи автоматичного виведення кінцевих точок API безпосередньо з доменних моделей. Підтримка швидкого прототипування включає ланцюжок трансформацій від User Stories до автоматичного виведення піддоменів, групування у Bounded Contexts і генерації коду через JHipster.

DDD приносить значні переваги при роботі зі складними доменами: глибоке розуміння бізнес-логіки, чітке розмежування контекстів для паралельної розробки, покращена комунікація через всюдисущу мову. Водночас методологія вимагає значних початкових інвестицій та досвідченої команди. Поєднання семантики DDD з методами візуального MDE дозволяє автоматизувати створення архітектури з вимог, роблячи процес інтерактивним та візуальним. Таким чином, DDD стає необхідним інструментом для виділення правильних абстракцій в основу програмних засобів архітектурного моделювання.

1.4.2 Low-code та No-code платформи розробки

У контексті еволюції методів розробки програмного забезпечення особливе місце посідають платформи Low-Code (LCDP) та No-Code (NCDP) – сучасна промислова адаптація принципів модельно-орієнтованого інжинірингу, де моделі стають безпосереднім засобом створення кінцевого продукту.

Платформи малокодової розробки забезпечують швидке створення застосунків з використанням декларативних методів високого рівня абстракції: візуального моделювання та drag-and-drop інтерфейсів. Платформи No-Code позиціонуються як підклас LCDP, що повністю виключає написання коду, орієнтуючись на цивільних-розробників [46]. Low-Code платформи допускають вставки ручного коду для розширення функціоналу. Ключова відмінність полягає в цільовій аудиторії: No-Code пропонують закриті системи з жорсткими шаблонами, тоді як Low-Code надають відкриті API та можливість інтеграції власного коду.

Якщо класичні MDE-інструменти критикувалися за складність, то сучасні Low-Code платформи приховують складність метамоделей за інтуїтивними інтерфейсами. Вони використовують візуальні DSL для опису бізнес-процесів,

структур даних та UI з автоматичною генерацією коду. Однак ці платформи часто страждають від прив'язки до постачальника та недостатньої підтримки складних архітектурних патернів.

Ключові характеристики Low-Code/No-Code платформ включають три аспекти. По-перше, у статті [17] наголошується, що в основі Low-Code платформ лежать принципи модельно-орієнтованої інженерії з використанням моделювання та метамоделювання для автоматизації та абстрагування. По-друге, платформи надають графічне середовище з декларативними візуальними моделями замість синтаксису мов програмування. По-третє, автоматична генерація коду та хмарна інфраструктура забезпечують трансформацію моделей у готові до розгортання застосунки.

Low-Code та No-Code платформи є наймасовішим прикладом успішної імплементації MDE, емпірично доводячи, що візуальне моделювання здатне знизити когнітивне навантаження та пришвидшити доставку продуктів. Однак, проведений аналіз виявив суттєву прогалину в існуючих інструментах [47], комерційні LCDP здебільшого фокусуються на реалізації типових бізнес-процесів та UI (рівень реалізації), часто нехтуючи глибоким архітектурним моделюванням складних доменів (рівень проєктування).

1.4.3 AI-driven розробка: еволюція парного програмування

Традиційна концепція парного програмування в методології Extreme Programming передбачала співпрацю двох інженерів: «водія», який пише код, та «навігатора», який аналізує стратегію. З появою великих мовних моделей ця парадигма трансформувалася в AI-augmented pair programming, де роль «водія» перебирає штучний інтелект. Згідно з дослідженням [48], відбувається фундаментальний зсув до симбіотичного партнерства, де AI бере на себе рутинну генерацію коду, дозволяючи інженеру зосередитися на розв'язанні складних проблем, що фактично є першим кроком до «неявного» модельно-орієнтованого підходу. Дослідження підтверджує, що в такій синергії людина зміщує фокус з написання синтаксичних конструкцій на валідацію логіки та архітектурне керування. Логічним розвитком цієї тенденції став феномен «vibe coding» (2025) – підхід, за

якого розробник керує процесом через високорівневі промпти, довіряючи деталі реалізації нейромережі. Це стихійна форма MDE, де роль моделі виконує природна мова, а трансформацію в код здійснює LLM. Однак, на відміну від формального MDE, «vibe coding» позбавлений детермінованості та суворої метамоделі, що створює ризики для підтримки великих систем. Для індустріального застосування необхідний перехід до MDE 2.0, де генеративні можливості AI поєднуються з візуальними або текстовими DSL, забезпечуючи швидкість «vibe coding» та надійність архітектурних рішень.

Вплив ШІ на інженерію керовану моделями охоплює кілька напрямків, що фундаментально змінюють традиційний MDE-конвеєр. По-перше, автоматизація створення моделей: LLM аналізують текстові специфікації та генерують UML-діаграми, метамоделі або DSL-конструкції, скорочуючи час моделювання з днів до годин, хоча потребують валідації для усунення семантичних неоднозначностей. По-друге, оптимізація трансформацій: AI навчається на прикладах відображень «модель-код» і генерує правила трансформацій, що раніше вимагало глибокої експертизи. По-третє, автоматична валідація з прогнозуванням архітектурних антипатернів через аналіз семантики моделі в контексті предметної області. По-четверте, машинне навчання на історичних даних виявляє кореляції між характеристиками моделей та дефектами, створюючи систему раннього попередження. Критичним обмеженням залишається непрозорість LLM, що ускладнює інтеграцію в сертифіковані процеси з вимогами повної простежуваності. Перспективним напрямком стає розробка гібридних систем, де формальні метамоделі слугують вказівним знаком для AI-генерації, забезпечуючи баланс між гнучкістю генеративних моделей та детермінованістю традиційного MDE.

Отже, еволюція від класичного парного програмування до «vibe coding» демонструє незворотний перехід до AI-оркестрування, де критичним викликом стає втрата контролю над архітектурною цілісністю. Це обґрунтовує необхідність впровадження гібридних методів візуального модельно-орієнтованого інжинірингу (MDE 2.0), здатних забезпечити абстракцію для верифікації згенерованих рішень та гарантувати їх відповідність проєктним вимогам.

1.5 Висновки до розділу

У першому розділі здійснено комплексне дослідження предметної області та суміжних напрямків, що включає аналітичний огляд сучасного стану інструментарію розробки програмного забезпечення. Проведений аналіз еволюції методологій розробки від структурного підходу 1970-х років через CASE-засоби, RAD, об'єктно-орієнтоване проєктування до сучасних Low-code/No-code платформ та AI-driven розробки, – продемонстрував стійку тенденцію до підвищення рівня абстракції та автоматизації рутинних операцій. Класифікація мов програмування за поколіннями (1GL–6GL) підтвердила закономірність поступового переходу від машиноцентричних до людиноцентричних підходів.

Систематизовано термінологію модельно-орієнтованих підходів (MBE, MDE, MDD, MDA, MBSE), встановлено їх ієрархічні відношення, від найширшої концепції MBE до конкретної специфікації MDA. Охарактеризовано візуальні мови програмування за типами (блокові, потокові, діаграмні) та обґрунтовано їх потенціал для зниження когнітивного навантаження розробника.

Виявлено критичну прогалину сучасного інструментарію: розглянуті засоби або фокусуються на високорівневому архітектурному моделюванні без ефективної генерації коду, або забезпечують швидке прототипування без належного контролю архітектурної якості. Сформульовано шість гіпотетичних переваг інтеграційного підходу, що поєднує MDE та VPL з урахуванням когнітивно-ергономічного дизайну. Отримані результати обґрунтовують доцільність розробки методології візуального модельно-орієнтованого інжинірингу для архітектурного моделювання.

В наступному розділі здійснено порівняльний аналіз 26 інструментів за 12 категоріями з визначенням їх сильних і слабких сторін за критеріями наявності метамоделі, можливостей генерації коду, двонаправленої синхронізації, когнітивної складності та типу візуалізації. Також розглянуто сучасний стан та ключові тенденції архітектурного моделювання програмного забезпечення.

РОЗДІЛ 2
ОГЛЯД ВІЗУАЛЬНИХ МЕТОДІВ МОДЕЛЬНО-ОРІЄНТОВАНОГО ІНЖИНІРИНГУ ТА ПОСТАНОВКА ЗАДАЧІ

2.1 Огляд візуальних інструментів та методів для архітектурного моделювання програмного забезпечення

Модель C4 [18], розроблена Саймоном Брауном, представляє собою ієрархічний підхід до візуалізації архітектури програмного забезпечення через чотири послідовних рівні абстракції, що поступово деталізують систему від загального контексту до рівня коду, див. рис. 2.1.

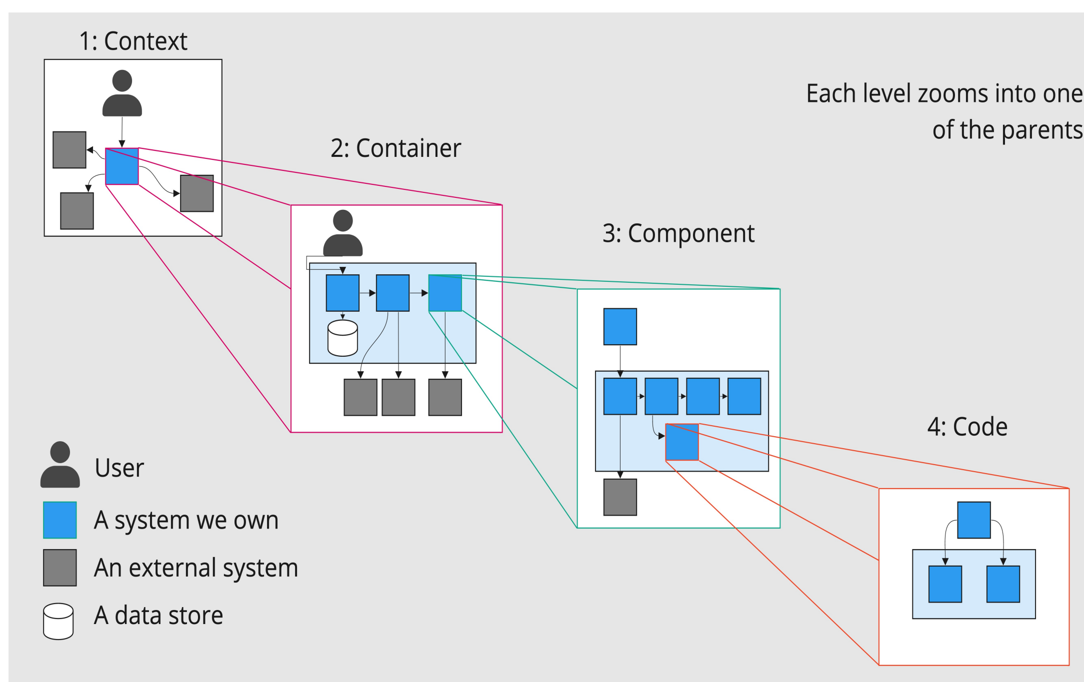

Рис. 2.1. Чотири рівні модель C4

На найвищому рівні Context відображається взаємодія системи із зовнішніми акторами та суміжними системами, далі рівень Containers розкладає систему на великі розгортувані одиниці, такі як вебзастосунки, бази даних чи мікросервіси, потім рівень Components деталізує внутрішню структуру контейнерів, і нарешті рівень Code показує реалізацію на рівні класів. Модель C4 позиціоную себе, як

незалежну щодо нотацій та інструментів моделювання, тобто не обмежується єдиним середовищем. Крім того, C4 фокусується переважно на статичній структурі системи. Для повноцінного архітектурного опису, поведінкових аспектів та динаміки взаємодії компонентів, вимагає доповнення іншими нотаціями.

Далі було розглянуто, Модель 4+1 [19]. Запропонована Філіпом Крухтеном та стандартизована IEEE, пропонує описувати архітектуру системи через п'ять взаємопов'язаних представлень, кожне з яких адресує потреби різних категорій стейкхолдерів, див. рис. 2.2.

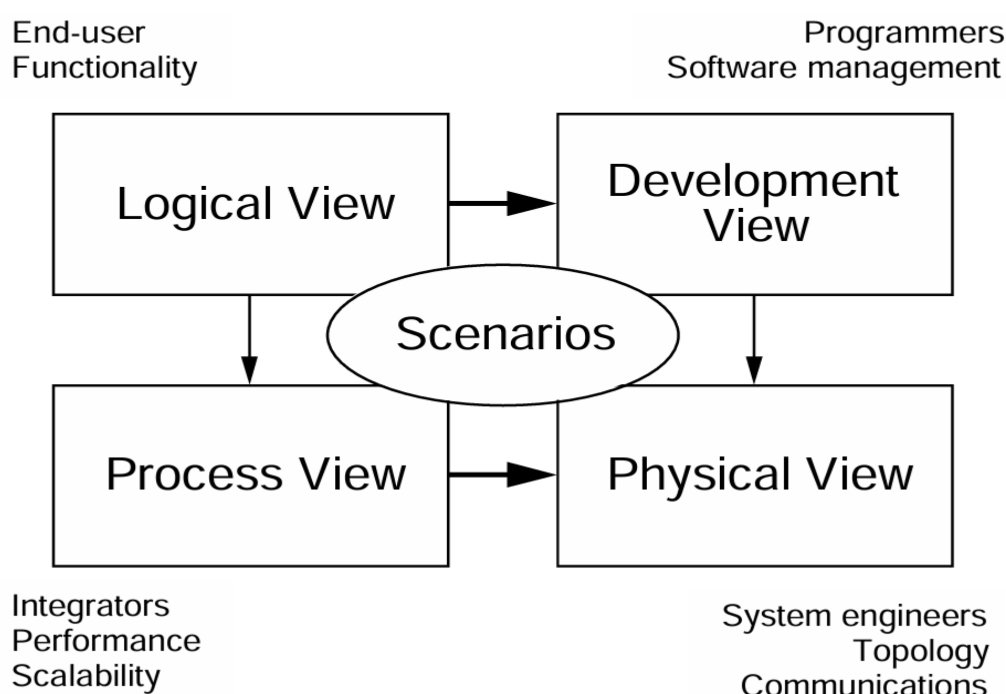

Рис. 2.2. Схема моделі 4+1

Логічне представлення розкриває функціональну структуру системи через класи та їх взаємозв'язки, процесне представлення описує динаміку виконання та паралелізм, фізичне представлення демонструє топологію розгортання на апаратному забезпеченні, розробницьке представлення відображає організацію програмних модулів, а сценарії використання об'єднують усі погляди та верифікують їх узгодженість. Застосовувалася в промислових проєктах (авіаційний контроль, телеком). Незважаючи на концептуальну елегантність та широке визнання в академічному середовищі, модель 4+1 страждає від проблеми підтримки

консистентності між п'ятьма представленнями, оскільки зміни в одному погляді мають бути вручну синхронізовані з іншими. Відсутність вбудованих механізмів автоматичної верифікації та трасування залежностей між представленнями призводить до поступової деградації архітектурної документації, що робить цей підхід когнітивно затратним для практичного застосування у великих проєктах.

Unified Modeling Language залишається найпоширенішою стандартизованою мовою для візуального моделювання [15], надаючи 14 типів діаграм, розділених на структурні, зокрема Class, Component, Deployment та Package, і поведінкові, включаючи Sequence, Activity та State Machine, тоді як SysML розширює UML для моделювання складних систем через додаткові діаграми вимог та параметричні діаграми. Для архітектурного моделювання програмного забезпечення особливо важливі Component діаграми для опису модульної структури (рис. 2.3) та Deployment діаграми для представлення фізичної топології системи.

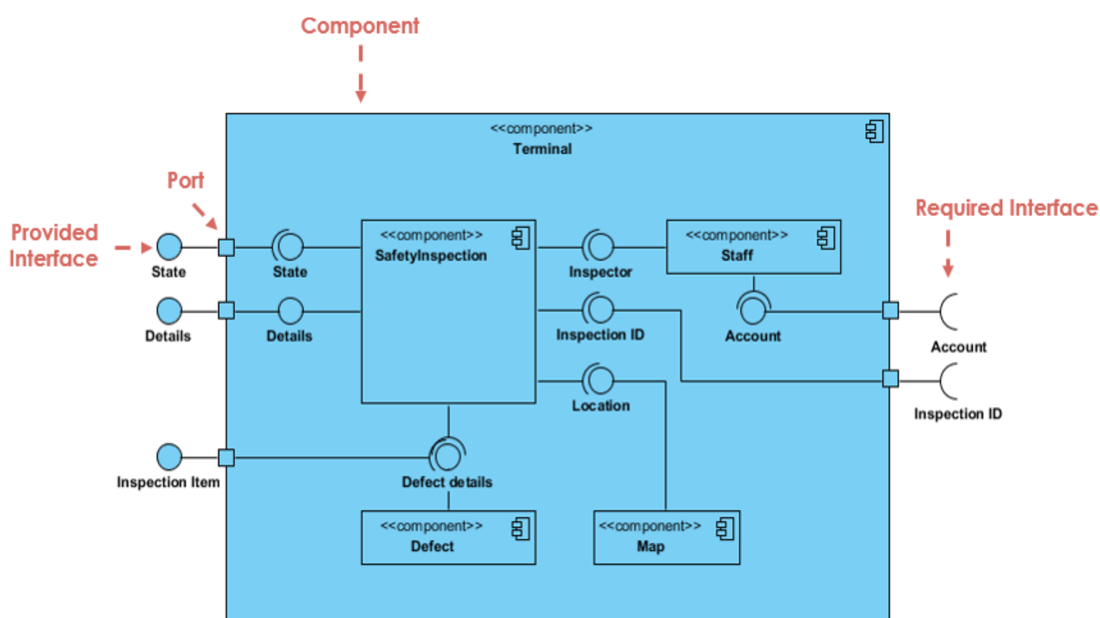

Рис. 2.3. Компонентна діаграма нотації UML

Проте, незважаючи на потужність та універсальність, UML характеризується надмірною складністю, яка створює значне когнітивне навантаження на розробників, оскільки повноцінне володіння всіма 14 типами діаграм та їх семантикою вимагає тривалого навчання. Критичним недоліком є також розрив між моделями та виконуваним кодом, адже автоматична генерація

повнофункціонального коду з UML-моделей залишається не розв’язаною задачею, що перетворює моделювання на документаційну активність, відірвану від реального процесу розробки.

ArchiMate [12], розроблений The Open Group як відкритий стандарт для моделювання корпоративної архітектури, пропонує багатошарову структуру, що охоплює стратегічний, бізнесовий, прикладний та технологічний рівні організації, забезпечуючи цілісне бачення від бізнес-цілей до технічної інфраструктури, див. рис. 2.4. Мова містить близько 60 типів елементів та зв'язків, організованих у логічні шари, що дозволяє моделювати складні залежності між бізнес-процесами, застосунками та технологічними компонентами. ArchiMate має формальну метамодель, що робить його придатним для інтеграції з MDE-інструментами, зокрема через підтримку в таких середовищах як Archi та BiZZdesign. Водночас суттєвим обмеженням є орієнтація стандарту на рівень Enterprise Architecture, що означає фокус на високорівневому стратегічному плануванні, а не на деталях програмної реалізації. Відсутність безпосереднього зв'язку з кодом та неможливість генерації виконуваних артефактів робить ArchiMate інструментом для архітекторів-аналітиків, але не для розробників, що потребують наскрізної відстежуваності від архітектурних рішень до їх імплементації.

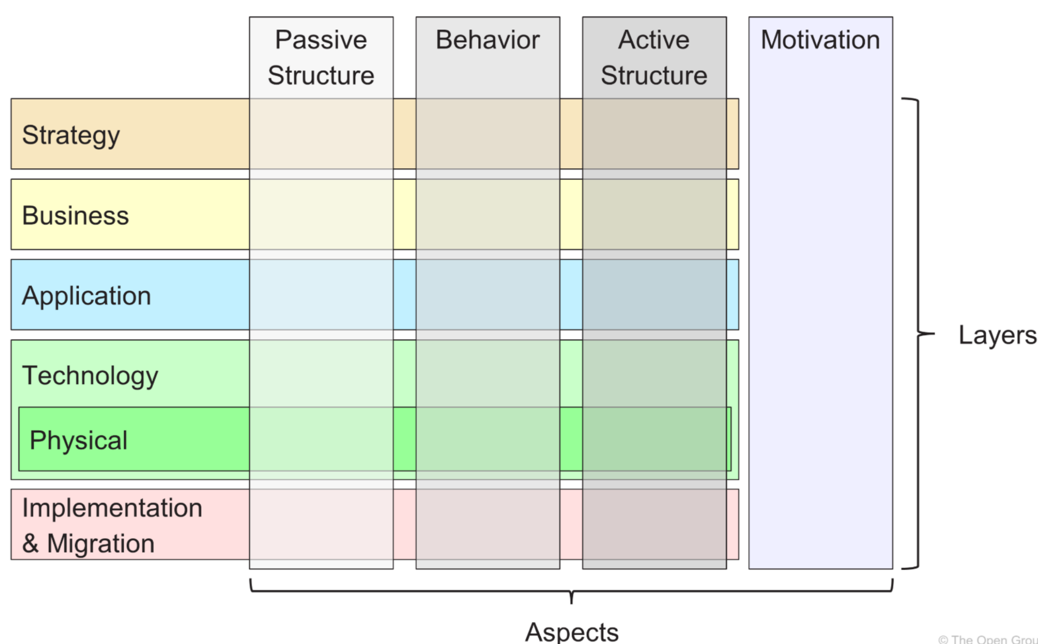

Рис. 2.4. Повношарова ArchiMate діаграма

Варто також згадати, AADL , стандартизований SAE International (компанія), є спеціалізованою мовою для моделювання архітектури вбудованих систем реального часу та критичних застосунків, що забезпечує формальну семантику для аналізу продуктивності, надійності та безпеки, проте його вузька доменна спрямованість обмежує застосовність у загальному контексті програмної інженерії.

В інженерії програмного забезпечення частково застосовують модель TOGAF Content Metamodel, яка у поєднанні з ArchiMate формує комплексний фреймворк для Enterprise Architecture, де фаза розробки «C» методології TOGAF безпосередньо адресує архітектуру інформаційних систем, однак цей підхід залишається на рівні стратегічного планування без прямого зв'язку з програмною реалізацією.

JetBrains MPS реалізує радикально інший підхід до мовного інжинірингу через концепцію проєкційного редагування, де користувач маніпулює безпосередньо абстрактним синтаксичним деревом (AST) замість текстового представлення, що дозволяє створювати мови з довільними нотаціями, включаючи таблиці, математичні формули та діаграми в межах єдиного редактора (рис. 2.5).

```
DecisionSample

    Money discount;
    discount = create(createPerson());

    if (discount > 400 USD || discount >= 350 EUR) {
      discount = 300 EUR;
    }

    System.out.println("Your name: " + createPerson());
    System.out.println("Your discount: " + discount);

  }

  public Money create(map<string, Object> person) {
    return Money Default: 0 EUR
                      | isLevel_1(person)            | isLevel_2(person)                          | ;
      isChild(person) | 500 EUR                      | 1000 EUR                                   |
      isAdult(person) | 50 EUR + this.seasonalBonus() | 100 EUR + this.seasonalBonus()             |
    isRetired(person) | 200 EUR                      | 250 EUR + (person["name"] == "Susan" ?     |
                      |                              |     this.seasonalBonus() : 0 EUR)          |
  }

  private Money seasonalBonus() {
    return 100 EUR;
  }
```

Рис. 2.5. Приклад предметно-орієнтовані мови MPS-редактора

Ключовою перевагою MPS є можливість композиції мов, коли різні DSL сумісно інтегруються в одному проєкті без конфліктів граматик, що принципово

неможливо у традиційних парсерних підходах. Платформа успішно застосовується у таких проєктах як mbeddr для вбудованих систем та KernelF для фінансової бізнес-логіки. Однак проєкційний підхід створює значний когнітивний бар'єр для розробників, звичних до текстових редакторів, оскільки взаємодія з AST вимагає перебудови взаємодії з кодом. Крім того, створення нових мов потребує глибокого розуміння внутрішньої архітектури платформи, що обмежує доступність для широкого кола користувачів.

MetaEdit+ від компанії MetaCase є комерційним DSL-середовищем, що базується на метаметамоделі GOPPRR, яка описує домен через концепції Graph, Object, Port, Property, Relationship та Role, забезпечуючи швидке прототипування візуальних мов моделювання (рис. 2.6).

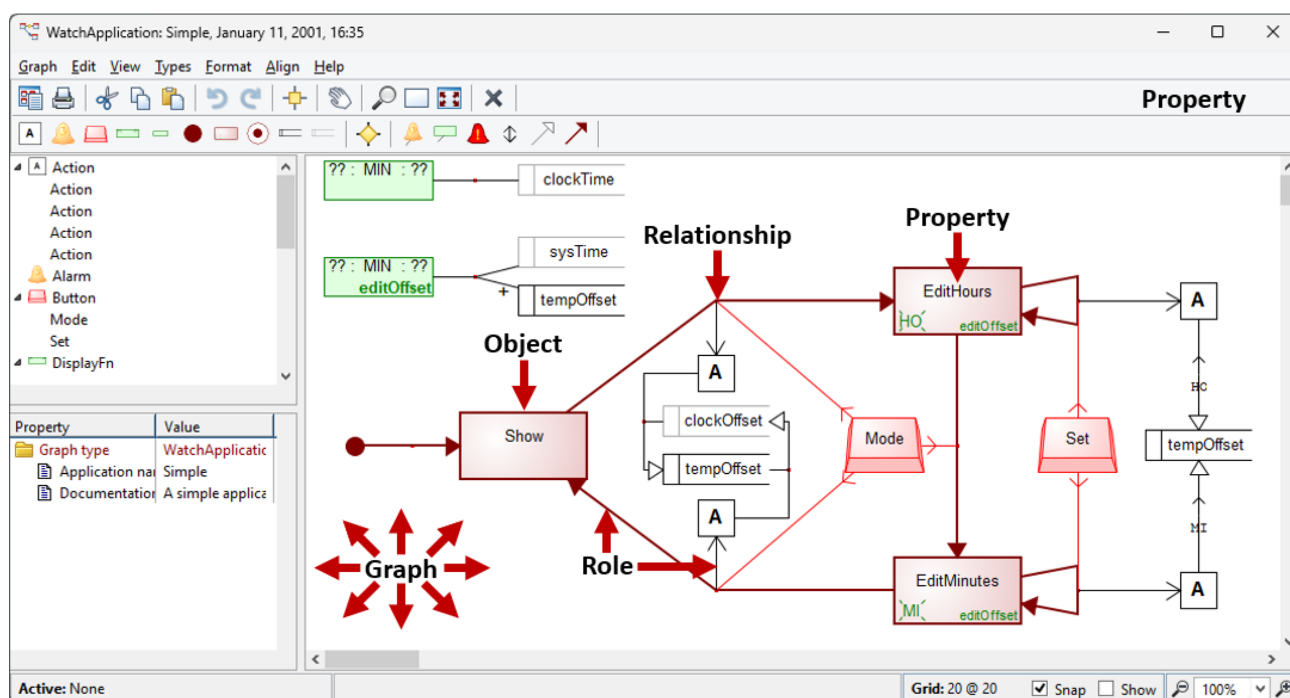

Рис. 2.6. Схема GOPPRR-метаметамоделі

MetaEdit+ Workbench, включаєє інструменти для проєктування та використання мов моделювання. MetaEdit+ Modeler, включає інструменти для використання мов моделювання. Платформа знайшла застосування в автомобільній промисловості та телекомунікаціях завдяки інтуїтивному графічному редактору метамоделей. Проте закритість платформи та комерційна модель ліцензування обмежують доступність для академічних досліджень. Інструмент сильно

орієнтований на структуру конкретної DSL. При зміні вимог до проєкту можливо знадобиться суттєва модифікація метамоделі.

Eclipse Modeling Framework (EMF) разом із метамодельною мовою Ecore становить фундамент модельно-орієнтованої екосистеми у світі Java, забезпечуючи стандартизовану реалізацію специфікації MOF для визначення структурованих метамоделей та автоматичну генерацію Java-класів із підтримкою нотифікацій, рефлексії та серіалізації у форматі XMI. EMF надає потужний механізм для опису доменних моделей через декларативні Ecore-схеми, які трансформуються у повнофункціональний код моделі та базові деревоподібні редактори. Проте суттєвим обмеженням є орієнтація переважно на технічних спеціалістів із глибокими знаннями Eclipse-платформи, оскільки створення навіть простої доменної моделі вимагає розуміння концепцій EClass, EReference, EAttribute (рис. 2.7) та їх семантики. Крім того, EMF сам по собі не надає візуальних засобів редагування моделей, змушуючи інтегрувати додаткові інструменти, а відсутність вбудованої підтримки поведінкової семантики обмежує можливості створення повноцінних виконуваних моделей.

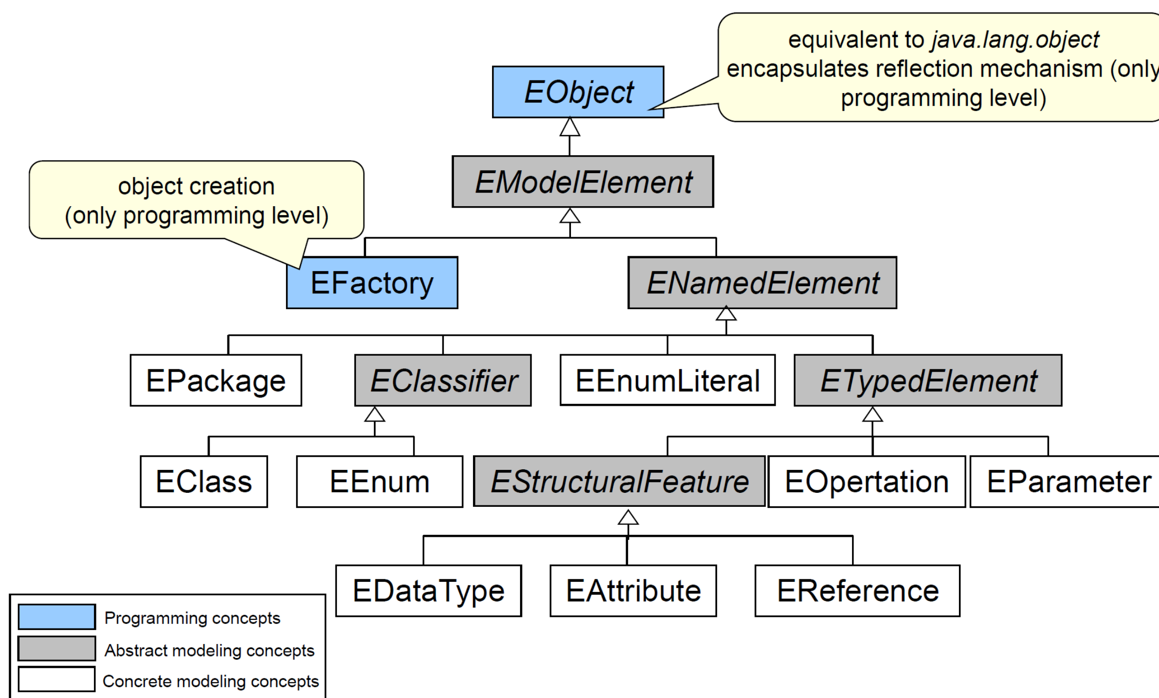

Рис. 2.7. Структура Ecore-метамоделі

Нащадком EMF є Eclipse Sirius, який являє собою потужний фреймворк для декларативного створення графічних середовищ моделювання на основі EMF-метамоделей, що дозволяє визначати візуальні нотації через спеціалізовану мову опису Viewpoint Specification без необхідності низькорівневого програмування графічних редакторів. Palladio є одним із прикладів використання Eclipse Sirius для свого середовища (рис. 2.8). Palladio – підхід до оцінки продуктивності та прогнозування продуктивності на етапі проєктування для компонентних програмних архітектур. Sirius забезпечує гнучке відображення елементів метамоделі на візуальні примітиви, такі як вузли, контейнери та з'єднання, підтримуючи множинні представлення однієї моделі та динамічне оновлення діаграм при зміні даних. Водночас значним обмеженням залишається жорстка прив'язка до Eclipse-платформи, що ускладнює розгортання для кінцевих користувачів та унеможливлює веборієнтовані сценарії використання. Складність налаштування візуальних стилів та поведінки редактора вимагає глибокого занурення у специфікацію Sirius.

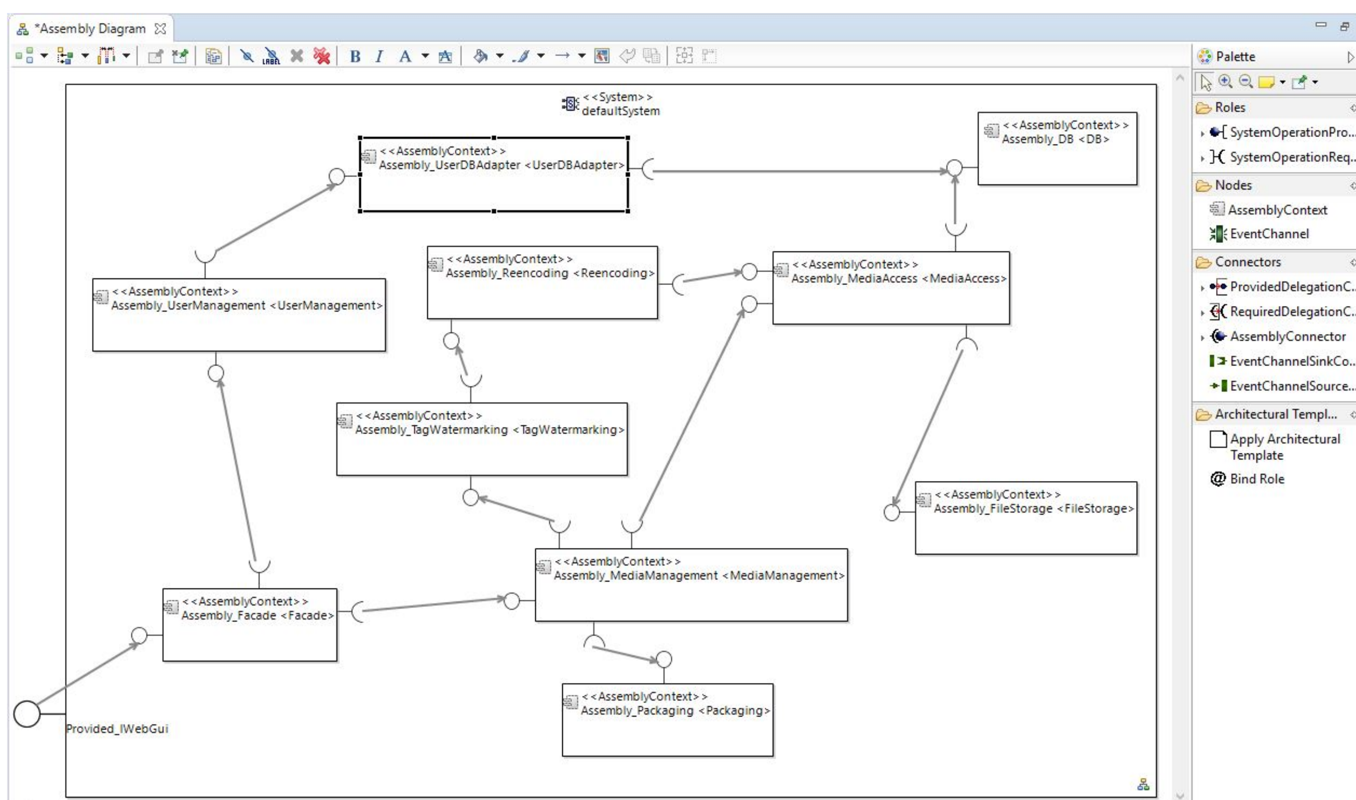

Рис. 2.8. Компонентна програмна архітектура в моделі Palladio

Acceleo [20] є інструментом для «модель-текст» трансформацій в екосистемі Eclipse, що реалізує стандарт OMG MOFM2T та дозволяє генерувати програмний код, документацію чи конфігураційні файли з EMF-моделей через декларативні шаблони з вбудованими OCL-виразами. Інструмент ефективно розв'язує задачу останньої частини MDE-ланцюжка, перетворюючи абстрактні моделі на конкретні текстові артефакти. Проте принципове обмеження Acceleo полягає в однонаправленості генерації, оскільки зміни у згенерованому коді не синхронізуються назад до моделі. А для «модель-модель» трансформацій в Eclipse-екосистемі використовується мова трансформації ATL [21].

WebGME представляє веборієнтовану платформу для модельно-орієнтованого інжинірингу, побудовану на технологіях Node.js та MongoDB, що забезпечує колаборативне редагування моделей у браузері з підтримкою версіювання та синхронізації у реальному часі. Платформа усуває необхідність встановлення десктопних застосунків та спрощує розгортання середовищ моделювання для розподілених команд. Водночас WebGME поступається рішенням на базі Eclipse в зрілості екосистеми та кількості доступних розширень, а продуктивність при роботі з великими моделями обмежується можливостями браузерного середовища.

Варто також згадати Xtext, інструмент забезпечує створення текстових DSL з автоматичною генерацією парсерів та редакторів, а інтеграція з Sirius чи веборієнтованим Sprotty (інструмент візуалізації діаграм) дозволяє доповнити текстові моделі графічними представленнями для покращення сприйняття складних архітектурних структур.

Eclipse Papyrus є провідним open-source інструментом для моделювання на базі стандартів UML 2.5 та SysML 1.6, що забезпечує повну підтримку специфікацій OMG та можливість розширення через механізм профілів, дозволяючи адаптувати стандартні нотації під специфічні доменні потреби. Платформа інтегрується з екосистемою Eclipse Modeling, що робить її привабливою для академічних досліджень та промислових проєктів з високими вимогами до стандартизації. Papyrus активно використовується в аерокосмічній та оборонній галузях завдяки підтримці профілю MARTE (UML-профіль) для моделювання систем реального

часу. Водночас інструмент успадковує складність UML-специфікації, створюючи значне когнітивне навантаження на користувачів через необхідність оперувати десятками типів діаграм та сотнями елементів нотації. Eclipse Papyrus успадковує типові недоліки Eclipse-платформи: перевантажений інтерфейс, високе споживання ресурсів та складність налаштування, що обмежує його застосування в повсякденній розробці. На рис. 2.9 зображено вигляд інтерфейсу програми Papyrus.

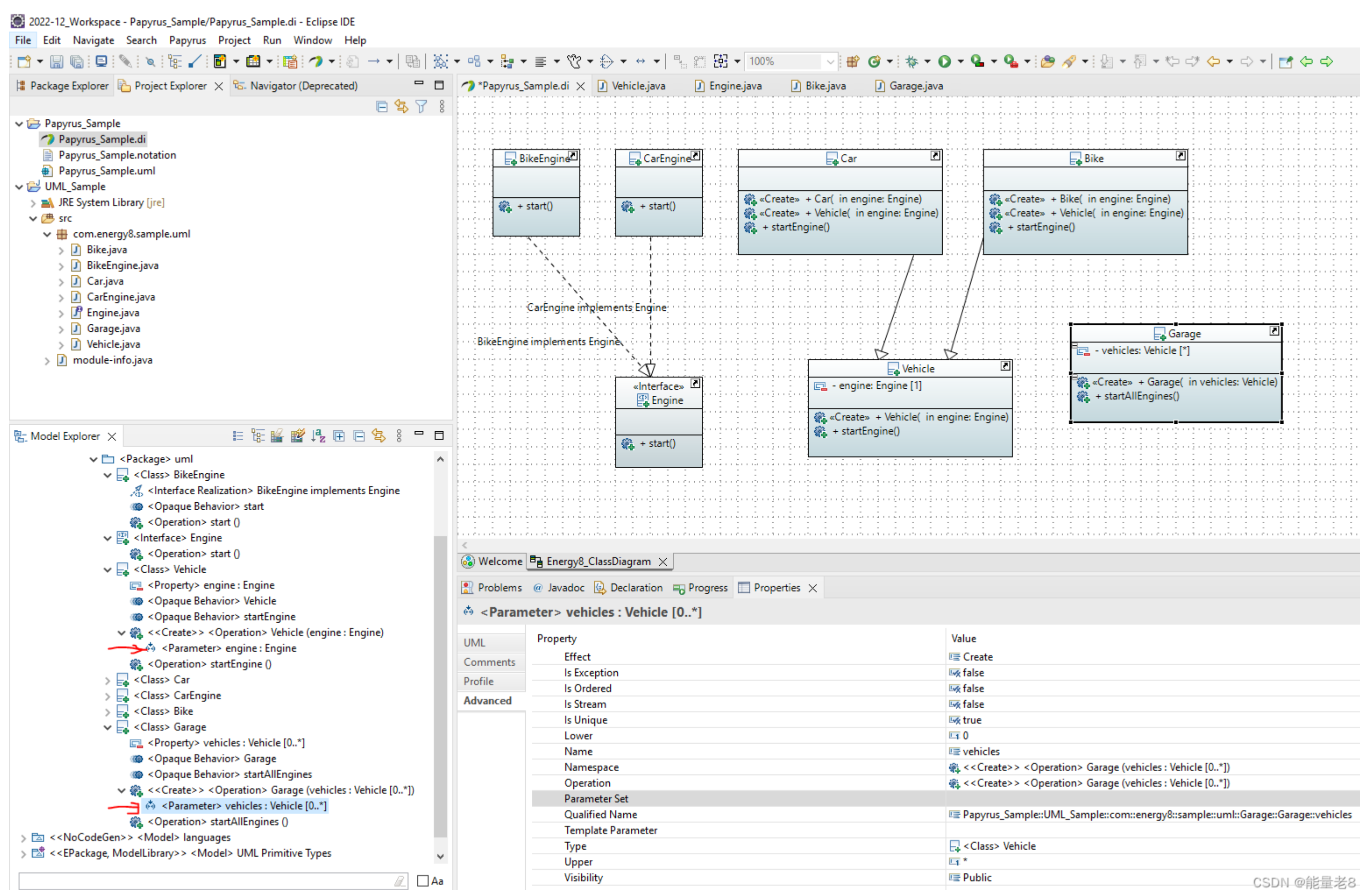

Рис. 2.9. Скріншот інтерфейсу Eclipse Papyrus з прикладом UML діаграми

Sparx Enterprise Architect є одним з найпоширеніших комерційних CASE-інструментів, що підтримує широкий спектр нотацій, включаючи UML, SysML, BPMN, ArchiMate та TOGAF, забезпечуючи єдине середовище для моделювання від бізнес-архітектури до технічної реалізації. Платформа надає розвинені можливості колаборативної роботи через централізований репозиторій, трасування вимог, генерацію документації. Enterprise Architect підтримує двосторонню (round-trip) синхронізацію між UML-моделями та кодом для багатьох популярних мов програмування. Це включає автоматичну генерацію коду з моделі (forward

engineering), реверсну інженерію (reverse engineering) з коду в модель та синхронізацію змін у обидва боки. Enterprise Architect знайшов широке застосування в корпоративному середовищі завдяки зрілості програмного функціонала (див. інтерфейс рис. 2.10) та активній підтримці розробника. Проте універсальність інструменту обертається його ж недоліком, оскільки інтерфейс перевантажений опціями та налаштуваннями, що вимагає тривалого навчання та створює когнітивний бар'єр для нових користувачів. Крім того, підтримка багатьох нотацій залишається поверхневою без глибокої семантичної інтеграції між ними, а двонаправлена синхронізація з кодом працює не для всіх аспектів.

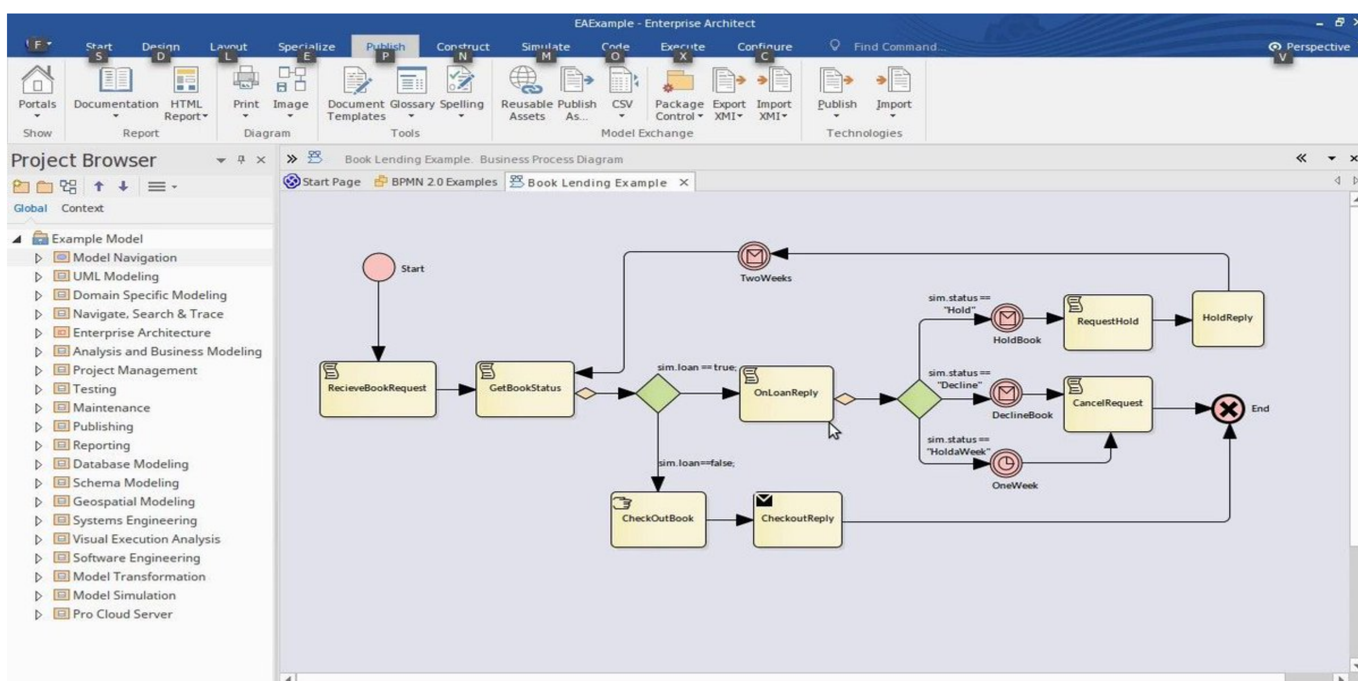

Рис. 2.10. Скріншот інтерфейсу Sparx Enterprise Architect

Modelio представляє open-source середовище для UML та BPMN моделювання з модульною архітектурою, що дозволяє розширювати функціональність через плагіни для підтримки SysML, TOGAF та генерації коду для Java, C++, C# і SQL. Інструмент забезпечує XMI-сумісність для обміну моделями з іншими CASE-засобами та підтримує базову двонаправлену синхронізацію з кодом. Водночас Modelio поступається комерційним аналогам у стабільності та повноті реалізації стандартів, а обмежена спільнота розробників плагінів звужує можливості адаптації під специфічні потреби проєктів.

Archi є спеціалізованим open-source редактором для моделювання корпоративної архітектури за стандартом ArchiMate, що здобув значну популярність у спільноті Enterprise Architecture завдяки інтуїтивному інтерфейсу (рис. 2.11) та безкоштовній ліцензії. Інструмент підтримує всі шари ArchiMate від стратегічного до технологічного, забезпечує експорт в HTML-звіти та інтеграцію з jArchi для скриптової автоматизації. Попри зручність для високорівневого архітектурного моделювання, Archi залишається інструментом документування без зв'язку з програмною реалізацією, а відсутність механізмів валідації архітектурних рішень та генерації артефактів обмежує його роль виключно візуалізаційною функцією.

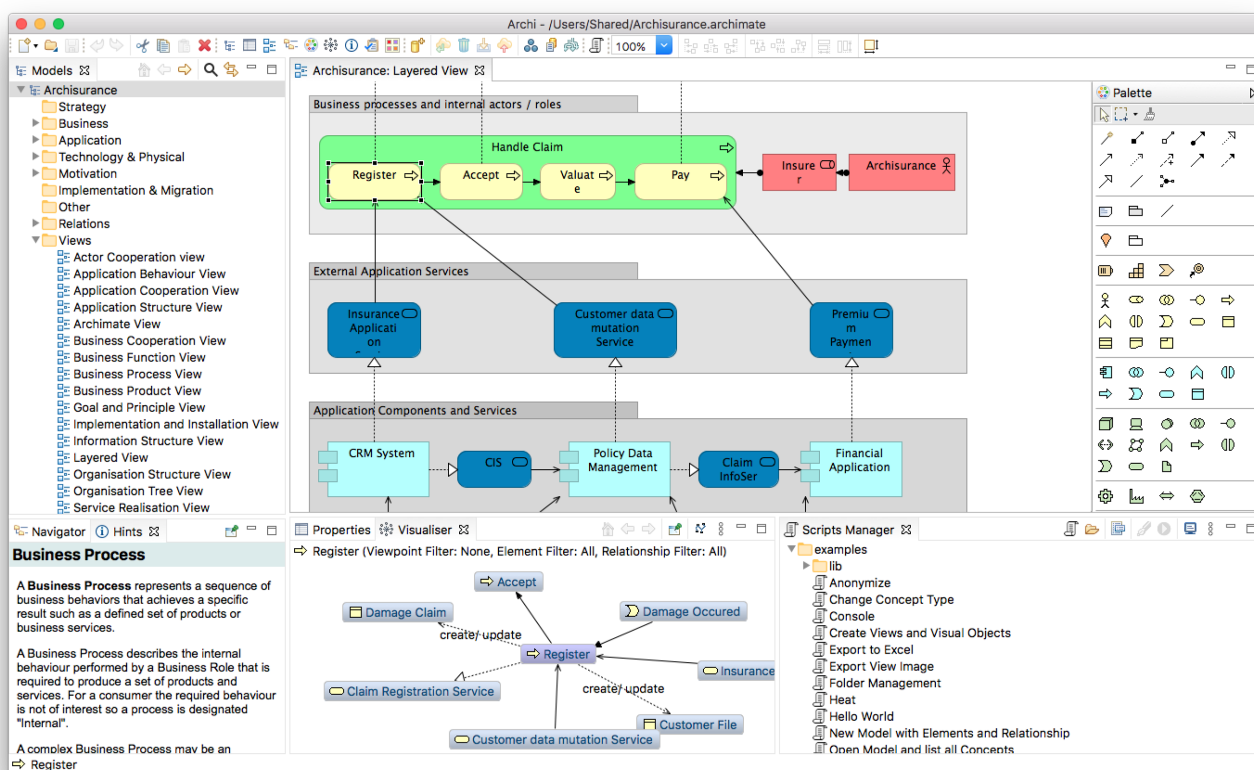

Рис. 2.11. Скріншот інтерфейсу Archi

Diagrams.net (draw.io) є безкоштовним вебредактором загального призначення з бібліотеками форм для C4, ArchiMate та UML, проте відсутність семантичної моделі перетворює його на інструмент малювання без можливостей валідації чи генерації артефактів.

Context Mapper [42] є спеціалізованим текстовим DSL для формального опису тактичних та стратегічних патернів Domain-Driven Design, що дозволяє моделювати Bounded Contexts, Context Maps, Aggregates та їх взаємозв'язки через декларативний синтаксис з подальшою автоматичною генерацією візуальних діаграм у форматі PlantUML, наведено приклад на рис. 2.12.

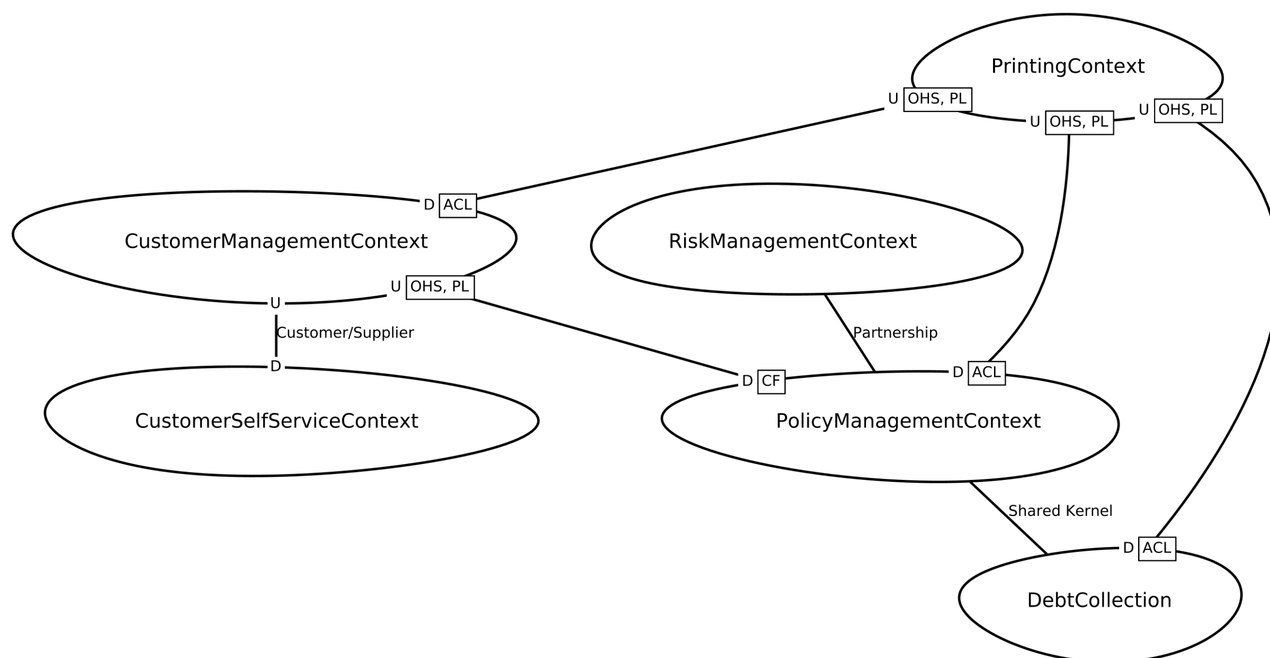

Рис. 2.12. Приклад графічної контекстної карти (Context Mapper)

Інструмент інтегрується з Eclipse та VS Code, підтримує генерацію сервісних контрактів у форматі MDSL та забезпечує трансформацію моделей для мікросервісної декомпозиції. Context Mapper заповнює важливу нішу, надаючи формальну мову для концепцій, які традиційно описувалися лише неформально у вигляді діаграм на дошках. Водночас принциповим обмеженням залишається текстова природа DSL, що вимагає від користувачів вивчення специфічного синтаксису замість інтуїтивної візуальної маніпуляції доменними концепціями. Візуалізація у Context Mapper є вторинною та односпрямованою, оскільки діаграми генеруються з тексту, але редагування діаграм не оновлює модель, що порушує принцип безпосередньої маніпуляції та створює когнітивний розрив між ментальною моделлю архітектора та її представленням в робочому інструменті.

«EventStorming» є найпоширенішою візуальною воркшоп-технікою для колаборативного виявлення доменних подій та бізнес-процесів через кольорові стікери, проте результати сесій залишаються неформалізованими артефактами без машинозчитуваної семантики та можливості автоматичної трансформації у програмні моделі. Це обмеження частково долається сучасними цифровими інструментами, такими як Qlerify (з AI-підтримкою для генерації архітектури та коду) чи інтеграціями Miro з Context Mapper, які дозволяють імпортувати стікери в формальні DSL і генерувати артефакти

Structurizr DSL, розроблений як офіційний інструмент для моделі C4, втілює концепцію «architecture-as-code», дозволяючи описувати архітектуру програмного забезпечення через текстовий синтаксис з подальшою автоматичною генерацією інтерактивних діаграм, що забезпечує версіонування моделей у Git та інтеграцію з CI/CD процесами. Приклад діаграми контейнерів для системи можна побачити на рис. 2.13.

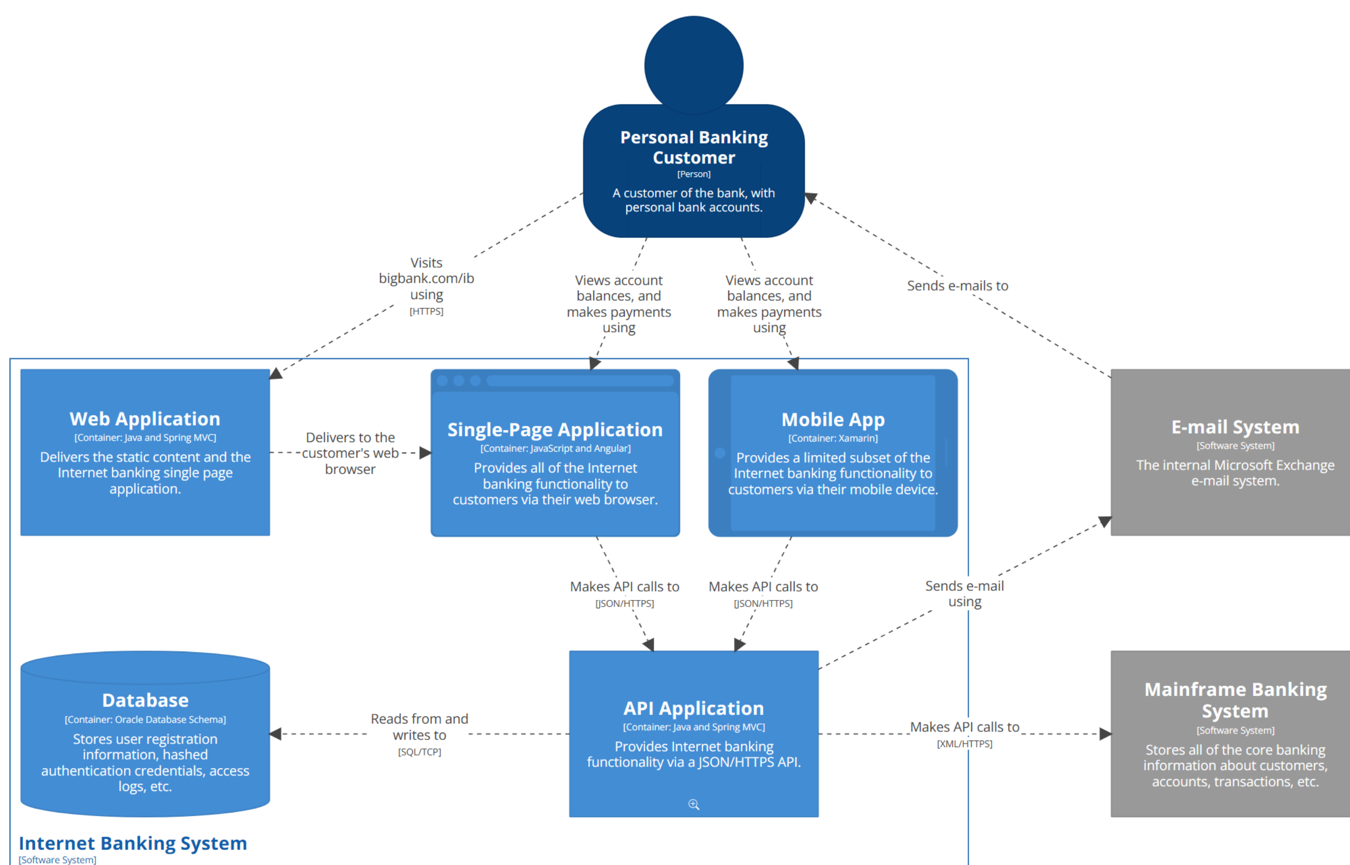

Рис. 2.13. Приклад діаграма контейнерів від Structurizr DSL

Інструмент підтримує всі чотири рівні C4 та надає можливість генерації у формати PlantUML, Mermaid та експорту в Structurizr Cloud для спільного перегляду. Попри переваги для DevOps-орієнтованих команд, Structurizr зберігає фундаментальне обмеження текстових DSL, оскільки архітектор змушений мислити синтаксисом замість візуальних патернів, а редагування відбувається у текстовому редакторі без можливості безпосередньої маніпуляції елементами діаграми. Односпрямованість трансформації від тексту до візуалізації створює когнітивний розрив між уявленням архітектора та процесом його формалізації. DSL спеціалізується на створенні C4 моделей і не забезпечує широкого спектра функцій для генерації детальних діаграм UML або інтеграції з деякими специфічними середовищами проєктування. Це може обмежувати використання для складних або нестандартних архітектурних потреб.

Інший інструмент Mermaid забезпечує створення діаграм через Markdown-подібний синтаксис з нативною підтримкою у GitHub та GitLab, проте обмежена виразність та відсутність семантичної моделі роблять його придатним лише для швидких ескізів.

PlantUML є найпоширенішим текстовим DSL для створення UML та інших типів діаграм. Розширення C4-PlantUML адаптує його синтаксис для моделювання архітектури за методологією C4, забезпечуючи інтеграцію з IDE, wiki-системами та документаційними платформами. Екосистема PlantUML включає розширення для AWS, Azure, Kubernetes та інших технологічних стеків, що робить його універсальним інструментом технічної документації. Простота синтаксису та можливість зберігання діаграм у системах контролю версій поряд із кодом забезпечили широке прийняття інструменту серед розробників. Водночас інструмент має суттєві обмеження: автоматична розмітка забезпечує мінімальний контроль над розташуванням елементів, генеруючи неоптимальні візуалізації для складних систем. Критичним недоліком є відсутність семантичної моделі, PlantUML оперує лише текстовим представленням без можливості валідації архітектурних правил чи генерації артефактів. Це робить інструмент придатним переважно для

документування, а не для повноцінного архітектурного моделювання з підтримкою трансформацій. Приклад діаграми контейнерів наведено на рис. 2.14.

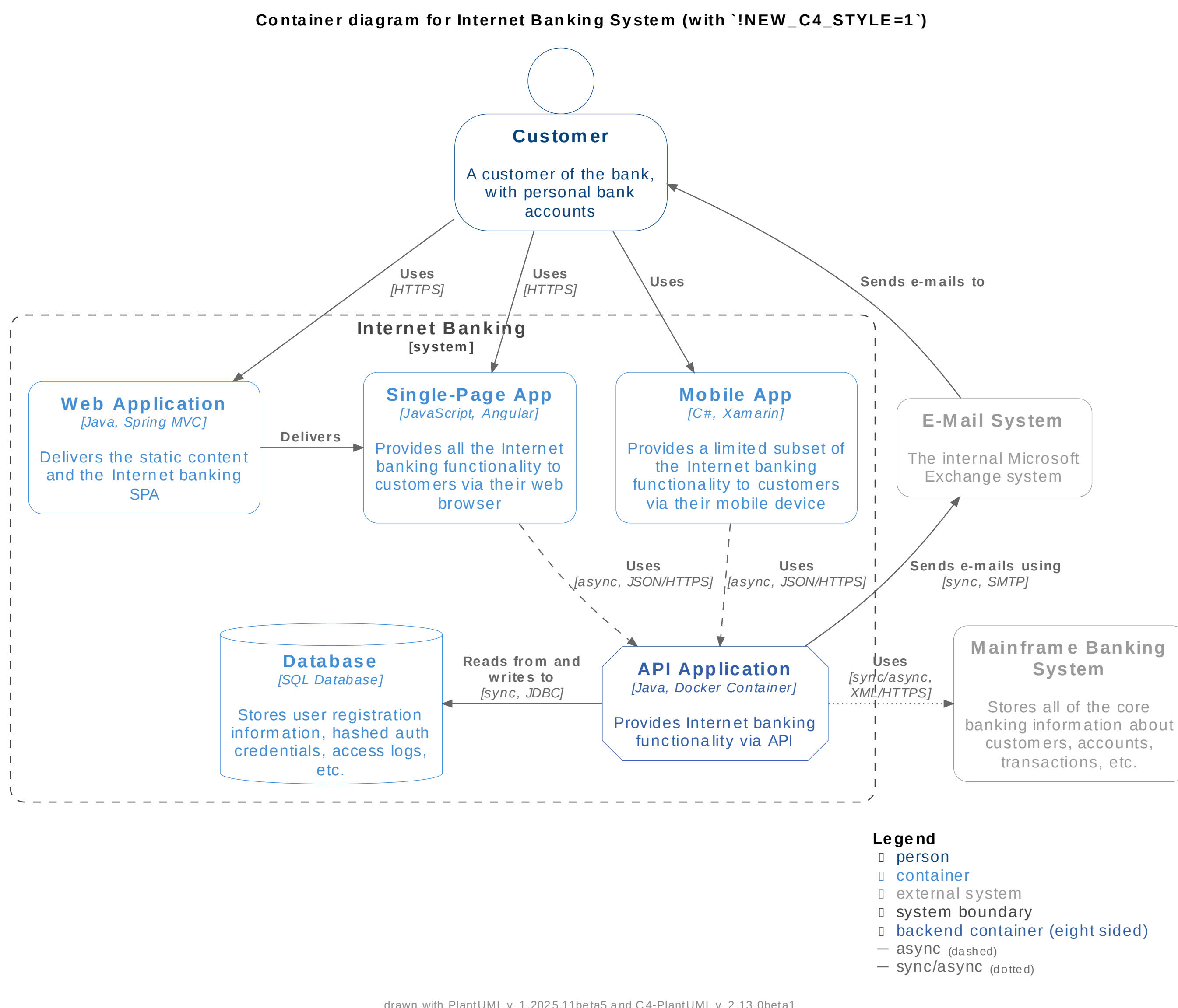

Рис. 2.14. Приклад діаграма контейнерів від PlantUML

IcePanel представляє хмарний інструмент для візуального моделювання архітектури за методологією C4, що поєднує графічний редактор (рис. 2.15) з підтримкою DDD-контекстів та інтерактивних сценаріїв взаємодії компонентів. На відміну від текстових DSL, IcePanel надає можливість безпосередньої візуальної маніпуляції елементами діаграм, що знижує когнітивний бар'єр входження. Платформа підтримує колаборативне редагування в реальному часі та версіонування архітектурних рішень, дозволяючи командам синхронізувати бачення системи та

відстежувати еволюцію архітектури. Інструмент також пропонує механізми експорту діаграм у формати PNG, SVG та JSON, що частково компенсує відсутність програмного API для інтеграції з CI/CD-конвеєрами. Водночас комерційна модель та залежність від хмарної інфраструктури обмежують доступність, а відсутність повноцінної інтеграції з MDE-екосистемою та механізмів генерації коду залишає інструмент на рівні документаційного засобу, непридатного для автоматизації життєвого циклу розробки.

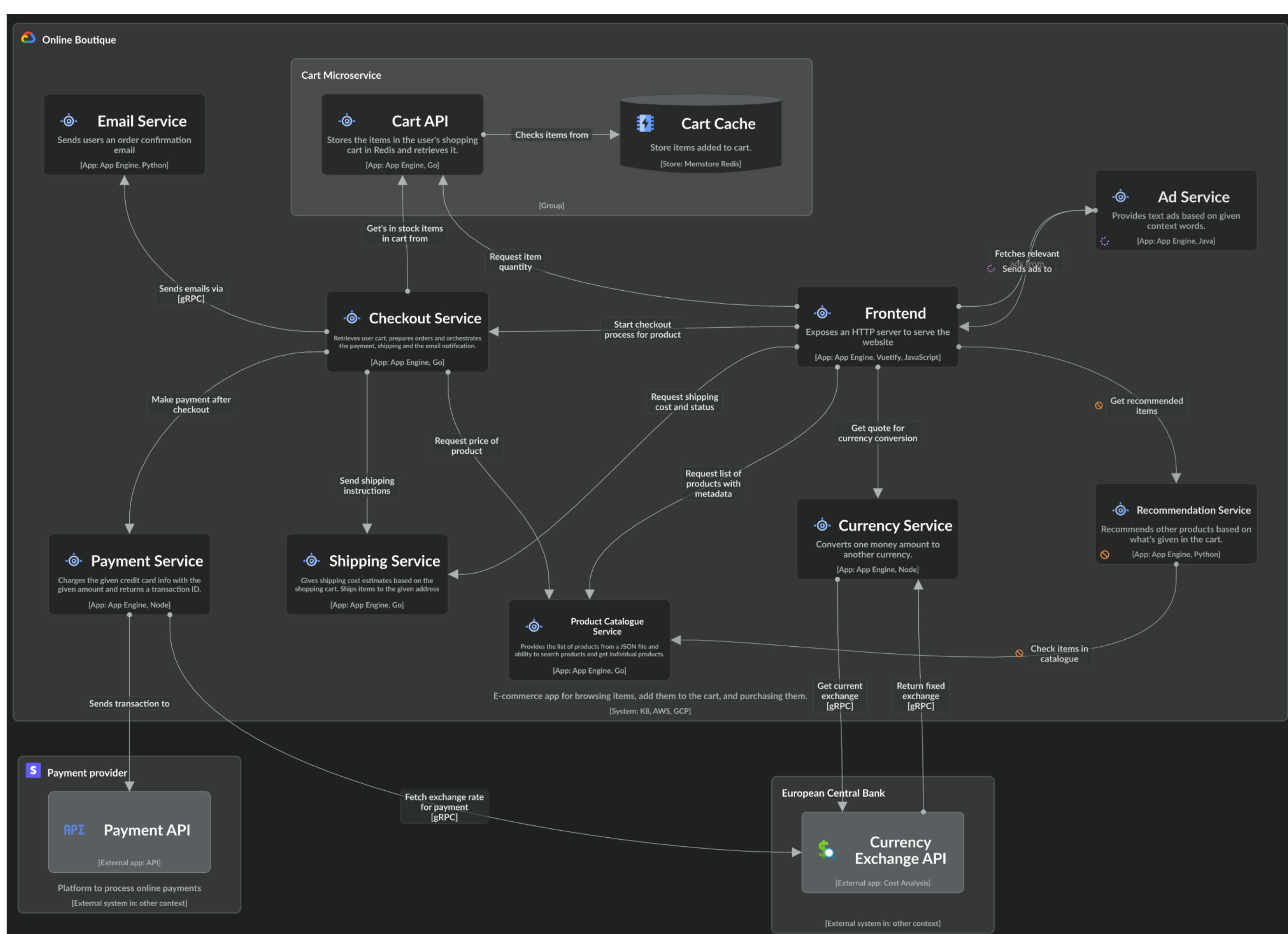

Рис. 2.15. Скріншот інтерфейсу IcePanel з прикладом C4-моделі (2 рівень)

Capella є потужним MBSE-інструментом на базі методології Arcadia для моделювання складних систем в аерокосмічній та оборонній галузях, проте його орієнтація на системну інженерію загалом, а не на програмне забезпечення зокрема, обмежує застосовність для архітектурного моделювання ПЗ.

System Composer інтегрується з екосистемою MATLAB/Simulink для моделювання архітектури вбудованих систем та систем керування, однак специфічний домен застосування та відсутність підтримки загальних архітектурних патернів ПЗ виводять його за межі типових потреб програмної інженерії.

2.2 Сучасний стан архітектурного моделювання програмного забезпечення

Архітектурне моделювання є критичним етапом інженерії програмного забезпечення, що забезпечує візуалізацію структури та поведінки системи. Упродовж останніх років відбувся зсув від громіздких графічних нотацій до прагматичних, тексто-орієнтованих та AI-асистованих підходів.

Трансформація ролі UML. Попри статус міжнародного стандарту ISO/IEC, UML зазнав змін у способах використання: від інструменту проєктування до засобу документування та комунікації. На практиці розробники обмежуються кількома основними діаграмами через складність стандарту. Актуальною залишається проблема розриву між моделлю та кодом, без постійної синхронізації UML-діаграми швидко перетворюються на «мертві» артефакти.

Модель C4. Як відповідь на виклики UML зростає популярність моделі C4 Саймона Брауна. Її ієрархічна структура дозволяє різним учасникам бачити архітектуру на потрібному рівні деталізації: бізнес-аналітики працюють із контекстом системи, архітектори – з контейнерами, розробники – з компонентами. Такий підхід допомагає уникати когнітивного перевантаження та створює спільну мову для команд.

Architecture as Code. Парадигма пропонує описувати архітектуру текстовими DSL (Structurizr DSL). Це забезпечує версійність через зберігання в Git, автоматизацію генерації візуалізацій у CI/CD та доступність для аналізу інструментами. У контексті DDD розвиваються інструменти формалізації стратегічних патернів, що дозволяють автоматично генерувати специфікації сервісів, демонструючи перехід до «виконуваної архітектури».

AI-асистоване моделювання. Інтеграція великих мовних моделей відкрила нову еру: AI допомагає генерувати архітектурні описи з текстових вимог, виявляти невідповідності, синхронізувати зміни між кодом і моделлю. Водночас остаточна перевірка має виконуватися формальними методами: автоматичною верифікацією, математичними доведеннями чи симуляціями.

Таким чином, сучасне архітектурне моделювання рухається до більшої гнучкості та автоматизації, перетворюючись із статичних схем на живі артефакти, тісно пов'язані з кодом і процесами розробки.

2.3 Тенденції та перспективи розвитку інструментів моделювання

Еволюція інструментів розробки програмного забезпечення протягом останніх десятиліть характеризується постійним підвищенням рівня абстракції. Цей процес не призводить до створення слабко спроєктованих рішень, а навпаки спрямований на абстрагування від технічної реалізації до першоджерела вимог через описову природну мову.

Найперспективнішим напрямком 2025–2030 років є синергетичне поєднання MDE, DDD, low-code платформ та AI-augmented моделювання, де кожен підхід виконує специфічну роль: DDD забезпечує структуризацію знань про предметну область, MDE формалізує їх у платформо-незалежні моделі з автоматичною трансформацією, low-code платформи роблять моделі виконуваними, а ШІ автоматизує генерацію та валідацію.

З опрацьованого під час дослідження матеріалів виявлено наступні сім перспективних тенденцій розвитку MDE.

Першою тенденцією є AI-асистоване моделювання. Автоматична генерація архітектурних моделей з природномовних вимог, валідація на узгодженість, виявлення антипатернів та рекомендаційні системи для архітектурних патернів. Обробка природної мови перетворює текстові вимоги у формалізовані UML-діаграми, скорочуючи час створення моделей.

Другою важливою тенденцією є розвиток хмарних платформ для колаборативного моделювання. Перехід від асинхронної роботи з файлами до синхронної хмарної взаємодії за принципом спільного редагування документів, що забезпечує підтримку географічно розподілених команд.

Третьою тенденцією є інтеграція модельно-орієнтованого підходу з платформами низькокодової розробки. Створення виконуваних моделей, що усувають розрив між проєктуванням та реалізацією. Автоматична генерація коду через M2M та M2T трансформації реалізує концепцію розробки на основі намірів.

Четвертою перспективною тенденцією є застосування концепції цифрових двійників до програмної архітектури. Архітектурні моделі при такому підході розглядаються як живі цифрові двійники реальних систем, що забезпечує моніторинг часу виконання безпосередньо через архітектурні моделі. Синхронізація між моделлю та реальною системою дозволяє виявляти розбіжності та забезпечувати відповідність реалізації визначеній архітектурі. Контроль архітектурної відповідності переміщується в середовище виконання та CI/CD-конвеєри з використанням функцій придатності. Синхронізація типу цифровий двійник стане стандартною практикою замість поточної ситуації частих розбіжностей між моделлю та кодом.

П'ятою тенденцією є розвиток візуальних доменно-специфічних мов. Створення спеціалізованих візуальних мов для конкретних доменів дозволяє значно підвищити виразність моделей та наблизити їх до понять предметної області. Графічні редактори доменно-орієнтованих мов та середовища метамоделювання забезпечують можливість створення власних нотацій та правил, адаптованих до специфічних потреб організацій та проєктів.

Шостою тенденцією є інтеграція модельно-орієнтованого інжинірингу з практиками безперервної розробки та операцій. Концепція ModelOps передбачає застосування CI/CD напряму з моделей, автоматизоване тестування моделей та побудову конвеєрів їх розгортання.

Сьомою тенденцією є забезпечення гібридності підходів до представлення архітектури. Візуалізація та текстове представлення більше не розглядаються як

взаємовиключні опції. Сучасні інструменти забезпечують двонаправлену синхронізацію між текстовими предметно-орієнтованими мовами та візуальними діаграмами, між природномовними запитами та формалізованими моделями уніфікованої мови моделювання, корпоративної архітектури.

Важливо підкреслити, що перехід до вищих рівнів абстракції та делегування рутинної роботи інструментам не скасовує інженерної відповідальності, а змінює її характер від написання коду до вибору правильних абстракцій, валідації згенерованих рішень та забезпечення якісних атрибутів системи. Зближене використання методологій стає визначальною характеристикою сучасної інженерії програмного забезпечення, де найперспективнішим напрямком є синергетичне поєднання модельно-орієнтованого інжинірингу, предметно-орієнтованого проєктування, низькокодових платформ, хмарно-орієнтованої архітектури та моделювання підсиленого штучним інтелектом. Такі тенденції здатні забезпечити оптимальний баланс між зниженням когнітивного навантаження на архітекторів та збереженням повного інженерного контролю над архітектурою, саме це робить їх найперспективнішими напрямками розвитку інструментів архітектурного моделювання програмного забезпечення на найближчі роки.

2.4 Підсумки огляду візуальних інструментів та методів

Аналіз сучасних підходів до архітектурного моделювання засвідчив трансформацію галузі від громіздких графічних нотацій до прагматичних, тексто-орієнтованих та AI-асистованих підходів. UML еволюціонував від інструменту проєктування до засобу документування, модель C4 пропонує ієрархічну структуру для зниження когнітивного перевантаження, парадигма Architecture-as-Code забезпечує версійність через зберігання в Git, а інтеграція великих мовних моделей відкриває можливості автоматичної генерації архітектурних описів. Виявлено сім перспективних тенденцій розвитку MDE на період 2025–2030 років: AI-асистоване моделювання, хмарні платформи для колаборативної роботи, інтеграція з low-code

платформами, застосування концепції цифрових двійників, розвиток візуальних DSL, інтеграція з DevOps-практиками та забезпечення гібридності представлення архітектури. Найперспективнішим напрямком є синергетичне поєднання MDE, DDD, low-code платформ та AI-augmented моделювання.

У межах огляду систематизовано 26 інструментів та методів за 12 категоріями (порівняльна характеристика наведена в табл. А.1): архітектурні нотації (C4, 4+1), формальні стандарти OMG та Open Group (UML, SysML, ArchiMate, AADL), методологічні фреймворки (TOGAF), MDE-платформи та інструменти (EMF, Sirius, WebGME, Acceleo, Xtext), мовні середовища (JetBrains MPS, MetaEdit+), CASE-інструменти (Papyrus, Sparx EA, Archi, Modelio, IcePanel), графічні редактори (Diagrams.net), DDD-орієнтовані інструменти (Context Mapper), візуальні методики (EventStorming), текстові DSL категорії Architecture-as-Code (Structurizr, PlantUML, Mermaid) та MBSE-інструменти для системної інженерії (Capella, System Composer).

Не всі розглянути інструменти спеціалізуються на моделюванні архітектури програмного забезпечення, однак їх було включено до дослідження через їхню часткову корисність в проблемній сфері.

Аналіз наявності формальної метамоделі показав, що більша половина розглянутих інструментів (15 із 26, тобто 58%) має повноцінну формальну метамодель, що забезпечує машинозчитувану семантику та можливість інтеграції з MDE-ланцюжком. Часткову підтримку метамоделі, яка обмежується внутрішньою структурою даних без повної відповідності стандартам MOF чи Ecore, демонструють 5 інструментів (19%). Водночас 6 інструментів (23%) не мають жодної формальної семантики, функціонуючи виключно як засоби візуалізації без можливості автоматичних трансформацій. Щодо можливостей генерації коду, ситуація виявляється значно менш оптимістичною. Лише 5 інструменти (19%) забезпечують повноцінну генерацію виконуваного коду з моделей, причому переважно це MDE-платформи та мовні середовища, такі як EMF, Acceleo, JetBrains MPS та MetaEdit+. Часткова генерація, що обмежується структурними скелетами класів без поведінкової логіки, доступна у 10 інструментах (39%). Більшість же інструментів, а саме 11 із 26 (42%), не підтримують генерацію коду взагалі,

залишаючись на рівні документаційних засобів. Найбільш критичним є показник двонаправленої синхронізації між архітектурною моделлю та програмним кодом. Жоден із розглянутих інструментів (0%) не забезпечує повної двонаправленої синхронізації, де зміни в моделі автоматично відображаються в коді, а модифікації коду оновлюють модель. Часткову синхронізацію, обмежену переважно структурними елементами, підтримують лише 6 інструментів (23%), серед яких Sparx Enterprise Architect, Modelio та Eclipse Papyrus. Переважна більшість, 19 інструментів (73%), не має жодних механізмів синхронізації, а для одного інструменту (UML як стандарт) цей критерій не застосовується.

За типом візуалізації домінують інструменти з графічним WYSIWYG-редагуванням, які становлять 17 із 26 (65%) і включають CASE-інструменти, MDE-платформи та архітектурні нотації. Підхід «текст до діаграми», характерний для Architecture-as-Code інструментів, представлений 4 інструментами (15%), зокрема Structurizr DSL, PlantUML та Mermaid. Змішаний підхід, що поєднує текстове та графічне редагування, застосовують 2 інструменти (8%), а унікальний проєкційний підхід реалізує лише JetBrains MPS (4%). Для 2 інструментів (8%), переважно MDE-компонентів на кшталт EMF та Acceleo, візуалізація не є основною функцією.

Аналіз когнітивної складності виявив, що найбільша група інструментів (10 із 26, тобто 38%) характеризується високою складністю освоєння та використання, включаючи більшість MDE-платформ та формальних стандартів. Середню когнітивну складність мають 8 інструментів (31%), переважно комерційні CASE-засоби та текстові DSL. Низьку складність, що забезпечує швидке освоєння, демонструють лише 6 інструментів (23%), серед яких архітектурні нотації C4 та прості редактори на кшталт Diagrams.net. Дуже високу когнітивну складність, що вимагає суттєвих інвестицій у навчання, мають 2 інструменти (8%), а саме JetBrains MPS із його нетрадиційною парадигмою проєкційного редагування та AADL із вузькоспеціалізованою семантикою для систем реального часу. Розподіл за типом ліцензування свідчить про домінування відкритого програмного забезпечення у сфері архітектурного моделювання. Open-source ліцензії має 21 інструмент (81%), включаючи всю екосистему Eclipse Modeling, PlantUML, Archi та інші. Комерційні

ліцензії застосовують 4 інструменти (15%), серед яких Sparx Enterprise Architect, MetaEdit+, IcePanel та System Composer. Модель freemium, що поєднує безкоштовну базову версію з платними розширеними можливостями, використовує лише Structurizr (4%).

Проведений статистичний аналіз виявляє кілька критичних закономірностей у сучасному стані інструментарію для архітектурного моделювання програмного забезпечення. По-перше, спостерігається парадокс між наявністю формальних метамоделей та їх практичною реалізацією: хоча 58% інструментів мають формальну метамодель, лише 15% забезпечують повноцінну генерацію коду, що свідчить про розрив між теоретичними можливостями MDE та їх практичним втіленням. По-друге, найбільш критичним є показник двонаправленої синхронізації: жоден з розглянутих інструментів (0%) не забезпечує повної двонаправленої синхронізації між архітектурною моделлю та програмним кодом. Часткова підтримка (23%) обмежується переважно структурними елементами, такими як класи та інтерфейси, залишаючи поведінкову логіку поза межами синхронізації. По-третє, існує обернена кореляція між потужністю інструменту та його когнітивною доступністю: найпотужніші MDE-платформи, зокрема JetBrains MPS та Eclipse Sirius, характеризуються високою або дуже високою когнітивною складністю (46% інструментів), тоді як інструменти з низькою складністю (23%) не мають формальної семантики. По-четверте, домінування текстових DSL у категорії Architecture-as-Code (15% загальної вибірки) демонструє тенденцію до версіонування архітектурних моделей, проте ціною відмови від візуального WYSIWYG-редагування, яке переважає в 65% інструментів.

Варто зазначити, що цей огляд зосереджений на візуальних інструментах архітектурного моделювання і не охоплює суміжні категорії інструментів, які потребують окремого дослідження. Малокодові платформи розробки (Low-Code/No-Code), такі як OutSystems, Mendix та Microsoft Power Platform, детально розглянуто у працях [17, 49]. Візуальні мови програмування та їх середовища, включаючи Scratch, Blockly, LabVIEW та Node-RED, систематизовано у дослідженнях [41, 50]. Ці

категорії інструментів, хоча й пов'язані з візуальним підходом до розробки програмного забезпечення, мають інший фокус застосування та цільову аудиторію.

Результатом аналізу є виявлення фундаментальної прогалини, жоден існуючий інструмент не поєднує одночасно: візуальне WYSIWYG-редагування з низьким когнітивним навантаженням, формальну метамодель для MDE-сумісності, двонаправлену синхронізацію з програмним кодом, підтримку DDD-концепцій та генерацію виконуваних артефактів. Ця прогалина безпосередньо обґрунтовує актуальність дослідження методів візуально орієнтованого модельно-орієнтованого інжинірингу, спрямованих на створення інтегрованого підходу, який би поєднував переваги візуального моделювання архітектури з можливостями автоматичної генерації, водночас ураховуючи когнітивні обмеження людського сприйняття через принципи людиноцентричного дизайну. Запропонований у роботі підхід, що передбачає поєднання візуальних MDE-рішень із візуальними мовами програмування та елементами DDD, адресує саме цю системну прогалину, пропонуючи еволюційний синтез існуючих практик. Такий синтез має потенціал трансформувати архітектурне моделювання з документаційної активності, відірваної від розробки, на центральний елемент повного циклу створення програмного забезпечення, де архітектура стає виконуваним артефактом, а не лише візуальним описом намірів.

2.5 Висновки до розділу

У другому розділі здійснено комплексне дослідження предметної області візуального модельно-орієнтованого інжинірингу, що охопило порівняльний аналіз 26 інструментів за 12 категоріями. Аналіз виявив критичну системну прогалину: попри те, що 58% інструментів мають формальну метамодель, лише 19% забезпечують генерацію коду. Крім того, встановлено обернену кореляцію між потужністю інструментів та їх когнітивною доступністю, що свідчить про невирішеність фундаментальних проблем галузі.

Дослідження архітектурних нотацій продемонструвало, що модель C4 ефективно знижує когнітивне навантаження через ієрархічну структуру, тоді як модель 4+1 страждає від проблеми консистентності між представленнями. UML залишається формальним стандартом, проте його надмірна складність та розрив між моделлю і кодом обмежують практичне застосування. ArchiMate орієнтований на Enterprise Architecture без зв'язку з програмною реалізацією. Найбільш критичним є показник двонаправленої синхронізації, де жоден із розглянутих інструментів не забезпечує повної синхронізації між архітектурною моделлю та кодом, а часткова підтримка (23%) обмежується структурними елементами.

Виявлено сім перспективних тенденцій розвитку на період 2025–2030: AI-асистоване моделювання, хмарні колаборативні платформи, інтеграція з low-code, концепція цифрових двійників архітектури, візуальні DSL, ModelOps та гібридність представлення. Результатом аналізу є обґрунтування актуальності розробки інтегрованого підходу, що поєднує візуальне WYSIWYG-редагування з формальною метамоделлю, двонаправленою синхронізацією та підтримкою DDD-концепцій.

Варто зазначити, що до порівняльного аналізу свідомо включено інструменти, які не спеціалізуються безпосередньо на архітектурному моделюванні програмного забезпечення, зокрема засоби корпоративної архітектури (ArchiMate, TOGAF), малокодові платформи розробки та візуальні мови програмування. Таке розширення меж дослідження обумовлене методологічним принципом, згідно з яким інноваційні розв’язання складних проблем часто виникають на перетині різних галузей шляхом запозичення та адаптації підходів із суміжних областей. Корпоративна архітектура пропонує зрілі практики багаторівневого моделювання та управління складністю, малокодові платформи демонструють успішні приклади автоматичної генерації коду з візуальних моделей, а візуальні мови програмування напрацювали ефективні когнітивно-ергономічні рішення для зниження порогу входження та підвищення продуктивності. Синтез цих підходів може забезпечити комплексне розв’язання виявлених проблем архітектурного моделювання.

РОЗДІЛ 3
КОНЦЕПТУАЛЬНЕ ПРОЄКТУВАННЯ ІНСТРУМЕНТУ ВІЗУАЛЬНОГО МОДЕЛЬНО-ОРІЄНТОВАНОГО ІНЖИНІРИНГУ

3.1 Аналіз виявлених проблем, концептуалізація проєктованого рішення

Після дослідження предметної області варто підсумувати результати, знайдено 14 проблем, які відповідають напрямку теми даної наукової роботи, виклад у наступному списку:

1. Традиційні текстові підходи до проєктування архітектури ускладнюють розуміння системи різними зацікавленими сторонами та ускладнюють еволюцію архітектури програмного забезпечення.

2. Когнітивні обмеження розробників, зокрема обмежена місткість робочої пам'яті (4 елементи для простих об'єктів та 1-2 для складних), суттєво впливають на продуктивність та якість роботи при проєктуванні складних систем.

3. Швидкість обробки текстової інформації значно нижча за візуальну, а навігація по тисячах рядків коду призводить до значного когнітивного навантаження, зростання кількості помилок та втрати контексту.

4. Чітке розділення підходів MDE: одні інструменти моделюють архітектуру високого рівня без генерації коду, інші генерують код без засобів контролю архітектури програмного забезпечення.

5. Показник двонаправленої синхронізації між архітектурною моделлю та програмним кодом становить 0% для повної синхронізації та лише 23% для часткової, що свідчить про фундаментальну невирішеність цієї проблеми.

6. Парадокс: 58% інструментів мають формальну метамодель, але лише 19% забезпечують генерацію коду.

7. Існує обернена кореляція між потужністю інструменту та його когнітивною доступністю, де найпотужніші платформи характеризуються високою складністю освоєння, тоді як інструменти з низькою складністю не мають формальної семантики (система правил, яка однозначно визначає значення конструкцій

інструменту.). Виникає парадокс семантики, де інструменти з багатою формальною семантикою дають величезну силу для моделювання та аналізу, але вони важкі для когнітивного освоєння. З іншого боку інструменти, які легко зрозуміти, часто жертвують формальною точністю й виразністю.

8. Домінування текстових DSL у категорії «Architecture-as-Code» створює вибір між версійним контролем та інтуїтивністю візуального редагування.

9. UML характеризується надмірною складністю, яка створює значне когнітивне навантаження на розробників, а автоматична генерація повнофункціонального коду з UML-моделей залишається нерозв'язаною задачею.

10. Проблема підтримки консистентності між множинними архітектурними представленнями, де зміни в одному погляді мають бути вручну синхронізовані з іншими через відсутність вбудованих механізмів автоматичної верифікації та трасування залежностей.

11. Візуальні мови програмування мають проблеми масштабованості у великих проєктах, обмежену виразність порівняно з текстовими мовами та складність версійного контролю.

12. Комерційні малокодові платформи фокусуються на реалізації типових бізнес-процесів та інтерфейсів, часто нехтуючи глибоким архітектурним моделюванням складних доменів.

13. Еволюція до AI-оркестрування при розробці програмного забезпечення створює ризик втрати контролю над архітектурною цілісністю та детермінованістю великих систем.

14. Жоден розглянутий інструмент не поєднує візуальне редагування з низьким когнітивним навантаженням, формальну метамодель, двонаправлену синхронізацію, підтримку DDD та генерацію артефактів.

Для знайдених проблем можна сформувати наступні рекомендації по їх розв’язанню, імплементація яких покращить інструменти архітектурного моделювання програмного забезпечення.

Застосування візуальних нотацій знизить бар'єр сприйняття системи та спростить еволюцію архітектури (проблема 1). Візуальне середовище стає

когнітивним протезом завдяки преатентивній обробці, принципам гештальт-психології та зовнішньому когнітивному розвантаженню.

Застосування когнітивно-ергономічних принципів зменшить негативний вплив когнітивних обмежень (проблема 2): мінімалістичний інтерфейс, розділення робочих просторів, обмеження одночасного відображення до 4-5 елементів.

Робота у справді візуальному середовищі, де мозок сприймає елементи як цілісні образи, розв’язує проблему низької швидкості обробки текстової інформації порівняно з візуальною та значного когнітивного навантаження при навігації по тисячах рядків коду (проблема 3). Мозок бачить елементи як цілісні образи, тому навігація стає інтуїтивною. Це забезпечується миттєвим розпізнаванням кольору, форми та розташування, використанням діаграм і схем, а також опорою на просторову пам’ять. Архітектура подається як мережа вузлів і зв’язків, що робить орієнтацію зрозумілою та природною. Замість лінійних каталогів застосовується гнучкий граф, який дає можливість рухатися у двох вимірах і масштабувати простір.

Інтеграційний підхід, що поєднує MDE та візуальні мови програмування з генерацією коду, розв'язує проблему розділення підходів (проблема 4), забезпечуючи контрольовану архітектуру та автоматизовану реалізацію.

Проблема двонаправленої синхронізації між архітектурною моделлю та програмним кодом може бути розв'язана двома шляхами (проблема 5). Перший шлях передбачає розробку спеціалізованих механізмів синхронізації та їх повну імплементацію в інструменті. Другий шлях передбачає перехід до нової парадигми програмування через візуальні мови програмування, де програмний код стає лише проміжним артефактом розробки, що генерується автоматично і не потребує ручного допрацювання, тоді модель є єдиним джерелом істини, а проблема синхронізації усувається на концептуальному рівні.

Розробка метамоделі з урахуванням вимог генерації коду вирішує парадокс між метамоделями та генерацією (проблема 6), включаючи структурну та поведінкову семантику.

Застосування принципу поступового розкриття складності розв’язує проблему оберненої кореляції між потужністю інструменту та його когнітивною

доступністю, відому як парадокс семантики (проблема 7). Інструмент надає простий інтерфейс для типових операцій, але дозволяє доступ до повної потужності формальної семантики за потреби, де когнітивно-ергономічний дизайн інтерфейсу приховує складність метамоделі за інтуїтивними візуальними операціями.

Гібридний підхід із синхронізацією між текстовим представленням (JSON, XML) та візуальним редактором розв'язує проблему 8.

Фокус на прагматичній підмножині UML замість спроби охопити всі 14 типів діаграм розв’язує проблему надмірної складності (проблема 9). Це досягається застосуванням ієрархічного підходу моделі C4, де різні рівні абстракції Context, Containers, Components та Code адресують потреби різних зацікавлених сторін, а також інтеграцією з візуальними мовами програмування для опису поведінкової логіки замість громіздких діаграм послідовностей чи діяльності UML.

Єдина внутрішня модель з якої автоматично генеруються всі представлення, розв’язує проблему підтримки консистентності між множинними архітектурними представленнями та необхідності ручної синхронізації змін (проблема 10). Зміна в будь-якому представленні оновлює внутрішню модель, що автоматично синхронізує всі інші представлення, з вбудованими механізмами автоматичної верифікації консистентності та трасування залежностей між елементами різних представлень.

Ієрархічна декомпозиція з можливістю згортання модельованих підсистем та спеціалізована система версіонування розв'язує проблеми масштабованості візуальних мов програмування у великих проєктах, обмеженої виразності та складності версійного контролю (проблема 11). Це включає суворе логічне групування та доменне розшарування програмних пакетів як простору зв'язаних об'єктів, чотирирівневу систему версіонування з окремою історією для кожного елемента, мультиелементарними віхами для узгоджених станів групи елементів, епохами для цілісних проєкцій моделі та традиційними Git-репозиторіями для згенерованого коду, а також розширення виразності через інтеграцію з текстовими вставками для специфічних випадків (напівструктуровані файли з власною семантикою), де візуальне представлення є неефективним.

Інтеграція надто складних інструментів у малокодові платформи суперечить їхній базовій ідеї простоти та доступності, адже перевантажене середовище втрачає актуальність для користувачів (проблема 12). Натомість доцільно створювати спеціалізований клас інструментів програмного забезпечення, орієнтований на складні завдання з підтримкою архітектурного моделювання безпосередньо у середовище візуальної розробки.

Розробка та впровадження гібридних методів візуального модельно-орієнтованого інжинірингу у формі MDE 2.0 розв’язує проблему ризику втрати контролю над архітектурною цілісністю та детермінованістю великих систем при еволюції до AI-оркестрування (проблема 13). Наразі такі методи перебувають переважно на стадії дослідження або лише починають свій розвиток

Проблема відсутності інструменту, який би поєднував візуальне редагування з низьким когнітивним навантаженням, формальну метамодель, двонаправлену синхронізацію, підтримку DDD та генерацію артефактів, потребує системного підходу до розв'язання (проблема 14). Розробка окремого прототипу, хоча й демонструє можливість інтеграції запропонованих концепцій, неминуче успадкує недоліки застосованих підходів через їхню початкову несумісність. Повноцінне рішення вимагає побудови нового стандарту на основі систематизації накопиченого досвіду у сферах модельно-орієнтованого інжинірингу, візуальних мов програмування та когнітивної ергономіки, розробленого у співпраці з провідними експертами та організаціями галузі.

Взявши за основу знайдені рішення в даній науковій роботі, а також беручи до уваги опубліковані результати статті [10] про теоретичні аспекти візуальних мов програмування, пропонується створити нову парадигму розробки програмного забезпечення під назвою «Когнітивно-ергономічна холістична візуальна парадигма розробки», абревіатура «КЕХВПР». Відповідна англійська назва «Cognitive-Ergonomic Holistic Visual Development Paradigm», абревіатура «CEHVDP». Наступний текст у формі 4 пунктів описує її суть.

1. Когнітивно-ергономічний дизайн середовища розробки. Інструментарій враховує когнітивні обмеження людини при проєктуванні інтерфейсу та організації

робочого простору для зниження ментального навантаження та підвищення продуктивності інженера.

2. Холістична тришарова метамодель з валідацією повноти. Всеохопне моделювання охоплює три семантично пов'язані рівні: бізнес-архітектура, програмна архітектура та програмна логіка. Рівні поступово деталізують один одного, метамодель автоматично валідує повноту зв'язків між ними.

3. Візуальне WYSIWYG-редагування як основний режим роботи. Інженер працює виключно з графічними об'єктами через візуальний інтерфейс. Модель є єдиним джерелом істини, всі представлення генеруються автоматично. Текстове втручання допускається лише для специфічних випадків (DSL-фрагменти, скрипти).

4. Візуальна мова програмування для повної генерації коду. Програмна логіка моделюється спеціалізованою візуальною мовою з формальною семантикою, достатньою для автоматичної генерації повнофункціонального коду без ручного допрацювання. Моделювання бізнес-архітектури та програмної архітектури здійснюється окремими нотаціями.

Визначені пункти є взаємопов'язаними, доповнюють один одного і залежні між собою наступним чином.

Когнітивно-ергономічний дизайн створює фундамент парадигми. Визнання обмежень людського сприйняття робить неможливим застосування традиційних текстових підходів для складних систем. Мозок обробляє візуальні образи значно швидше тексту, преатентивно [7] розпізнаючи форми, кольори та просторові відношення. Це створює запит на холістичне моделювання: система має надавати різні рівні абстракції для різних завдань, де кожен рівень оптимізований для специфічного когнітивного навантаження.

Холістична метамодель є наслідком когнітивних вимог і встановлює контракт для візуалізації. Поділ на три рівні дозволяє розділити когнітивне навантаження: бізнес-аналітик працює з акторами та прецедентами, архітектор проєктує пакети та класи, програміст моделює алгоритми – кожен на своєму рівні без опанування всієї складності одночасно. Семантичні зв'язки та автоматична валідація створюють захист: система не дозволить забути про реалізацію бізнес-вимоги або залишити

архітектурний елемент ізольованим. Поступова деталізація формує природний когнітивний шлях від абстрактного до конкретного.

Візуальне WYSIWYG-редагування є технічним втіленням попередніх принципів. Когнітивна ергономіка вимагає роботи з цілісними образами замість послідовностей символів, холістична метамодель потребує видимості зв'язків між рівнями, це можливе лише через графічне представлення. Обмеження текстового втручання є критичним, будь-яке змішування візуального та текстового руйнує когнітивні переваги. Це встановлює вимогу, що програмна логіка також має моделюватись виключно візуально. Текст допускається для виводу іменувань елементів нотацій та побудови елементів інтерфейсу IDE.

Візуальна мова програмування замикає парадигму, перетворюючи її на повноцінне середовище розробки. Без можливості моделювати виконувану логіку візуально інженер змушений перемикатися на текстовий код, що руйнує когнітивну послідовність та розриває метамодель. Повна генерація коду усуває проблему двонаправленої синхронізації: модель стає єдиним джерелом істини, а код компільованим артефактом. Утворюється замкнений цикл, де когнітивні обмеження диктують візуалізацію, візуалізація вимагає холістичної структури, структура реалізується через WYSIWYG, WYSIWYG доповнюється візуальною мовою, візуальна мова знижує когнітивне навантаження.

Автор парадигми планує розвивати її ідеї також в майбутніх дослідженнях.

Для середовищ розробки, які будуть реалізовувати цю парадигму пропонується наступний набір рішень.

Інструменти візуального модельно-орієнтованого інжинірингу для архітектурного моделювання програмного забезпечення мають охоплювати три взаємопов'язані рівні: бізнес-моделювання для визначення акторів, прецедентів та візуального опису історій користувачів (англ. story telling); архітектурне моделювання для проєктування внутрішньої структури програмної системи (детальніше див. розділ 2); та рівень програмної логіки для візуального викладу поведінки програми засобами візуальної мови програмування. Концепція Domain-Driven Design надає цінні патерни для виділення абстракцій та методики виявлення

доменних подій, однак пряма імплементація DDD як обов'язкового фреймворку обмежує запропоновану холістичну метамодель, яка є ширшою за межі предметно-орієнтованого проєктування. Доцільним є запозичення концептуальних досягнень DDD без жорсткої прив'язки до його термінології, що дозволяє адаптувати найкращі практики до потреб візуального модельно-орієнтованого інжинірингу та зберігає гнучкість для підтримки альтернативних методологій.

Оскільки використовується візуальна мова програмування питання двонапрямної синхронізації «модель-код» та «код-модель» стає не актуальним. Попри наведений опис архітектури для візуальних мов програмування в статті [10], деталі її реалізації не розкрито, для цього варто переглянути розділ 2.

Запропонована парадигма CEHVDP не вимагає обов'язкової реалізації на базі MOF та EMF через три ключові обмеження: архітектурну надмірність, що суперечить принципам мінімізації когнітивного навантаження; жорстку технологічну залежність від екосистеми Eclipse; та семантичні розбіжності між об’єктною природою MOF і потребами візуальних мов. Відтак, доцільнішим є власний метамоделювальний підхід, який забезпечує гнучкість, пряме відображення тришарової структури та відсутність зайвих абстракцій. Проте пріоритетом для інтеграції з проєктами, що існують, має стати реалізація трьох механізмів реверсної інженерії: модуля парсингу програмного коду для імпортування програмної логіки та її візуалізації через нотацію VPL, аналізатора структури проєкту для парсингу та побудови архітектурних моделей та інструменту трансформації зовнішніх бібліотек у візуальні компоненти для повторного використання.

Рекомендується спеціалізована чотирирівнева система версіонування, що забезпечує повноцінне управління історією змін візуальних моделей та інтеграцію зі стандартними інструментами розробки. Перший рівень забезпечує збереження історії кожного окремого елемента з використанням порядкових індексів або часових міток та можливістю оптимізації через видалення надлишкових проміжних станів. Другий рівень реалізує мультиелементарне версіонування через механізм віх, що фіксують узгоджений стан групи пов'язаних елементів для запобігання втрати цілісності зв'язків при скасуванні змін. Третій рівень представляє версіонування

епох як цілісних проєкцій стану програмного забезпечення на певний момент часу. Четвертий рівень використовує традиційні Git-репозиторії для збереження згенерованого коду та інших артефактів, забезпечуючи сумісність з інструментами CI/CD та практиками DevOps, де версії дозволяють однозначно ідентифікувати частини коду, що належать різним епохам моделювання.

Рекомендується для розробки інтерфейсів програм в середовищі реалізувати RAD-підхід, де графічні елементи (із набору інструментів) будуть базуватися на одній із декларативних мов розмітки (наприклад HTML5) або декларативному UI-фреймворку (наприклад Jetpack Compose для Kotlin). Особливості реалізації архітектури такого підходу для VPL варто глянути в статті [10].

Для перевірки концепції парадигми в цьому розділі наведено етапи проєктування прототипу інструменту візуального модельно-орієнтованого інжинірингу, а в четвертому розділі викладено процес його реалізація та тестування.

Сформовано наступні задачі для проєктування прототипу:

– спроєктувати візуальну нотацію для архітектурного моделювання та для програмної логіки;

– реалізувати архітектурне представлення згідно з парадигмою «CEHVDP» та принципами візуальних мов програмування [10];

– запровадити механізми генерації програмної структури та коду Java-програм із моделей представлень;

– провести тестування прототипу реалізованого продукту.

Представлення бізнес-архітектури, програмної логіки та функціонал валідації моделей, контролю версіювання на ранніх етапах прототипу реалізовано не буде.

Отже, на основі систематизації виявлених 14 проблем сформульовано відповідні рекомендації та запропоновано нову парадигму «Когнітивно-ергономічна холістична візуальна парадигма розробки» (CEHVDP). Ця парадигма інтегрує чотири взаємопов'язані принципи: когнітивно-ергономічний дизайн середовища, холістичну тришарову метамодель з валідацією повноти, візуальне WYSIWYG-редагування як основний режим роботи та візуальну мову програмування для повної генерації коду. На базі сформульованих принципів визначено завдання для

проєктування прототипу інструменту, що реалізує запропоновану парадигму та дозволить верифікувати її практичну застосовність.

3.2 Проєктування архітектурного представлення

Інструмент візуального модельно-орієнтованого інжинірингу складається з двох важливих представлень «архітектурного» та «програмної логіки». Архітектурне представлення – формальна візуальна модель, що описує структурну організацію системи: декомпозицію на пакети, класи, інтерфейси, компоненти, їх залежності та взаємозв'язки на рівні проєктних рішень. Багаторівневе моделювання дозволяє описувати архітектуру на різних рівнях деталізації від високорівневих концептуальних моделей до детальних технічних специфікацій. Кожен рівень використовує відповідні візуальні нотації та абстракції, забезпечуючи поступове уточнення архітектурних рішень. Для кращого розуміння впливу принципів VPL [10] на архітектурне представлення інформація про них згрупована в табл. 3.1.

Таблиця 3.1

Нюанси архітектури VPL для архітектурного представлення

Нюанс	Суть
Доменна декомпозиція	Кожен рівень архітектури поділяється на домен, який може мати підрівні та розбиватися на менші частини (Принцип 3)
Розділення рівнів перегляду	Дочірній контент та батьківські зв'язки не відображаються одночасно на одній діаграмі (Принцип 4)
Компонентний підхід	Система складається з модулів, що підтримують декомпозицію, повторне використання та поширення бібліотек (Принцип 5)
Посилання замість реалізацій	Класи містять посилання на домени даних і методів, а не безпосередні реалізації (Принцип 6)
Графова структура пакетів	Пакети представляються як двовимірна область, де кожна вершина графа може приховувати новий підграф (Принцип 10)
Семантична мережа класів	Класи пов'язані через семантичну модель, де вузли – класи, а ребра – прецеденти їх методів (Принцип 15)
Лінійний шаблон зв'язків	Частини архітектури мають лише двох сусідів, що спрощує перевірку синтаксису та забезпечує гнучкість

Наведені принципи мають практичне застосування і впливають на архітектурне представлення наступним чином, див. табл. 3.2

Таблиця 3.2

Вплив нюансів VPL на архітектурне представлення

Нюанс	Практичний результат для інженера
Доменна декомпозиція	Інженер працює із групованими елементами одного домену, утримуючи в пам'яті 4 об'єктів замість десятків
Розділення рівнів перегляду	Зникає когнітивне перевантаження від одночасного відображення батьків, поточного рівня та дітей
Компонентний підхід	Виділення нового модуля або бібліотеки виконується переміщенням візуальних блоків без ручного виправлення залежностей
Посилання замість реалізацій	Зміна структури методу або даних в одному місці автоматично оновлює всі класи, що на них посилаються
Графова структура пакетів	Навігація у двох вимірах із масштабуванням замість прокручування лінійних списків тек
Семантична мережа класів	Залежності між класами видимі одразу на діаграмі, не потрібно шукати зв'язки по файлах
Лінійний шаблон зв'язків	Автоматична валідація коректності архітектури без складних правил перевірки

У візуальній нотації поняття «Корінь» використовується як початковий контейнер проєкту, що є точкою входу до всієї архітектурної ієрархії системи. Корінь містить всі елементи нижчого рівня та слугує єдиним джерелом для навігації до всіх структурних елементів. Приховує внутрішні елементи до моменту відкривання.

У візуальній нотації поняття «Пакет» використовується як структурна одиниця групування всіх інших елементів. Фокус робиться на зберігання доменів класів та класів, але може містити також інші елементи, в тому числі свого типу (окрім кореня). Приховує внутрішні елементи до моменту відкривання.

У візуальній нотації поняття «Клас» використовується як узагальнене позначення будь-якого типу структурного елемента мови Java, що бере участь у побудові архітектури системи. Це дозволяє спростити моделювання та уніфікувати представлення різних сутностей, адже незалежно від їх конкретної природи (клас, інтерфейс чи перелік) вони виконують роль будівельних блоків об’єктно-орієнтованої моделі. Містить домени методів та домени даних. Приховує внутрішні

елементи до моменту відкривання. На рис. 3.1 наведено візуальну нотацію архітектурного моделювання.

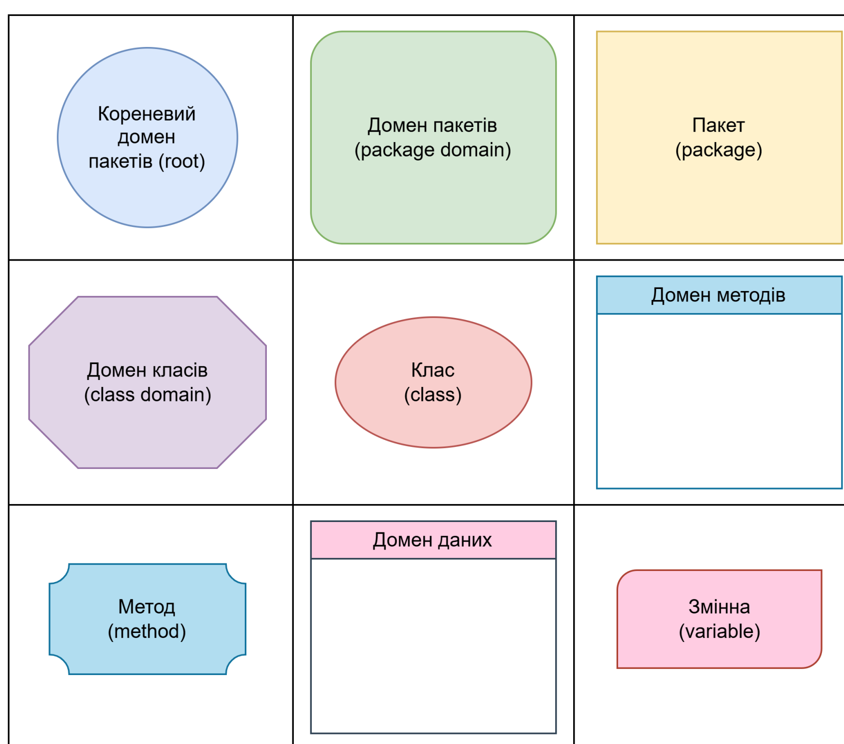

Рис. 3.1. Візуальна нотація інструменту для архітектурного моделювання

Під доменом пакетів розуміється простір групування пакетів, однак може розміщувати інші елементів, в тому числі свого типу (окрім кореня). Приховує внутрішні елементи до моменту відкривання.

Під доменом класів розуміється простір розміщення класів. Домен класів визначає спільну категорію для структурних елементів (класів, інтерфейсів, переліків). Може існувати в просторі всіх верхніх рівнів, а вміщувати в собі тільки класи. Приховує внутрішні елементи до моменту відкривання.

У візуальній нотації поняття «Метод» використовується як елемент, що описує операцію або поведінку класу. Методи розміщуються в домені методів і містять посилання на програмну логіку, яка моделюється візуально.

У візуальній нотації поняття «Змінна» використовується як елемент, що описує дані класу. Змінні розміщуються в домені даних і визначають стан об'єкта.

Під доменом методів розуміється простір розміщення методів, притаманних певному класу. Кількість доменів методів в одному класі не обмежена, як і кількість їх внутрішнього змісту. Необов'язково мають приховувати свій внутрішній зміст, користувач сам визначає, чи згортати/розгортати елемент.

Під доменом даних розуміється простір розміщення змінних (полів, атрибутів), притаманних певному класу. Кількість доменів даних в одному класі не обмежена, як і кількість їх внутрішнього змісту. Необов'язково мають приховувати свій внутрішній зміст, користувач сам визначає, чи згортати/розгортати елемент.

Під спільним доменом методів та даних розуміється єдиний простірв середині класу. Це стандартне представлення, що відображає всі елементи програмної логіки в одному вікні, забезпечуючи цілісне сприйняття структури класу.

Вплив когнітивної ергономіки на взаємодію під час моделювання архітектури наведено в табл. 3.3.

Таблиця 3.3

Вплив когнітивної ергономіки на взаємодію під час моделювання архітектури

Когнітивний фактор	Вплив на взаємодію з архітектурним представленням
Обмеження робочої пам'яті (4 об'єкти)	Рекомендоване відображення не більше 4 елементів одного типу одночасно; групування за кольором та формою об'єднує елементи в єдине ціле за принципами гештальт-психології
Швидкість обробки візуальної інформації (13-100 мс)	Тип елемента нотації та характер залежності розпізнаються миттєво без читання текстових підписів
Преатентивна обробка	Колір позначає тип елемента, розмір – його значущість, положення – належність до домену; мозок сприймає ці характеристики без свідомих зусиль
Закон близькості (Гештальт)	Класи, розташовані поруч у домені, сприймаються як логічно пов'язана група без явних з'єднувальних ліній
Зовнішнє когнітивне розвантаження	Діаграма архітектури слугує зовнішнім сховищем, інженер не утримує структуру пакетів у голові, а бачить її на екрані
Просторова пам'ять	Двовимірне розташування пакетів використовує навігаційні схеми мозку; інженер запам'ятовує «де» знаходиться клас, а не «як він називається»
Ієрархічне згортання	Просування вглиб виконується прокручуванням коліщатка миші; система запам'ятовує батьківський шлях для швидкого повернення
Обмеження кольорового кодування	Типи елементів архітектури (пакет, клас) розрізняються без перевантаження сприйняття
Drag-and-drop маніпуляції	Переміщення класу між пакетами виконується інтуїтивним перетягуванням замість текстових команд рефакторингу

Також може виникати питання при побудові ієрархії, який тип елемента використовувати, починати варто з «Пакетів», а «Домени пакетів» застосовуються лише тоді, коли потрібно групувати пакети через їх перевищення кількості. Рівень ієрархії всіх елементів нотації можна побачити в табл. 3.4. На практиці будується складніша ієрархічна структура, де кількість вкладених пакетів та їх доменів може вимірюватися десятками.

Таблиця 3.4

Зведена таблиця елементів нотації для моделювання архітектури

Поняття	Визначення	Рівень ієрархії
Корінь	Початковий контейнер проєкту, точка входу до архітектури	1 (найвищий)
Домен пакетів	Простір розміщення пакетів певного рівня	2
Пакет	Структурна одиниця групування класів та підпакетів (пакетів, доменів пакетів)	3
Домен класів	Простір розміщення групи класів	4
Клас	Узагальнений структурний елемент (клас/інтерфейс/перелік)	5
Спільний домен методів та даних	Єдиний простір методів і змінних класу	6
Домен методів	Простір розміщення методів класу	6 (альтернативний вигляд)
Домен даних	Простір розміщення змінних класу	6 (альтернативний вигляд)
Метод	Елемент опису операції/поведінки класу	7
Змінна	Елемент опису даних/стану класу	7
Програмна логіка методу	Візуальна модель алгоритму методу	8
Вкладення програмної логіки	Деталізовані рівні алгоритму	9+

Перебуваючи на найвищому рівні можна буде побачити тільки один елемент «Корінь», просуваючись вглиб на будь-якому рівні можна буде побачити не більше 4 структурних елементів кожного типу. В домені методів, даних та їх спільному домені обмеження щодо кількості елементів не застосовується.

Перехід від архітектурного представлення до програмної логіки виконується шляхом просування до найглибшого рівня «Корінь» → «Домен пакетів» → «Пакет» → «Домен класів» → «Клас» → Спільний домен методів та даних → «Програмна логіка методу» → «Наступні вкладення програмної логіки». Для зручності

просування в інтерфейсі користувача дану функцію можна реалізувати за допомогою прокручення коліщатка миші. Програма запам'ятовує останній активний елемент, і таким чином застосовує до нього просування вглиб. Для зміни активного елемента на моделі потрібно перемістити на нього вказівник миші. Просування вгору здійснюється автоматично, оскільки програма запам'ятовує батьківський елемент. Стандартним вікном вважається таким, що одночасно відображає всі елементи програмної логіки (методи, дані) в одному представлені.

У рамках реалізації прототипу передбачено механізм «CognitiveArchitectureRulesValidator», який при кожній зміні стану архітектури автоматично перевірятиме її відповідність когнітивним обмеженням. Якщо буде зафіксовано порушення правила, користувач отримає повідомлення з рекомендацією здійснити рефакторинг для відновлення когнітивної цілісності. У перспективі планується додати режим «strict cognitive rules», що унеможливить внесення змін, які порушують когнітивну ергономіку: будь-яке розширення архітектури вимагатиме негайного внесення коригувань і рефакторингу.

3.3 Проєктування представлення програмної логіки

Під представлення програмної логіки розуміють формальну візуальну модель, що описує алгоритмічну поведінку системи: послідовність операцій, умовні розгалуження, цикли, обробку даних та взаємодію між елементами на рівні виконуваного коду. Семантична повнота візуальних моделей досягається через формальну специфікацію метамоделей, які визначають правила побудови та інтерпретації діаграм. Метамодель описує доступні типи елементів, їх властивості, можливі зв'язки та обмеження, забезпечуючи консистентність та валідність створюваних моделей. Це створює основу для автоматизованої валідації, трансформації та генерації коду. Для кращого розуміння впливу принципів VPL [10] на представлення програмної логіки інформація про них згрупована в табл. 3.5.

Таблиця 3.5

Нюанси архітектури VPL для представлення програмної логіки

Нюанс	Суть
Абстракція над TPL	Парадигми, алгоритми та структури даних з текстових мов програмування використовуються з вищим рівнем абстракції (Принцип 1)
Візуальні елементи як документація	Оскільки немає коду, візуальні блоки є одночасно абстракцією виконуваної логіки та її документацією (Принцип 2)
Доменна ізоляція методів і даних	Методи розміщуються в домені методів, змінні – в домені даних, з можливістю спільного або роздільного перегляду (Принцип 6)
Поліморфізм методів	Не клас визначає реалізацію методу, а метод при виклику визначає поведінку залежно від типу об'єкта (Принцип 7)
Поліморфізм даних	Схема даних визначає обсяг, тип та обмеження змінної залежно від типу об'єкта, який до неї звертається (Принцип 8)
Деактивація елементів	Кожен елемент логіки може бути вимкнений без видалення для налагодження або заміни реалізацій (Принцип 12)
Незалежність генерації коду	ООП-абстракції можуть перетворюватися на процедурний або інший тип коду (Принцип 14)
Архітектура потоків	Домен потоків визначає, на якому потоці виконувати метод, з можливістю динамічної зміни залежно від умов

Представлення програмної логіки починається зі спільного домену методів та даних – єдиного простору, що відображає всі елементи класу (методи та змінні) в одному представленні, забезпечуючи цілісне сприйняття його структури. Альтернативно інженер може працювати з розділеними представленнями: Домен методів містить простір розміщення операцій, притаманних певному класу, тоді як Домен даних визначає простір розміщення змінних, що описують стан об'єкта. При переході до конкретного методу відкривається рівень програмної логіки методу – візуальна модель алгоритму, що описує послідовність операцій, умовні розгалуження, цикли та обробку даних. Програмна логіка може містити наступні вкладення – деталізовані рівні алгоритму, де кожен вкладений блок (цикл, умова, підпрограма) розкривається як окремий простір для моделювання, що дозволяє працювати лише з одним рівнем складності одночасно та знижує когнітивне навантаження на інженера.

Наведені принципи мають практичне застосування і впливають на архітектурне представлення наступним чином, див. табл. 3.6

Таблиця 3.6

Вплив нюансів VPL на представлення програмної логіки

Нюанс	**Практичний результат для інженера**
Абстракція над TPL	Інженер оперує візуальними блоками алгоритмів без написання синтаксично складного коду.
Візуальні елементи як документація	Відпадає потреба окремо документувати логіку – діаграма сама є актуальною документацією
Доменна ізоляція методів і даних	Можливість сфокусуватися лише на методах або лише на даних через окремі вікна перегляду
Поліморфізм методів	Одна візуальна модель методу автоматично адаптується до різних контекстів виклику
Поліморфізм даних	Схема даних налаштовується автоматично без дублювання визначень для різних типів
Деактивація елементів	Швидке тестування альтернативних реалізацій без видалення та відновлення блоків
Незалежність генерації коду	Одна модель може генерувати код для різних платформ або парадигм
Архітектура потоків	Візуальне призначення методів конкретним потокам (наприклад, GUI на головному, обчислення на фоновому); моделювання механізмів передачі даних між потоками та синхронізації без написання низькорівневого коду

Візуальна нотація базується на принципі трьохзонного горизонтального розподілу робочого простору. Ліва зона призначена для коментарів, де інженер може словесно описувати пункти алгоритму або окремі розділи логіки. Центральна зона містить основний потік програмної логіки, який називається центральною магістраллю. Права зона використовується для розміщення відгалужень логіки, зокрема вкладених структур циклів та хибних умов. Вертикальний потік визначає послідовність виконання операцій зверху вниз, завдяки чому більшість елементів не потребують стрілок для визначення напряму потоку, оскільки порядок виконання інтуїтивно зрозумілий із вертикального розміщення.

При проєктуванні програмної логіки спочатку виникає потреба описати сигнатуру методу. Елемент визначення повернення методу вказує, якого типу буде значення, що повертається після виконання методу. Цей елемент має форму сховища даних, виконаний у сірому кольорі та розташовується на початку моделі методу у верхньому лівому куті. Елемент визначення аргументів методу вказує, які аргументи приймає метод. Вигляд цього елемента нагадує масив у формі таблиці, де кожна комірка може містити визначення змінної із зазначенням її типу та імені. Колір цього

елемента також сірий. На рис. 3.2 наведено візуальну нотацію моделювання програмної логіки.

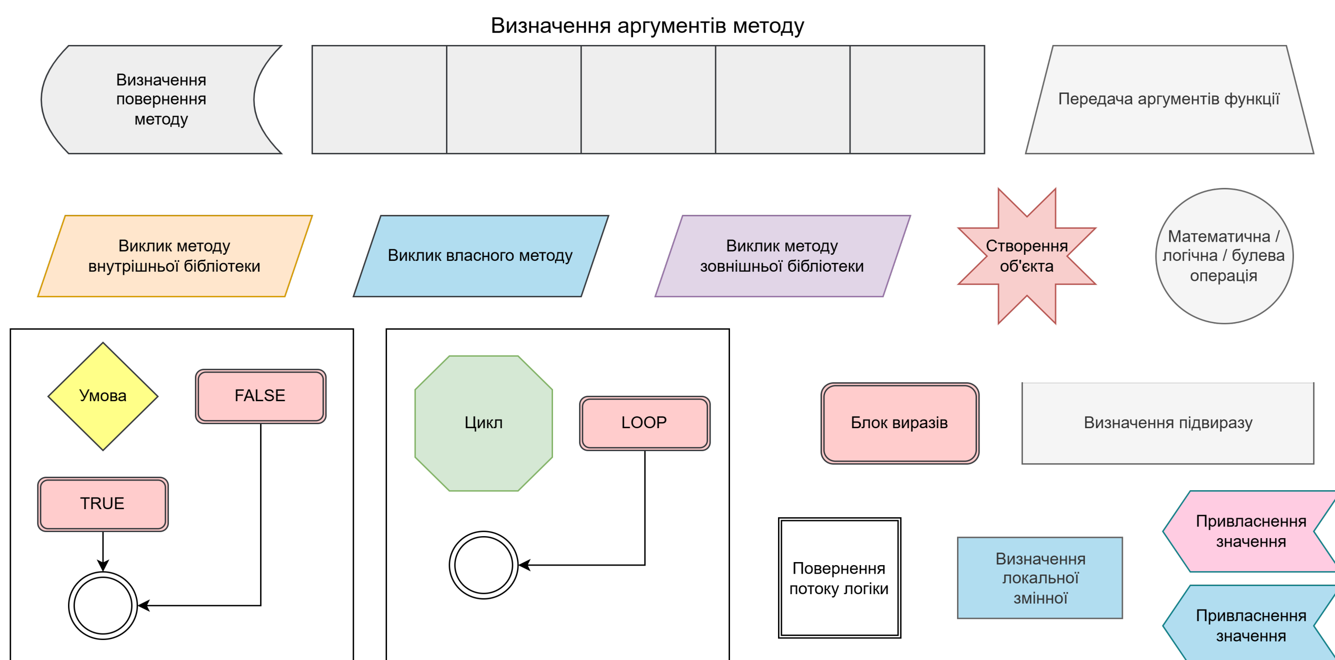

Рис. 3.2. Візуальна нотація інструменту елементів програмної логіки

Виклики методів поділяються на три типи, кожен з яких має унікальне візуальне оформлення у формі паралелограма. Виклик методу стандартної бібліотеки охоплює всі функції стандартної бібліотеки мови програмування, наприклад для Java це написання повідомлення в консоль, читання введення користувача або використання стандартних математичних функцій. Цей елемент виконаний у помаранчевому кольорі. Виклик власних методів передбачає звернення до методів з інших доменів методів або інших класів і позначається блакитним кольором. Виклик методу зовнішньої бібліотеки стосується залежностей, які власноруч імпортовано в проєкт, та має фіолетовий колір. Вибір конкретного методу для виклику здійснюється через графічний елемент із доступної множини, яка визначена заздалегідь. При виклику будь-якого методу за наявності в його сигнатурі аргументів праворуч від графічного елемента виклику з'являються комірки у формі трапеції сірого кольору, де кожна трапеція має бути заповнена відповідними даними.

Створення нового екземпляра об'єкта має власну нотацію у формі восьмикутної зірки червоного кольору, що забезпечує виразне виділення операції інстанціювання (створення конкретного екземпляра) серед іншого коду. Для визначення локальної змінної використовується окремий графічний елемент у формі прямокутника із гострими внутрішніми кутами блідо-блакитного кольору, який містить назву та тип змінної. Для привласнення значення змінній застосовується окремий графічний елемент у формі стрілкоподібного блоку, який існує у двох кольорових варіантах залежно від області видимості. Блідо-синій колір використовується для привласнення значення локальним змінним, тоді як блідо-червоний колір застосовується для привласнення значення властивостям класу.

Особливу увагу варто звернути на блоки умови та циклу, які мають власні графічні елементи, що легко помітити серед іншого коду. Блок умови має форму ромба жовтого кольору і візуально виділяється серед послідовного коду. Блок циклу представлений октагоном зеленого кольору, що дозволяє легко ідентифікувати його як ітераційну конструкцію. Цикли згідно з принципами візуальних мов програмування приховують свою внутрішню структуру в контейнері, в який можна поглибитися для перегляду вкладеної логіки.

Вкладена логіка на моделі відображається через графічний елемент чотирикутника з подвійними закругленими сторонами рожевого кольору. Такі контейнери мають три різновиди: LOOP для тіла циклу, TRUE для гілки істинної умови та FALSE для гілки хибної умови. Блок умови має два розгалуження, де TRUE продовжує основну логіку по центру, а FALSE відгалужується праворуч. Контейнери LOOP та FALSE завжди приховані та знаходяться правіше від центральної магістралі логіки. Контейнер TRUE знаходиться також по центру, але має одну відмінність: якщо кількість елементів коду не перевищує чотирьох, цей елемент може бути розгорнутий або згорнутий за бажанням інженера.

Будь-які цикли та розгалуження мають завершуватися елементом у формі кола з подвійною стороною, що свідчить про повернення логіки до головного потоку та об'єднання розгалуженої логіки з центральною магістраллю. У робочому середовищі вкладеної логіки будь-яке завершення потоку має супроводжуватися

спеціальним графічним елементом у формі квадрата з подвійною стороною. Цей елемент використовується як для стандартного виходу, коли закінчилася логіка, так і для операцій типу «return» або «break».

Стрілки в нотації присутні лише для повернення потоку із контейнерів LOOP, TRUE та FALSE. Це зроблено з метою попереднього перегляду вкладеної логіки та допомагає інженеру зрозуміти, де вкладена логіка об'єднується з головною. Вказівні стрілки допомагають визначити, де закінчується внутрішня логіка та де відбувається об'єднання з центральною магістраллю.

Операції над даними передбачають наявність певних виразів. Якщо вираз становить понад чотири елементи, його приховують в елемент блоку виразів, який має форму прямокутника з подвійними закругленими сторонами рожевого кольору. Вирази включають два важливі графічні елементи. Елемент у формі кола сірого кольору позначає математичну, логічну або булеву операцію. Елемент підвиразу у формі прямокутника без верхньої сторони сірого кольору допомагає візуально розділити частини складного виразу на менші частини.

Взаємодія з моделлю передбачає декілька способів навігації та перегляду. Прокручування колещатка миші на елементах LOOP, TRUE або FALSE здійснює перехід до вкладеного простору моделювання. Наведення вказівника миші на контейнер активує попередній перегляд перших чотирьох рядків вкладеного коду. Згортання та розгортання контейнера TRUE, коли він містить не більше чотирьох елементів, виконується одним клацанням миші. Додавання коментаря розміщує його ліворуч від відповідного блоку логіки.

Така організація простору моделювання програмної логіки забезпечує оптимальне використання як вертикальної площини для потоку логіки, так і горизонтальної площини для структурного розділення на коментарі, логіку та відгалуження. Візуальна нотація переслідує ціль розбити складну логіку на прості частини, легкі для перегляду та аналізу.

Вплив когнітивної ергономіки на взаємодію під час моделювання програмної логіки наведено в табл. 3.7.

Таблиця 3.7

Вплив когнітивної ергономіки на взаємодію під час моделювання програмної логіки

Когнітивний фактор	**Вплив на взаємодію з представленням програмної логіки**
Обмеження робочої пам'яті	Відображення лише одного рівня вкладення алгоритму; вміст циклу або умови розкривається окремо, а не одночасно з батьківським контекстом
Швидкість обробки візуальної інформації (13-100 мс)	Тип блоку (змінна, умова, цикл, виклик методу) розпізнається за формою миттєво; текстовий код потребував би сотень мілісекунд на читання ключових слів
Преатентивна обробка	Колір блоку сигналізує про його категорію (потік даних, змінна, виклик); розмір показує складність; положення – порядок виконання
Закон спільної долі (Гештальт)	Блоки, що виконуються послідовно, візуально вирівняні та сприймаються як єдиний потік без додаткових позначень
Зовнішнє когнітивне розвантаження	Алгоритм візуалізований на діаграмі – інженер аналізує логіку очима, а не відтворює її подумки з текстового коду
Послідовне просування вглиб	Подвійний клік або прокручування відкриває вміст блоку; система зберігає контекст для повернення на рівень вище
Відсутність синтаксичних помилок	Візуальне з'єднання блоків унеможливлює пропуск дужки, крапки з комою або неправильний відступ
Розділення вікон перегляду	Окреме вікно для методів, окреме для даних знижує одночасне навантаження; стандартне вікно показує обидва домени для цілісного сприйняття
Автоматичне приховування деталей	Другорядні параметри методу або типи змінних з'являються лише при наведенні курсора, не засмічуючи основний вигляд

Варто зазначити, що не всі принципи візуальних мов програмування [10] описані в даній науковій роботі, а також не всі будуть реалізовані для прототипу.

Для прототипу цільовою мовою трансляції обрано Java, що забезпечує достатній рівень функціональності та практичної перевірки парадигми. Водночас це рішення не обмежує майбутній розвиток: архітектура інструменту передбачає можливість розширення списку підтримуваних мов програмування. Таким чином, Java виступає стартовою платформою для демонстрації працездатності, а у перспективі може бути доповнена іншими мовами, наприклад, C#, Python чи JavaScript – без зміни базових принципів реалізації.

Отже, здійснено проєктування прототипу візуального інструменту архітектурного моделювання програмного забезпечення, що складається з двох взаємопов'язаних представлень: архітектурного та програмної логіки. Розроблена візуальна нотація архітектури включає 9 типів елементів з ієрархією рівнів.

3.4 Висновки до розділу

У третьому розділі систематизовано 14 виявлених проблем сучасних інструментів архітектурного моделювання та сформульовано відповідні рекомендації. На цій основі запропоновано нову парадигму «Когнітивно-ергономічна холістична візуальна парадигма розробки» (CEHVDP), що інтегрує чотири взаємопов'язані принципи: когнітивно-ергономічний дизайн середовища, холістичну тришарову метамодель з валідацією повноти, візуальне WYSIWYG-редагування та візуальну мову програмування для повної генерації коду.

На базі сформульованих принципів визначено завдання проєктування прототипу інструменту візуального модельно-орієнтованого інжинірингу з підтримкою архітектурного моделювання та генерації Java-коду для верифікації запропонованої концепції.

Здійснено проєктування для двох взаємопов'язаних представлень: архітектурного та програмної логіки. Візуальна нотація архітектури включає 9 типів елементів з ієрархією рівнів вкладеності для поступового уточнення архітектурних рішень. Для програмної логіки розроблено нотацію основних алгоритмічних конструкцій з унікальними формами та колірним кодуванням. Спроєктовано механізм вкладень для роботи з одним рівнем складності одночасно та інтуїтивну навігацію між рівнями вкладеності.

Обґрунтовано застосування принципів візуальних мов програмування для обох представлень системи. Визначено вплив когнітивних факторів на взаємодію з інструментом, зокрема обмеження робочої пам'яті до 4 елементів одного типу та принципи гештальт-психології. Спроєктовано механізм «Cognitive Architecture Rules Validator» для автоматичної валідації відповідності архітектури когнітивним обмеженням.

У наступному розділі наведено результати реалізації прототипу та експериментальних досліджень для оцінки його ефективності.

РОЗДІЛ 4
РЕАЛІЗАЦІЯ І ТЕСТУВАННЯ ІНСТРУМЕНТУ ВІЗУАЛЬНОГО МОДЕЛЬНО-ОРІЄНТОВАНОГО ІНЖИНІРИНГУ

4.1 Технічна реалізація архітектури та механізмів інструменту

Для розробки інструменту архітектурного моделювання програмного забезпечення було обрано сучасний технологічний стек на базі екосистеми JVM (віртуальна машина Java). Основною мовою реалізації виступає Kotlin – сучасна мова програмування, що поєднує об'єктно-орієнтовану та функціональну парадигми, забезпечує високу продуктивність, безпеку типів та лаконічний синтаксис. Для побудови графічного інтерфейсу використовується Jetpack Compose Desktop – декларативний UI-фреймворк від Google, який дозволяє створювати інтуїтивні та адаптивні інтерфейси з мінімальною кількістю коду завдяки reactive підходу до управління станом.

Візуалізація архітектурних діаграм здійснюється через компонент Canvas, який надає низькорівневий API для малювання графічних елементів. Всі компоненти моделі, включаючи класи, пакети та інші архітектурні елементи, рендеряться за допомогою базових геометричних примітивів: прямокутників, ліній, еліпсів, полігонів та кривих Безьє. Такий підхід забезпечує повний контроль над візуальним представленням та високу продуктивність рендерингу.

Цільовою мовою для автоматичної генерації програмного коду з архітектурних моделей обрано Java – промислову об'єктно-орієнтовану мову з широкою підтримкою у корпоративному сегменті та великою екосистемою бібліотек. Інструмент орієнтовано на роботу в середовищі операційної системи Windows 64-bit, що охоплює переважну більшість корпоративних робочих станцій та забезпечує максимальну сумісність з наявною інфраструктурою підприємств.

Архітектура прототипу під назвою «BioFusion IDE» реалізована як модульна система з чітким розділенням відповідальності між презентацією, бізнес-логікою,

управлінням станом та моделлю даних. Загальну діаграму компонентів архітектури інструменту можна побачити на рис. 4.1.

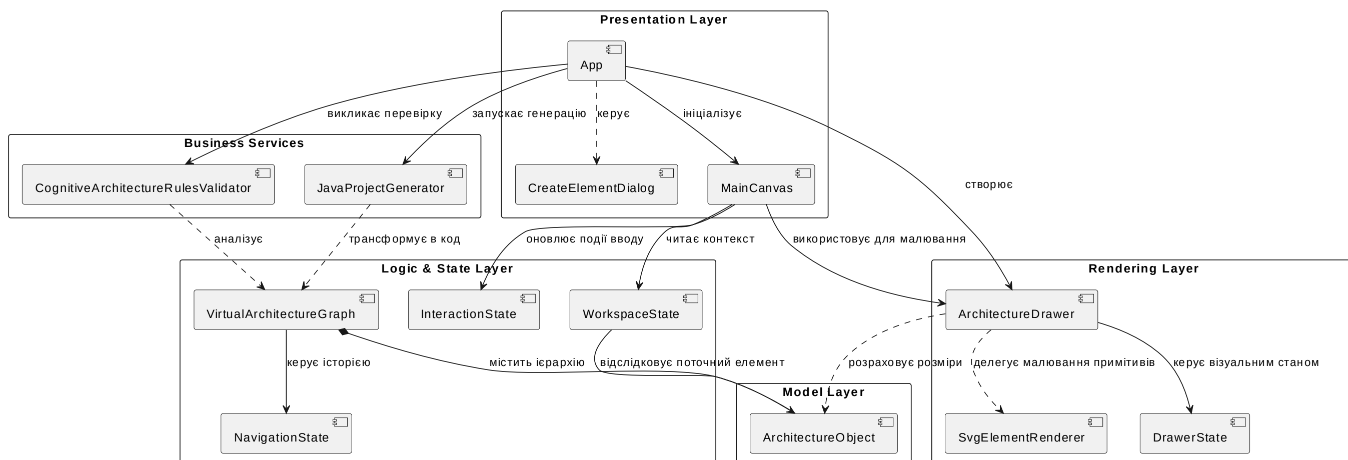

Рис. 4.1. Компонентна діаграма архітектури інструменту

Шар презентації очолює компонент App, який ініціалізує глобальні стани та координує відображення інтерфейсу. Він містить MainCanvas, що слугує візуальним контейнером для архітектурної моделі. Цей шар динамічно реагує на зміни в бізнес-логіці, використовуючи Jetpack Compose для реактивного оновлення UI при зміні станів діалогів або результатів валідації.

Шар управління станом та навігації базується на взаємодії компонентів WorkspaceState та VirtualArchitectureGraph. VirtualArchitectureGraph утримує деревоподібну структуру проєкту, де кожен вузол є спадкоємцем ArchitectureObject. WorkspaceState виступає посередником, контролюючи «глибину» перегляду: він визначає, який контейнер наразі розгорнутий для редагування (root, package або class), та відфільтровує елементи для відображення. NavigationState, вбудований у граф, забезпечує історію переходів (через стек навігації), дозволяючи користувачеві заглиблюватися в домени та повертатися назад.

Шар рендерингу та візуалізації зосереджений у класі ArchitectureDrawer та допоміжному SvgElementRenderer. ArchitectureDrawer не просто малює, а й виконує складні розрахунки компонування розмітки: для звичайних елементів використовується потокове розміщення з перенесенням рядків, тоді як для доменів даних та методів реалізовано вкладене відображення (діти малюються всередині

батьківського контейнера). Цей шар тісно пов'язаний зі станом DrawerState, який синхронізує анімації та візуальні ефекти перетягування.

Шар взаємодії відокремлено від малювання через InteractionState та спеціалізовані обробники: DragHandler, TapHandler та ScrollHandler. Вони перехоплюють події вводу на Canvas, модифікують проміжний стан взаємодії та передають команди до WorkspaceState або ArchitectureDrawer для оновлення позицій елементів чи зміни навігаційного контексту. Це дозволяє реалізувати плавний «drag-and-drop» та масштабування без блокування основного потоку UI.

Шар бізнес-логіки та генерації включає спеціалізовані сервіси. CognitiveArchitectureRulesValidator реалізує алгоритми перевірки структурних обмежень, аналізуючи граф на предмет перевантаження елементами одного типу. JavaProjectGenerator відповідає за трансляцію абстрактної моделі VirtualArchitectureGraph у реальну файлову структуру Java-проєкту, перетворюючи архітектурні об'єкти (Class, Variable, Method) у відповідні синтаксичні конструкції мови програмування.

Механізм "Перевірка когнітивних обмежень архітектури". Алгоритм перевірки когнітивних обмежень працює в класі CognitiveArchitectureRulesValidator. Його завдання пройти всю структуру проєкту та знайти місця, де на одному рівні зібралося занадто багато однакових елементів. Якщо їх більше ніж 4, це вважається проблемою для сприйняття.

Перевірка починається з методу validate. Він створює список для помилок і запускає рекурсивний метод validateElement для кореневого елемента. Цей метод має прапорець isClassChild, який на початку дорівнює false. Він визначає, чи користувач перебуває всередині класу, чи ні. Далі йде двоетапна логіка. Спершу перевіряється поточний рівень. Якщо isClassChild = false, алгоритм рахує дітей елемента, групуючи їх за типом. Якщо якогось типу більше ніж 4, створюється запис про помилку. Якщо ж isClassChild = true, перевірка пропускається – це дозволяє класам мати скільки завгодно методів і змінних.

Другий етап є рекурсією. Перед тим як перейти до дітей, алгоритм визначає новий контекст: змінна childIsClassChild стає true, якщо поточний елемент є класом

(ElementType.CLASS). Потім метод викликається для кожного дочірнього елемента з цим новим контекстом. Візуалізацію алгоритму можна побачити на діаграмі послідовності (рис. 4.2).

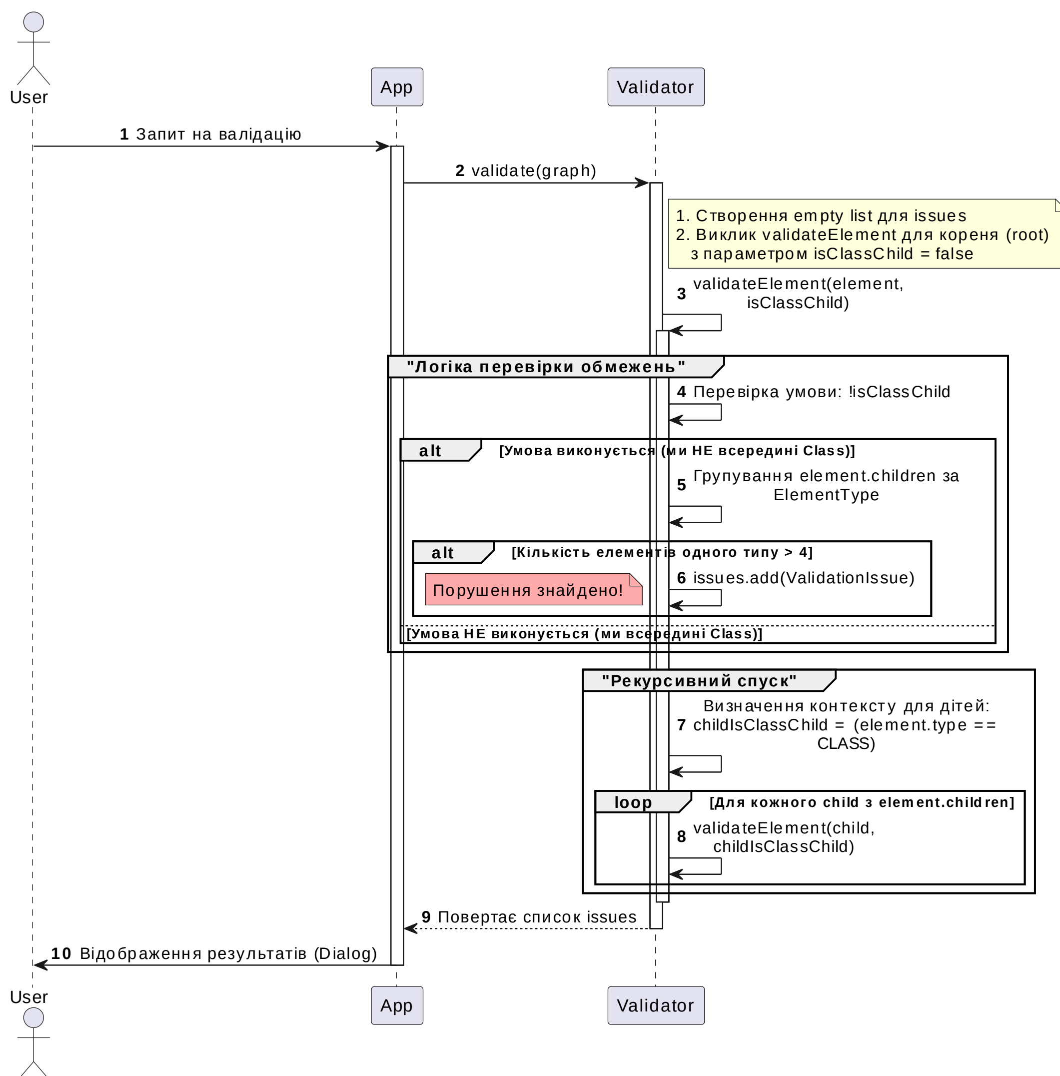

Рис. 4.2. Діаграма послідовності механізму валідації когнітивних правил моделі

Таким чином, як тільки алгоритм заходить у клас, усі вкладені рівні отримують «імунітет» від перевірки. Це означає, що всередині класів не виникають помилки навіть при великій кількості методів чи змінних.

Архітектурний граф VirtualArchitectureGraph є центральною структурою даних системи, що представляє ієрархічну модель програмного забезпечення у вигляді дерева. Кореневий елемент RootObject містить вкладені пакети, класи, методи та змінні. Граф підтримує навігацію вглиб структури, серіалізацію у JSON

для збереження та відновлення, а також операції додавання, видалення та переміщення елементів із автоматичним сортуванням.

Механізм "Генерація змодельованої структури". Алгоритм генерації коду в класі JavaProjectGenerator перетворює візуальну модель у справжню файлову структуру Java-проєкту. Початок роботи відбувається в методі generateProject. Він очищає директорію build: видаляє старі файли та створює її заново, щоб уникнути конфліктів. Далі запускається рекурсивний метод processElement, який проходить дерево елементів у глибину. Його поведінка залежить від типу вузла. Якщо це контейнер (наприклад, PACKAGE чи CLASS_DOMAIN), він створює відповідну папку в системі та формує рядок шляху пакета (наприклад, com.app.logic). Цей шлях передається далі при обході. Візуалізацію алгоритму можна побачити на діаграмі діяльності (рис. 4.3).

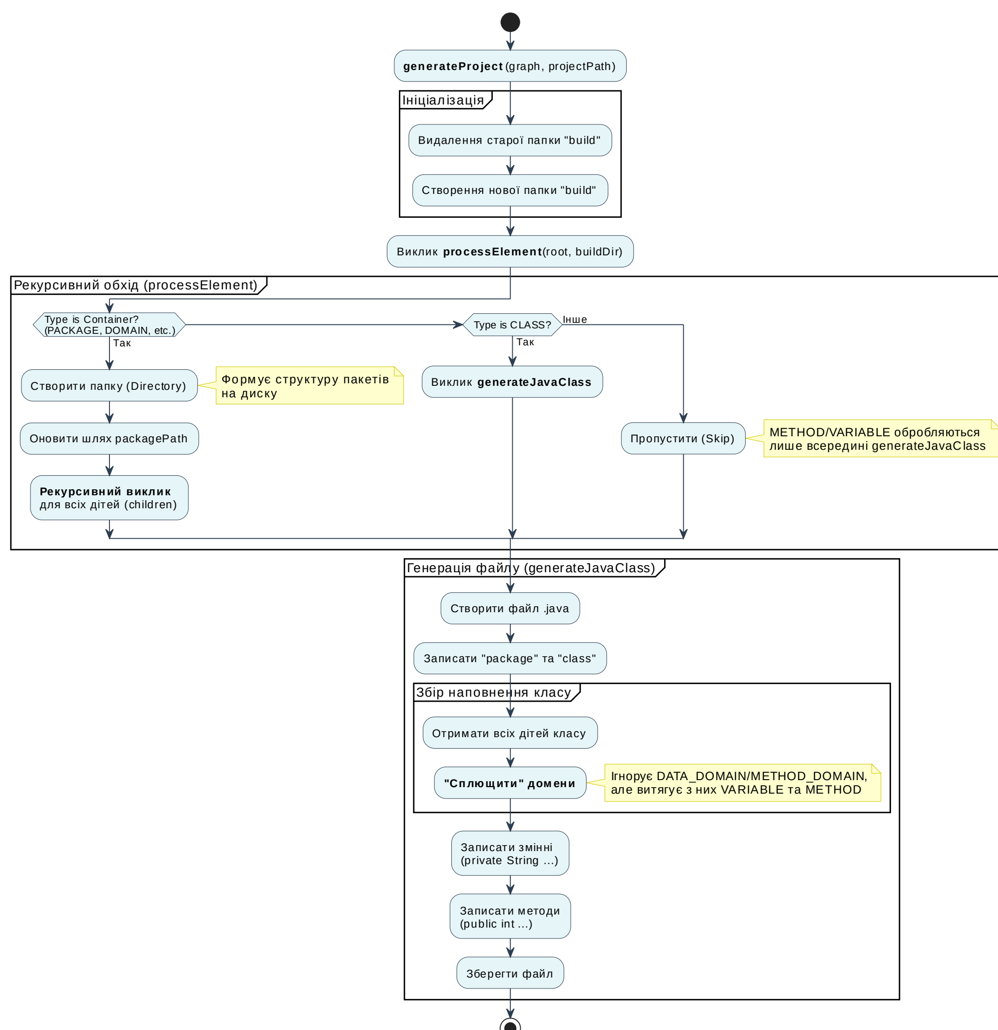

Рис. 4.3. Діаграма діяльності механізму генерації змодельованої архітектури

Коли алгоритм доходить до елемента CLASS, він зупиняє створення папок і переходить до генерації файлу класу через метод generateJavaClass. Саме тут відбувається аналіз вмісту класу: алгоритм переглядає дочірні елементи, пропускаючи структурні оболонки (DATA_DOMAIN, METHOD_DOMAIN), але витягує змінні та методи. Змінні записуються як поля класу, а методи створюються як шаблони згідно синтаксису мови програмування Java. Таким чином, складна візуальна структура з доменами перетворюється у зрозумілий Java-клас із полями та методами, готовий до подальшого опрацювання в проєкті.

4.2 Експериментальна оцінка та приклади використання інструменту

Прототип, створений у межах даної роботи, слугує інструментом верифікації концептуальних рішень та формує емпіричну основу для подальшої стандартизації. На рис. 4.4 можна побачити екран привітання інструменту та меню створення проєкту, користувачу пропонується вказати: ім’я проєкту, вибрати директорію збереження, опис та наразі доступну тільки одну мову програмування Java.

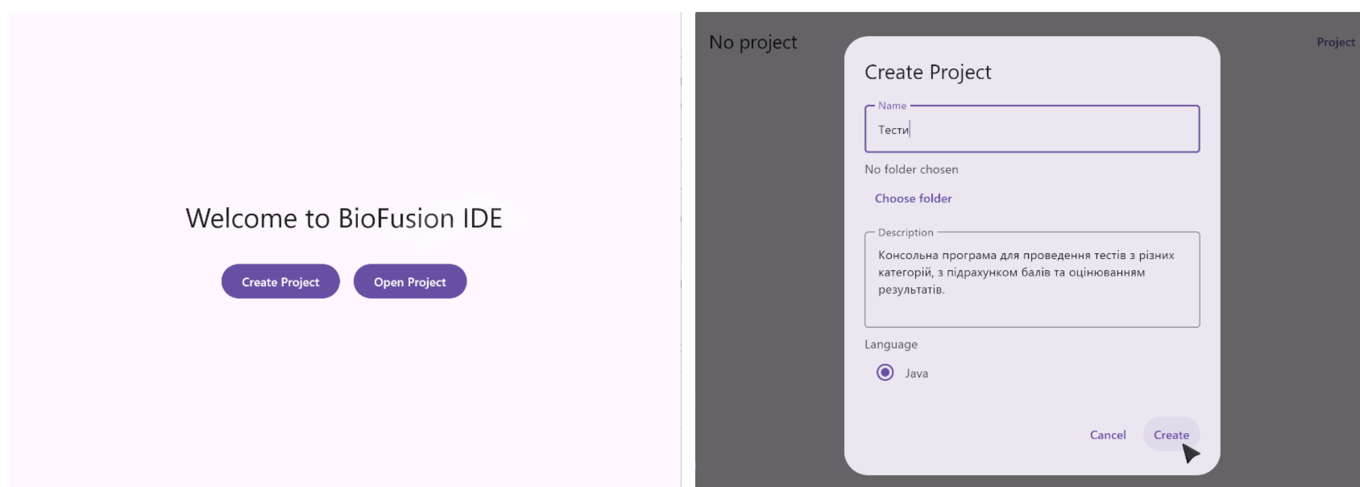

Рис. 4.4. Вигляд інтерфейсу інструменту «BioFusion IDE»

Для демонстраційного моделювання архітектури було обрано тему: «Інтерактивна вікторина з математики, історії та логіки», – консольна програма для проведення тестів з різних категорій, з підрахунком балів та оцінюванням

результатів. Кореневий пакет «quiz» містить точку входу «Main.java» (ініціалізація програми та головний цикл) і чотири основні пакети. Пакет доменів «core» об'єднує підпакети: «engine» із класом «QuizEngine» (координація проведення вікторин), «scoring» із класом «ScoreManager» (розрахунок балів та оцінок), «session» із класом «GameSession» (керування ігровими сесіями). Пакет доменів «content» містить підпакети: «math» із класом «MathQuiz» (математичні питання), «history» із класом «HistoryQuiz» (історичні питання), «logic» із класом «LogicQuiz» (логічні питання), а також вкладений пакет доменів «data» із класами «QuestionData» (метадані питань), «AnswerKey» (зберігання правильних відповідей), «DifficultyConfig» (налаштування складності), «CategoryInfo» (інформація про категорії). Пакет «model» із доменами класів включає: «base» із класом «Question» (модель питання), «result» із класом «Result» (результат вікторини), «player» із класом «Player» (дані гравця). Пакет «ui» із доменами класів містить: «input» із класом «InputHandler» (зчитування введення), «output» із класом «OutputHandler» (виведення повідомлень), «menu» із класом «MenuManager» (відображення меню).

Будь-який процес моделювання починається із заглиблення в елемент «Root» та відкриття діалогу створення нових елементів (рис. 4.5).

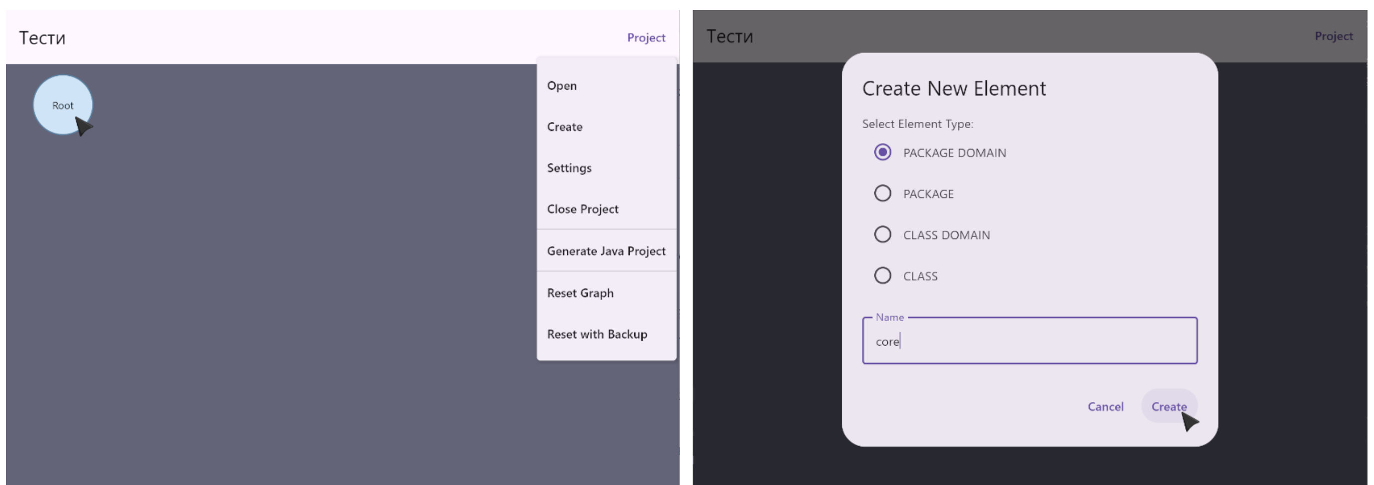

Рис. 4.5. Діалог створення нових елементів

В інструменті для зручності реалізовані також функції «Відкрити проєкт», «Відкрити налаштування» - де можна змінити назву та опис проєкту, «Закрити проєкт». Функції «Скинути граф» та «Скинути граф зі збереженням резервної копії»

очищають поточну модель проєкту, дозволяюся перемалювати весь проєкт заново. Результат моделювання першого рівня вкладеності можна побачити рис. 4.6.

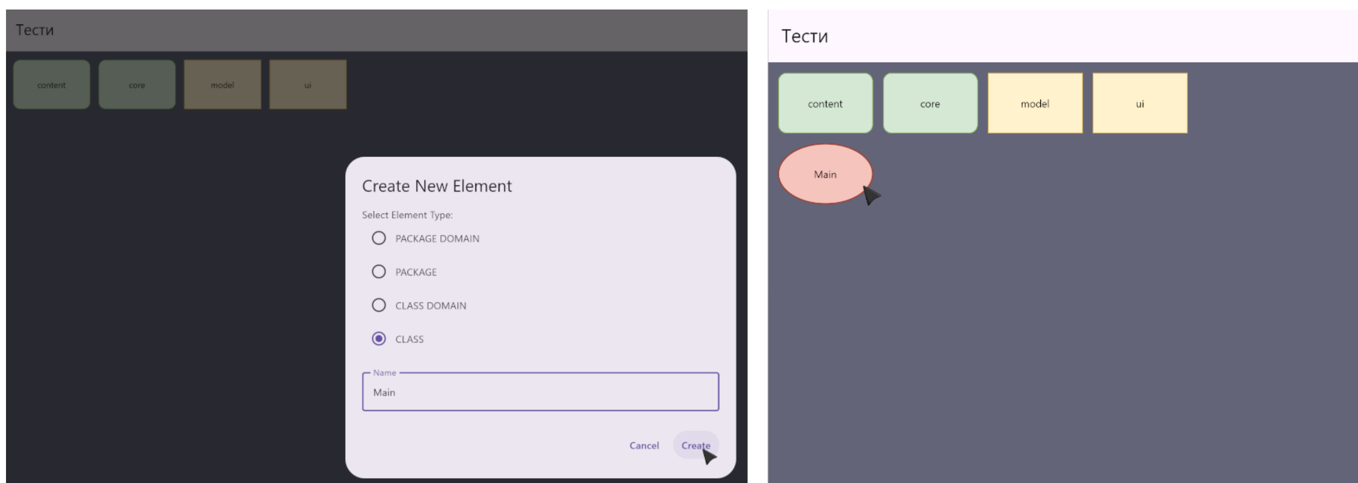

Рис. 4.6. Результат моделювання архітектури першого рівня

Здійснено моделювання подальших рівнів, на рис. 4.7 показано приклад порушення когнітивних правил за маршрутом «content/data/», коли кількість елементів однакового типу перевищує встановлений ліміт, користувач отримує повідомлення згенероване механізмом перевірки когнітивних обмежень архітектури.

Рис. 4.7. Діалог порушення когнітивних правил

Механізм «Генерація змодельованої структури» запускається через кнопку із меню, що знаходиться в правому верхньому куті. Дія виконується один кроком, після чого користувач отримує сповіщення про стан виконання операції. Для перегляду

дерева згенерованої структури проєкту використано стандартну консольну команду «tree /F» (рис. 4.8).

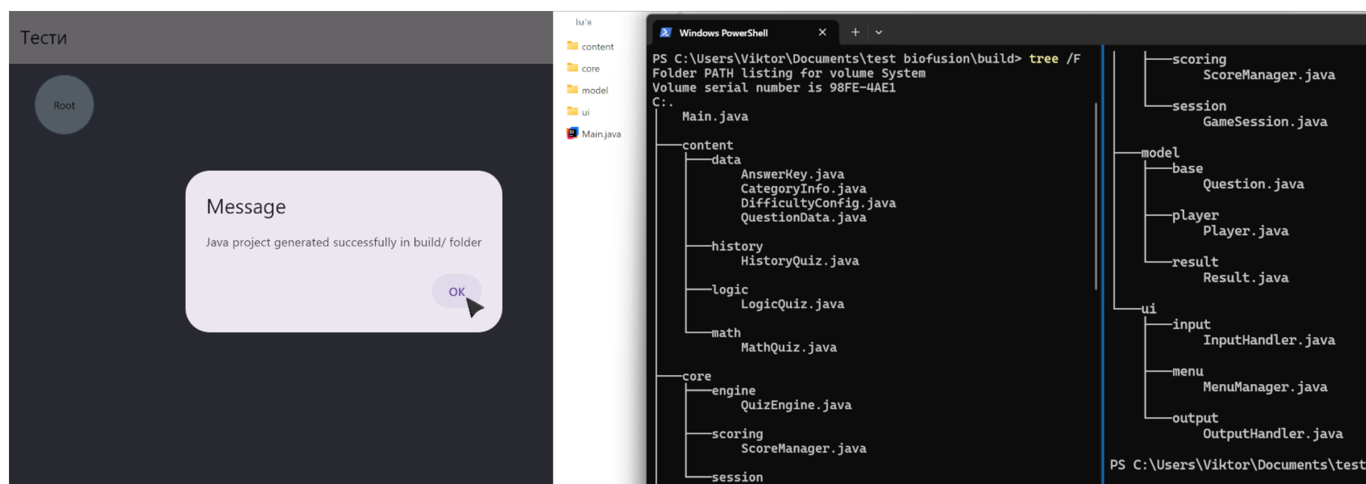

Рис. 4.8. Результат механізму генерації змодельованої архітектури

На рис. 4.9 показано приклад решти елементів графічної нотації архітектури програмного забезпечення, серед них: домени класів, домени даних та змінні, домени методів та методи.

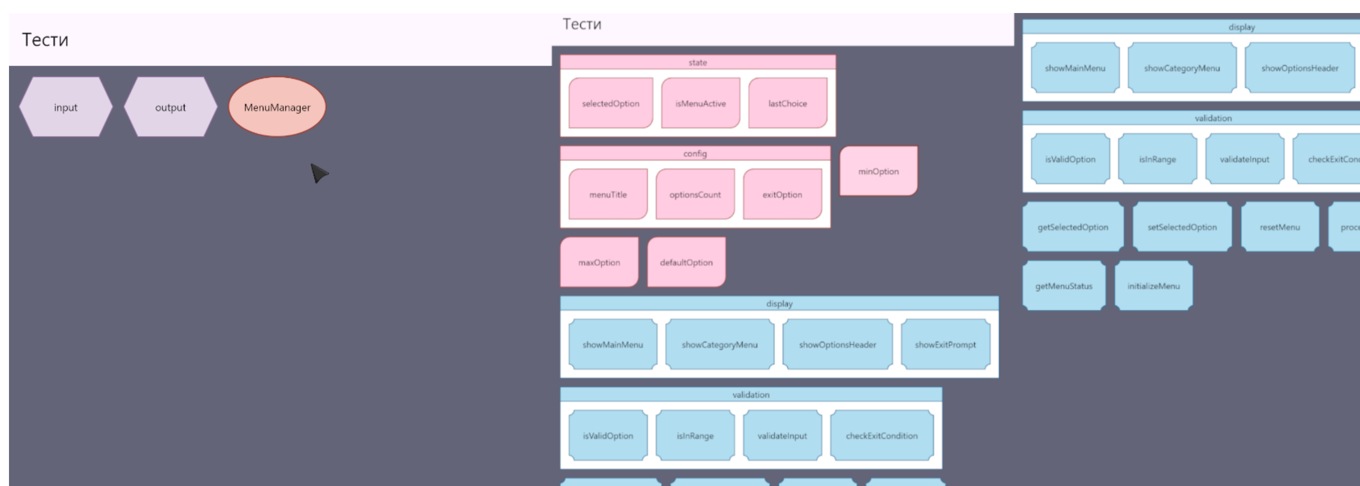

Рис. 4.9. Приклад архітектурного моделювання нижчих рівнів

Механізм генерації програмного коду потребує повної імплементації візуальної нотації для моделювання програмної логіки. Запропонована нотація ще потребує значного покращення, щоб повністю відповідати синтаксису Java. Проте навіть часткова реалізація здатна приносити користь, забезпечуючи генерацію

шаблонів класів, які інженери зможуть розвивати далі. Приклад генерації шаблонного коду зі структурних елементів архітектури (домени даних та методів, змінні, методи) наведений на рис. 4.10.

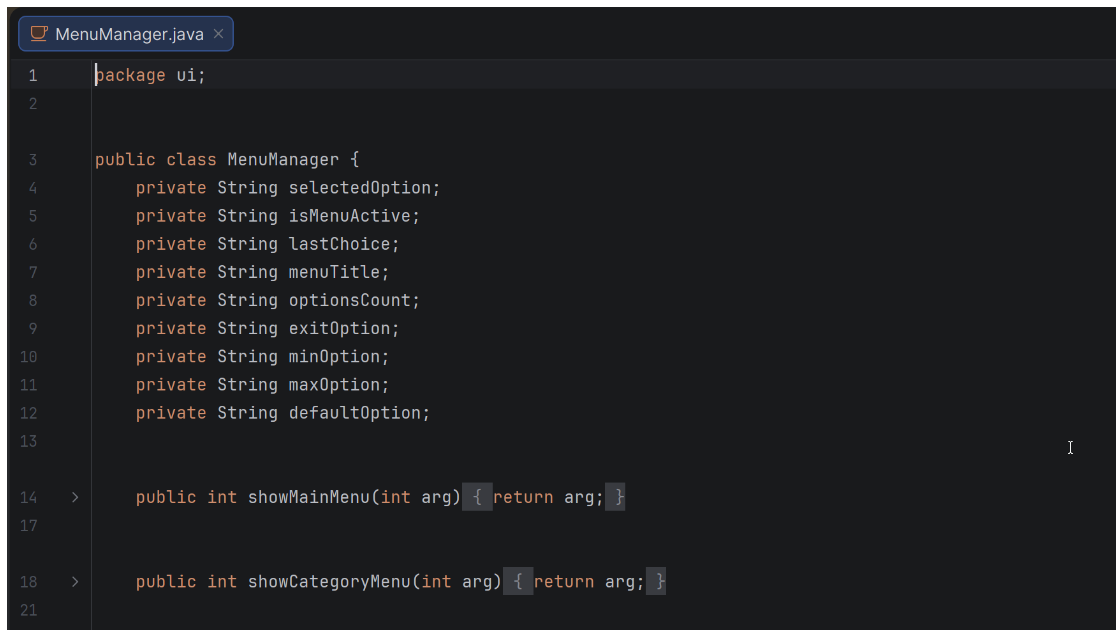

Рис. 4.10. Приклад генерації частково коду для класу

Експериментальна оцінка розробленого прототипу «BioFusion IDE» проводилася з метою верифікації концептуальних рішень, закладених у візуальну нотацію архітектурного моделювання, та формування емпіричної основи для подальшого розвитку інструменту. Дослідження охоплювало аналіз переваг та недоліків запропонованого методу, оцінку практичної застосовності механізмів валідації когнітивних обмежень та генерації коду, а також визначення перспектив розвитку інструменту.

Запропонований метод демонструє суттєві переваги. По-перше, зниження когнітивного навантаження завдяки обмеженню відображення до чотирьох елементів одного типу на рівні. По-друге, інтуїтивність навігації через графову структуру пакетів та механізм послідовного заглиблення з автоматичним збереженням батьківського контексту. По-третє, автоматизація рутинних операцій, дегенерація файлової структури Java-проєкту робиться одним кроком. По-четверте, візуальна валідація архітектурних рішень у реальному часі з негайним зворотним зв'язком про проблемні місця.

Водночас виявлено недоліки поточної реалізації: неповна реалізація візуальної нотації для моделювання програмної логіки, що обмежує генерацію коду шаблонними структурами; підтримка лише мови Java; відсутність механізмів колаборативної роботи та інтеграції з системами контролю версій; обмежена масштабованість для великих проєктів із сотнями класів.

Демонстраційне моделювання підтвердило працездатність основних механізмів. Процес від створення кореневого пакету до генерації файлової структури виконується інтуїтивно. Механізм валідації коректно ідентифікує порушення когнітивних обмежень та надає рекомендації щодо рефакторингу. Генерація файлової структури продемонструвала коректне перетворення ієрархії візуальних елементів на директорії та файли Java.

Прототип має значні перспективи для повної імплементації візуальної нотації Java. Першочерговим є завершення представлення програмної логіки для повного циклу модельно-орієнтованої розробки. Другим напрямком є розширення підтримки мов (Kotlin, TypeScript, Python). Важливою є інтеграція з екосистемами розробки: видобування даних з готових проєктів, підключення збірників проєктів та систем контролю версій. Планується розвиток режиму «strict cognitive rules» для примусового дотримання архітектурних стандартів. Ще одним напрямок є реалізація рівня бізнес-моделювання з візуальним представленням акторів, варіантів використання та прецедентів. Це дозволить описувати вимоги до системи безпосередньо в інструменті та інтегрувати їх в архітектурну модель. Варта також провадження функція візуального опису історій користувачів (англ. user story telling) для наративного опису сценаріїв взаємодії користувача з системою. Такий підхід забезпечить прямий зв’язок між бізнес-вимогами та архітектурними рішеннями.

Порівняльний аналіз з інструментами Enterprise Architect, Visual Paradigm, PlantUML на поточному етапі є передчасним. Прототип реалізує принципово відмінний підхід, заснований на когнітивній ергономіці, що ускладнює пряме зіставлення. Коректний аналіз вимагатиме повної реалізації функціоналу та контрольованих експериментів. Наразі прототип слугує доказом концепції, що підтверджує життєздатність підходу.

4.3 Висновки до розділу

У четвертому розділі здійснено технічну реалізацію прототипу інструменту візуального модельно-орієнтованого інжинірингу «BioFusion IDE» на основі сформованої концепції. Для розробки обрано технологічний стек на базі екосистеми JVM: мову Kotlin, фреймворк Jetpack Compose Desktop для побудови графічного інтерфейсу та компонент Canvas для візуалізації архітектурних діаграм. Реалізовано модульну архітектуру системи з чітким розділенням відповідальності між шарами презентації, управління станом, рендерингу, взаємодії та бізнес-логіки.

Реалізовано два ключові механізми інструменту. Механізм перевірки когнітивних обмежень архітектури «CognitiveArchitectureRulesValidator» автоматично аналізує структуру проєкту та ідентифікує порушення правила чотирьох елементів одного типу на рівні. Механізм генерації змодельованої структури «JavaProjectGenerator» перетворює візуальну модель на реальну файлову структуру Java-проєкту з автоматичним створенням директорій та шаблонних класів.

Проведено експериментальну оцінку реалізованого рішення на прикладі моделювання консольної програми «Інтерактивна вікторина з математики, історії та логіки». Дослідження підтвердило працездатність основних механізмів та виявило переваги методу: зниження когнітивного навантаження, інтуїтивність навігації, автоматизацію рутинних операцій та візуальну валідацію архітектурних рішень у реальному часі. Водночас визначено обмеження поточної реалізації: неповна підтримка представлення програмної логіки, орієнтація лише на мову Java та відсутність механізмів колаборативної роботи.

Аналіз та узагальнення результатів досліджень дозволили сформувати перспективні напрямки розвитку інструменту, зокрема завершення представлення програмної логіки, розширення підтримки мов програмування, інтеграцію з екосистемами розробки та реалізацію рівня бізнес-моделювання. Застосування принципів візуальних мов програмування забезпечує архітектурну гнучкість та еволюційну стійкість програмних систем, що підтверджує життєздатність запропонованого підходу.

ВИСНОВКИ

У магістерській роботі розв'язано актуальну наукову задачу розробки методів візуального модельно-орієнтованого інжинірингу для архітектурного моделювання програмного забезпечення, що поєднують принципи модельно-орієнтованої інженерії та візуальних мов програмування з урахуванням когнітивних обмежень розробників. Значення отриманих результатів для науки полягає у формуванні теоретичного підґрунтя для створення нового покоління інструментів архітектурного проєктування, а для практики у розробці прототипу, що демонструє можливість поєднання візуального редагування з автоматичною генерацією структури та коду.

Оцінка сучасного стану проблеми базується на комплексному аналізі 26 інструментів архітектурного моделювання за 12 категоріями. На основі аналізу ідентифіковано 14 ключових проблем та сформульовано відповідні рекомендації щодо їх розв'язання.

Методом розв’язання зазначених проблем обрано інтеграційний підхід, що поєднує сильні сторони модельно-орієнтованої інженерії та візуальних мов програмування. Запропоновано нову парадигму «Когнітивно-ергономічна холістична візуальна парадигма розробки» (CEHVDP), яка ґрунтується на чотирьох взаємопов'язаних принципах: когнітивно-ергономічний дизайн середовища розробки, холістична тришарова метамодель з валідацією повноти, візуальне WYSIWYG-редагування як основний режим роботи та візуальна мова програмування для повної генерації коду. Парадигма інтегрує когнітивну ергономіку, холістичне моделювання та візуальні мови програмування, забезпечуючи природний когнітивний шлях від абстрактного до конкретного.

Порівняно з відомими розв'язаннями, запропонований підхід адресує системну прогалину, яку не вирішує жоден із розглянутих 26 інструментів: одночасне поєднання візуального редагування з низьким когнітивним навантаженням, формальної метамоделі для MDE-сумісності, можливості генерації виконуваних артефактів та врахування когнітивних обмежень людського сприйняття. На відміну від класичних CASE-інструментів, що зосереджуються на

документуванні, та low-code платформ, що нехтують глибоким архітектурним моделюванням, запропонований підхід забезпечує контрольовану архітектуру з ефективною автоматизованою реалізацією.

Якісні результати дослідження включають розробку візуальної нотації для архітектурного моделювання з 9 типами елементів та ієрархією рівнів вкладеності, а також візуальної нотації для програмної логіки, що охоплює основні алгоритмічні конструкції. Визначено вплив когнітивних факторів на взаємодію під час моделювання, обґрунтовано застосування принципів преатентивної обробки та гештальт-психології. Кількісні результати підтверджують ефективність підходу: обмеження відображення до 4 елементів одного типу на рівні відповідає науково встановленим межам робочої пам'яті людини.

Правдивість результатів забезпечується систематичним порівняльним аналізом інструментів, що були розглянуті, за уніфікованими критеріями. Відповідністю розроблених рішень фундаментальним дослідженням когнітивної психології та практичною верифікацією через реалізацію прототипу «BioFusion IDE» версії 1.0.0. Експериментальна оцінка на прикладі моделювання консольної програми підтвердила працездатність запропонованого підходу, механізмів валідації когнітивних обмежень та генерації структури Java-проєктів.

Рекомендації щодо використання результатів охоплюють три напрямки. Для наукових досліджень запропоновані принципи парадигми CEHVDP можуть слугувати теоретичною основою для подальшого розвитку методів візуального модельно-орієнтованого інжинірингу. Для практичного застосування сформульовані методичні рекомендації щодо когнітивно-ергономічного проєктування інструментів архітектурного моделювання, включаючи мінімалістичний інтерфейс, обмеження одночасного відображення елементів та ієрархічну декомпозицію. Для подальшого розвитку визначено перспективні напрямки: завершення представлення програмної логіки, розширення підтримки мов програмування, інтеграція з екосистемами розробки та реалізація рівня бізнес-моделювання з візуальним представленням акторів та прецедентів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cowan N. The magical number 4 in short-term memory: a reconsideration of mental storage capacity. *Behavioral and brain sciences.* 2001. Т. 24, № 1. С. 87–114. URL: https://doi.org/10.1017/s0140525x01003922 (дата звернення: 11.12.2025).

2. Alvarez G. A., Cavanagh P. The capacity of visual short-term memory is set both by visual information load and by number of objects. *Psychological science*. 2004. Т. 15, № 2. С. 106–111. URL: https://doi.org/10.1111/j.0963-7214.2004.01502006.x (дата звернення: 11.12.2025).

3. Feature versus object in interpreting working memory capacity / W. Lin та ін. *Npj science of learning.* 2024. Т. 9, № 1. URL: https://doi.org/10.1038/s41539-024-00279-x (дата звернення: 11.12.2025).

4. Gonçales L. J., Farias K., da Silva B. C. Measuring the cognitive load of software developers: An extended Systematic Mapping Study. *Information and software technology.* 2021. Т. 136. С. 106563. URL: https://doi.org/10.1016/j.infsof.2021.106563 (дата звернення: 11.12.2025).

5. Mukhamadiev L. How cognitive load affects developer productivity. iDelsoft - Remote Talent Hiring. URL: https://idelsoft.com/blog/tpost/how-cognitive-load-affects-developer-productivity-and-how-to-fix-it (дата звернення: 11.12.2025).

6. Baletska Z. Reducing developer cognitive load in 2025: strategies for faster, smarter software delivery. Home | Agile Analytics | Grip op software development. URL: https://www.agileanalytics.cloud/blog/reducing-cognitive-load-the-missing-key-to-faster-development-cycles (дата звернення: 11.12.2025).

7. Yanhong W. Preattentive processing. *The ECPH encyclopedia of psychology. Singapore,* 2025. С. 1120–1121. URL: https://doi.org/10.1007/978-981-97-7874-4_343 (дата звернення: 11.12.2025).

8. Johansson R. C. G., Ulrich R. Serial processing of proximity groups and similarity groups. *Attention, perception, & psychophysics.* 2024. URL: https://doi.org/10.3758/s13414-024-02861-2 (дата звернення: 11.12.2025).

9. Skulmowski A. The cognitive architecture of digital externalization. *Educational psychology review.* 2023. Т. 35, № 4. URL: https://doi.org/10.1007/s10648-023-09818-1 (дата звернення: 11.12.2025).

10. Василик В. М., Гобир Л. М., Ваврик Т. О. Теоретичні аспекти візуальних мов програмування. *Вісник Херсонського національного технічного університету.* 2025. Т. 2, № 1(92). С. 38–44. URL: https://doi.org/10.35546/kntu2078-4481.2025.1.2.5 (дата звернення: 11.12.2025).

11. Model-Driven software engineering in practice: second edition / M. Brambilla та ін. Morgan & Claypool Publishers, 2017.

12. The archimate® enterprise architecture modeling language. www.opengroup.org. URL: https://www.opengroup.org/archimate-forum/archimate-overview (дата звернення: 11.12.2025).

13. SysML open source project: what is sysml? Who created it?. SysML.org. URL: https://sysml.org/ (дата звернення: 11.12.2025).

14. BPMN specification - business process model and notation. BPMN Specification - Business Process Model and Notation. URL: https://www.bpmn.org/ (дата звернення: 11.12.2025).

15. OMG® unified modeling language® (OMG UML®. Object Management Group. URL: https://www.omg.org/spec/UML/2.5.1/PDF (дата звернення: 11.12.2025).

16. BBVPL: a block-based visual programming language built on google’s blockly. International journal of advanced trends in computer science and engineering. 2021. Т. 10, № 3. С. 2524–2532. URL: https://doi.org/10.30534/ijatcse/2021/1441032021 (дата звернення: 11.12.2025).

17. Supporting the understanding and comparison of low-code development platforms / A. Sahay та ін. *2020 46th euromicro conference on software engineering and advanced applications (SEAA)*, м. Portoroz, Slovenia, 26–28 серп. 2020 р. 2020. URL: https://doi.org/10.1109/seaa51224.2020.00036 (дата звернення: 11.12.2025).

18. The C4 model for visualising software architecture. C4 model. URL: https://c4model.com/ (дата звернення: 11.12.2025).

19. Kruchten P. B. The 4+1 view model of architecture. *IEEE software.* 1995. Т. 12, № 6. С. 42–50. URL: https://doi.org/10.1109/52.469759 (дата звернення: 11.12.2025).

20. Acceleo | overview. Projects Gateway | The Eclipse Foundation. URL: https://eclipse.dev/acceleo/overview.html (дата звернення: 11.12.2025).

21. ATL | the eclipse foundation. The Community for Open Innovation and Collaboration. URL: https://eclipse.dev/atl/ (дата звернення: 11.12.2025).

22. Detecting meaning in RSVP at 13 ms per picture / M. C. Potter та ін. *Attention, perception, & psychophysics.* 2013. Т. 76, № 2. С. 270–279. URL: https://doi.org/10.3758/s13414-013-0605-z (дата звернення: 11.12.2025).

23. Kirchner H., Thorpe S. J. Ultra-rapid object detection with saccadic eye movements: visual processing speed revisited. *Vision research.* 2006. Т. 46, № 11. С. 1762–1776. URL: https://doi.org/10.1016/j.visres.2005.10.002 (дата звернення: 11.12.2025).

24. A survey of large language models / W. X. Zhao та ін. arXiv preprint arXiv:2303.18223, 2023. 144 с. URL: https://doi.org/10.48550/arXiv.2303.18223 (дата звернення: 11.12.2025).

25. The productivity effects of generative AI: evidence from a field experiment with github copilot / K. Z. Cui та ін. *An MIT exploration of generative AI.* 2024. URL: https://doi.org/10.21428/e4baedd9.3ad85f1c (дата звернення: 11.12.2025).

26. Lindemulder G., Kosinski M. What is a data flow diagram (DFD)?. IBM. URL: https://www.ibm.com/think/topics/data-flow-diagram (дата звернення: 11.12.2025).

27. Ahmed F., Robinson S., Tako A. A. Using the structred analysis and design technique (SADT) in simulation conceptual modeling. *2014 winter simulation conference - (WSC 2014),* м. Savanah, GA, USA, 7–10 груд. 2014 р. 2014. URL: https://doi.org/10.1109/wsc.2014.7019963 (дата звернення: 11.12.2025).

28. IDEFØ – function modeling method – IDEF. IDEF – Integrated DEFinition Methods (IDEF). URL: https://www.idef.com/idefo-function_modeling_method/ (дата звернення: 11.12.2025).

29. Lutkevich B. What is SSADM (structured systems analysis and design method)?. Search Software Quality. URL: https://www.techtarget.com/searchsoftwarequality/definition/SSADM (дата звернення: 11.12.2025).

30. Kissflow T. Rapid application development (RAD) | definition, steps & full guide. Kissflow. URL: https://kissflow.com/application-development/rad/rapid-application-development/ (дата звернення: 11.12.2025).

31. Lian V., Varoy E., Giacaman N. Learning object-oriented programming concepts through visual analogies. *IEEE transactions on learning technologies.* 2022. С. 1. URL: https://doi.org/10.1109/tlt.2022.3154805 (дата звернення: 11.12.2025).

32. End-user development, end-user programming and end-user software engineering: a systematic mapping study / B. R. Barricelli та ін. *Journal of systems and software.* 2019. Т. 149. С. 101–137. URL: https://doi.org/10.1016/j.jss.2018.11.041 (дата звернення: 12.12.2025).

33. What is infrastructure as code? - iac explained - AWS. Amazon Web Services, Inc. URL: https://aws.amazon.com/what-is/iac/ (дата звернення: 12.12.2025).

34. Schieferdecker I. K. Augmenting software engineering with AI and developing it further towards AI-assisted model-driven software engineering. *Semantic scholar.* 2024. URL: https://doi.org/10.48550/arXiv.2409.18048 (дата звернення: 12.12.2025).

35. Exploring the problems, their causes and solutions of AI pair programming: A study on GitHub and stack overflow / X. Zhou та ін. *Journal of systems and software.* 2024. С. 112204. URL: https://doi.org/10.1016/j.jss.2024.112204 (дата звернення: 12.12.2025).

36. Karpathy A. Software 2.0. Medium. URL: https://karpathy.medium.com/software-2-0-a64152b37c35 (дата звернення: 12.12.2025).

37. Model driven engineering for machine learning components: a systematic literature review / H. Naveed та ін. *Information and software technology.* 2024. Т. 169. С. 107423. URL: https://doi.org/10.1016/j.infsof.2024.107423 (дата звернення: 12.12.2025).

38. Bridging MDE and AI: a systematic review of domain-specific languages and model-driven practices in AI software systems engineering / S. Rädler та ін. Software and

systems modeling. 2024. URL: https://doi.org/10.1007/s10270-024-01211-y (дата звернення: 12.12.2025).

39. Model-driven engineering of safety and security software systems: a systematic mapping study and future research directions / A. Mashkoor та ін. *Journal of software: evolution and process.* 2022. URL: https://doi.org/10.1002/smr.2457 (дата звернення: 12.12.2025).

40. OMG meta object facility (MOF) core specification. Object Management Group. URL: https://www.omg.org/spec/MOF/2.5.1/PDF (дата звернення: 12.12.2025).

41. Idrees M., Aslam F. A comprehensive survey and analysis of diverse visual programming languages. *VFAST transactions on software engineering.* 2022. Т. 10, № 2. С. 47–60. URL: https://doi.org/10.21015/vtse.v10i2.1009 (дата звернення: 12.12.2025).

42. Özkan O., Babur Ö., van den Brand M. Domain-Driven Design in software development: a systematic literature review on implementation, challenges, and effectiveness. *Journal of systems and software.* 2025. № 230. С. 64. URL: https://doi.org/10.1016/j.jss.2025.112537 (дата звернення: 12.12.2025).

43. Kapferer S., Zimmermann O. Domain-Driven architecture modeling and rapid prototyping with context mapper. *Communications in computer and information science.* Cham, 2021. С. 250–272. URL: https://doi.org/10.1007/978-3-030-67445-8_11 (дата звернення: 12.12.2025).

44. Kapferer S., Zimmermann O. Domain-Driven service design. Service-Oriented computing. Cham, 2020. С. 189–208. URL: https://doi.org/10.1007/978-3-030-64846-6_11 (дата звернення: 12.12.2025).

45. Patterns on deriving apis and their endpoints from domain models / A. Singjai та ін. *EuroPLoP'21: european conference on pattern languages of programs 2021*, м. Graz Austria. New York, NY, USA, 2021. URL: https://doi.org/10.1145/3489449.3489976 (дата звернення: 12.12.2025).

46. Low-code development and model-driven engineering: two sides of the same coin? / D. Di Ruscio та ін. *Software and systems modeling.* 2022. Т. 21, № 2. С. 437–446. URL: https://doi.org/10.1007/s10270-021-00970-2 (дата звернення: 12.12.2025).

47. Firoz Mohammed Ozman. A systematic literature review on current developments of low code-no code solutions in the IT sector. *World journal of advanced engineering technology and sciences.* 2025. Т. 14, № 3. С. 162–169. URL: https://doi.org/10.30574/wjaets.2025.14.3.0072 (дата звернення: 12.12.2025).

48. The future of ai-driven software engineering / V. Terragni та ін. ACM transactions on software engineering and methodology. 2025. URL: https://doi.org/10.1145/3715003 (дата звернення: 12.12.2025).

49. Galhardo P., Silva A. R. d. Combining rigorous requirements specifications with low-code platforms to rapid development software business applications. *Applied sciences.* 2022. Т. 12, № 19. С. 9556. URL: https://doi.org/10.3390/app12199556 (дата звернення: 12.12.2025).

50. Ray P. P. A survey on visual programming languages in internet of things. *Scientific programming.* 2017. Т. 2017. С. 1–6. URL: https://doi.org/10.1155/2017/1231430 (дата звернення: 12.12.2025).

ДОДАТКИ

Додаток А

Порівняльна таблиця візуальних інструментів та методів для архітектурного моделювання програмного забезпечення

Таблиця А.1

Порівняльна таблиця інструментів та методів

№	Інструмент /Метод	Категорія	Візуаліза-ція	Мета-модель	Генерація коду	Синхр. з кодом	Когніт. склад-ність	Ліцензія	Ключове обмеження
1	**C4 Model**	Нотація	Графічна	✗	✗	✗	Низька	Open	Відсутня формальна семантика
2	**4+1 View Model**	Нотація	Графічна	✗	✗	✗	Середня	Open	Ручна синхронізація 5 представлень
3	**UML 2.5 / SysML 1.6**	Стандарт	Графічна	✓	◐	—	Висока	Open	Надмірна складність, 14+ діаграм
4	**ArchiMate 3.2**	Стандарт	Графічна	✓	✗	✗	Висока	Open	Фокус на EA, не на рівні коду
5	**AADL**	Стандарт	Текст +Граф	✓	◐	✗	Дуже висока	Open	Вузький домен (систем реального часу)
6	**TOGAF + ArchiMate**	Фреймворк	Графічна	✓	✗	✗	Висока	Open	Стратегічний рівень, не ПЗ
7	**EMF & Ecore**	MDE-платформа	—	✓	✓	✗	Висока	Open	Не надає візуального редактора
8	**Eclipse Sirius**	MDE-платформа	Графічна	✓	◐	✗	Висока	Open	Прив'язка до Eclipse, складне налаштува-ння
9	**Acceleo**	MDE-інструмент	—	✓	✓	✗	Середня	Open	Односпрямо-вана генерація (M2T)
10	**JetBrains MPS**	Language Workbench	Проєкцій-на	✓	✓	◐	Дуже висока	Open	Нетрадиційна парадигма редагування
11	**MetaEdit+**	Language Workbench	Графічна	✓	✓	✗	Середня	Комерці-йна	Закритість, пропрієтар-ний формат

Продовження табл. А.1

12	**WebGME**	MDE-платформа	Графічна	✓	◐	✗	Висока	Open	Менш зріла екосистема
13	**Xtext + Sirius/ Sprotty**	MDE-інструмент	Текст +Граф	✓	✓	◐	Висока	Open	Потребує інтеграції кількох інструментів
14	**Eclipse Papyrus**	CASE-інструмент	Графічна	✓	◐	◐	Висока	Open	Складність UML, лише структурна генерація
15	**Sparx EA**	CASE-інструмент	Графічна	◐	◐	◐	Середн я	Комерці -йна	Поверхнева інтеграція нотацій
16	**Archi**	CASE-інструмент	Графічна	◐	✗	✗	Низька	Open	Лише ArchiMate, без генерації
17	**Modelio**	CASE-інструмент	Графічна	◐	◐	◐	Середн я	Open	Нестабільніст ь, обмежена спільнота
18	**Diagrams.n et (draw.io)**	Редактор	Графічна	✗	✗	✗	Низька	Open	Відсутня семантична модель
19	**Context Mapper DSL**	DDD-інструмент	Текст →Граф	✓	◐	✗	Середн я	Open	Текстовий DSL, одностороння візуалізація
20	**EventStorm ing**	Методика	Графічна	✗	✗	✗	Низька	Open	Неформалізов ані артефакти
21	**Structurizr DSL**	As-Code	Текст →Граф	◐	✗	✗	Середн я	Freemiu m	Текст-first, немає WYSIWYG
22	**PlantUML + C4**	As-Code	Текст →Граф	✗	✗	✗	Середн я	Open	Немає семантики, обмежений layout
23	**IcePanel**	Хмарний CASE	Графічна	◐	✗	✗	Низька	Комерці -йна	Хмарна залежність, без генерації
24	**Mermaid**	As-Code	Текст →Граф	✗	✗	✗	Низька	Open	Обмежена виразність
25	**Capella**	MBSE	Графічна	✓	◐	✗	Висока	Open	Системна інженерія, не чисто ПЗ
26	**System Composer**	MBSE	Графічна	✓	◐	◐	Висока	Комерці -йна	Вбудовані системи, інший домен
Легенда: ✓ – повна підтримка , ◐ – часткова, ✗ – відсутня підтримка, «—» – не застосовується.									