БАКАЛАВРСЬКА РОБОТА

БР. ІП - 40.00.00.000 ПЗ

Група ІП-22-2

Кучера Максим

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Кучера Максим Іванович

(прізвище, ім'я, по батькові)

УДК 004.4

(індекс)

БАКАЛАВРСЬКА РОБОТА

Методи генерації прототипів програмних систем засобами модельно-орієнтованої інженерії

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного забезпечення

(шифр і назва спеціальності)

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

Здобувач освітнього рівня Кучера М.І.

(підпис, ініціали та прізвище здобувача)

Науковий керівник Вовк Роман Богданович, к.т.н., доцент

(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

доц. Бандура В.В.

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

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Кучері Максиму Івановичу

(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) “ Методи генерації прототипів програмних систем засобами модельно-орієнтованої інженерії”

керівник проекту (роботи) Вовк Р.Б., доцент, к.т.н,

затверджені наказом закладу вищої освіти від “ 21 ” квітня 2026р. № 179 /7

2. Строк подання студентом проекту (роботи) 10 червня 2026 р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

4. Зміст розрахунково - пояснювальної записки (перелік питань, які потрібно розробити)

1. Аналіз проблематики використання модельно-орієнтованої інженерії для розробки ПЗ
2. Алгоритмічне обґрунтування проєктування систем генерації прототипів ПЗ
3. Проектування архітектури механізму робочих процесів
4. Реалізація методів генерації прототипів програмних систем
5. Тестування функціональної придатності системи генерації прототипів ПЗ

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1. Схема, що представляє основні концепції модельно-орієнтованої інженерії (рис. 1.1, ст. 13)
2. Трирівнева ієрархія абстракції базису MDA (рис. 1.2, ст. 17)
3. UML-модель сценарію обробки замовлення (сценарій 1) (рис. 2.1, ст. 30)
4. UML-модель сценарію обробки замовлення (сценарій 2) (рис. 2.2, ст. 31)
5. Діаграма компонентів інтегрованої системи (рис. 2.4, ст. 33)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання **21 квітня 2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Аналіз проблематики використання модельно-орієнтованої інженерії для розробки ПЗ	04.05.2026	виконано
2	Алгоритмічне обґрунтування проєктування систем генерації прототипів ПЗ	15.05.2026	виконано
3	Проектування архітектури механізму робочих процесів	21.05.2026	виконано
4	Реалізація методів генерації прототипів програмних систем	28.05.2026	виконано
5	Тестування функціональної придатності системи генерації прототипів ПЗ	03.06.2026	виконано
6	Оформлення пояснювальної записки дипломної роботи завідувачем кафедри	10.06.2026	виконано

Студент – дипломник _____Кучера М.І.
(підпис)

Керівник роботи _____Вовк Р.Б.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 81 сторінки, 18 рисунків, список використаних джерел із 36 найменуваннями.

Метою роботи є розробка методів генерації прототипів програмних систем на основі модельно-орієнтованої інженерії з використанням UML-моделей та механізмів керування робочими процесами.

Об'єкт дослідження - процеси розробки програмного забезпечення на основі модельно-орієнтованої інженерії.

Предмет дослідження - методи та алгоритми генерації прототипів програмних систем із UML-моделей із використанням механізмів керування робочими процесами та архітектур, керованих даними.

В першому розділі проведений аналіз підтвердив доцільність використання модельно-орієнтованої інженерії як основи для автоматизованої генерації програмних систем.

В другому розділі розроблено методологічні та алгоритмічні засади побудови системи генерації прототипів програмного забезпечення.

В третьому розділі реалізовано та експериментально перевірено ефективність запропонованих методів генерації прототипів програмних систем**.**

Висновок: розроблені методи генерації прототипів програмних систем забезпечують ефективну автоматизацію процесу переходу від моделей до виконуваних програмних рішень.

КЛЮЧОВІ СЛОВА: МОДЕЛЬНО-ОРІЄНТОВАНА ІНЖЕНЕРІЯ, UML, ГЕНЕРАЦІЯ ПРОТОТИПІВ, ПРОГРАМНІ СИСТЕМИ, РОБОЧІ ПРОЦЕСИ, МЕТАДАНІ, АВТОМАТИЗАЦІЯ РОЗРОБКИ, ВЕБ-ЗАСТОСУНКИ, АРХІТЕКТУРА, DJANGO

ANNOTATION

The bachelor's thesis contains 81 pages, 18 figures, a list of used sources with 36 names.

The purpose of the work is to develop methods for generating prototypes of software systems based on model-oriented engineering using UML models and workflow management mechanisms.

The object of the study is software development processes based on model-oriented engineering.

The subject of the study is methods and algorithms for generating prototypes of software systems from UML models using workflow management mechanisms and data-driven architectures.

In the first section, the analysis confirmed the feasibility of using model-oriented engineering as a basis for automated generation of software systems.

In the second section, methodological and algorithmic principles for building a software prototype generation system are developed.

In the third section, the effectiveness of the proposed methods for generating prototypes of software systems is implemented and experimentally verified.

Conclusion: the developed methods for generating prototypes of software systems provide effective automation of the process of transition from models to executable software solutions.

KEYWORDS: MODEL-ORIENTED ENGINEERING, UML, PROTOTYPE GENERATION, SOFTWARE SYSTEMS, WORK PROCESSES, METADATA, DEVELOPMENT AUTOMATION, WEB APPLICATIONS, ARCHITECTURE, DJANGO

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

MDE – Model-Driven Engineering

MDA – Model-Driven Architecture

WF – Workflow

WfMS – Workflow Management System

CRUD – Create, Read, Update, Delete

MVC – Model-View-Controller

MVT – Model-View-Template

ORM – Object-Relational Mapping

XML – Extensible Markup Language

PIM – Platform-Independent Model

PSM – Platform-Specific Model

DSL – Domain-Specific Language

AST – Abstract Syntax Tree

RBAC – Role-Based Access Control

JWT – JSON Web Token

OOP – Object-Oriented Programming

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується зростанням складності програмних систем, підвищенням вимог до швидкості їх розробки, гнучкості, масштабованості та якості. У цих умовах традиційні підходи до проєктування програмного забезпечення дедалі частіше виявляються недостатньо ефективними, що зумовлює необхідність застосування нових інженерних парадигм. Однією з таких парадигм є модельно-орієнтована інженерія, яка передбачає використання моделей як основного артефакту розробки та забезпечує автоматизацію процесів трансформації вимог у програмну реалізацію.

Застосування модельно-орієнтованого підходу дозволяє значно підвищити рівень абстракції, забезпечити узгодженість між різними рівнями представлення системи та зменшити залежність від конкретних технологічних платформ. Водночас існуючі інструменти та методи автоматизованої генерації програмного забезпечення часто не забезпечують достатньої гнучкості, особливо у контексті динамічних систем, що потребують підтримки складних робочих процесів, умовної логіки та паралельного виконання.

Особливої актуальності набуває проблема генерації прототипів програмних систем, які дозволяють швидко перевіряти концепції, тестувати бізнес-логіку та адаптувати системи до змінних вимог. У цьому контексті важливим є поєднання можливостей UML-моделювання з архітектурними особливостями сучасних веб-фреймворків, зокрема підходів, керованих даними, та механізмів керування робочими процесами.

Дана робота присвячена дослідженню та розробці методів генерації прототипів програмних систем засобами модельно-орієнтованої інженерії, що дозволяють автоматизувати процес переходу від моделей до виконуваних систем, забезпечуючи при цьому гнучкість, адаптивність та розширюваність

програмних рішень.

Актуальність роботи

Актуальність дослідження зумовлена необхідністю підвищення ефективності процесів розробки програмного забезпечення в умовах постійного ускладнення інформаційних систем та зростання вимог до їх функціональності. Традиційні підходи до програмування, що базуються на ручному кодуванні, не забезпечують достатньої швидкості розробки та ускладнюють підтримку і модифікацію систем.

Модельно-орієнтована інженерія пропонує концептуально новий підхід до створення програмного забезпечення, однак її практичне застосування обмежується недостатньо розвиненими методами генерації виконуваних систем із моделей. Особливо це стосується задач побудови прототипів, які повинні бути не лише структурно коректними, але й функціонально придатними до використання в умовах реального часу.

Крім того, сучасні інформаційні системи дедалі частіше базуються на динамічних робочих процесах, що вимагає використання адаптивних механізмів керування, здатних змінювати поведінку системи без модифікації вихідного коду. Це створює додаткові виклики для систем автоматизованої генерації.

Таким чином, розробка методів генерації прототипів програмних систем, які поєднують модельно-орієнтований підхід із механізмами керування робочими процесами та архітектурами, керованими даними, є актуальним науково-практичним завданням.

Метою роботи є розробка методів генерації прототипів програмних систем на основі модельно-орієнтованої інженерії з використанням UML-моделей та механізмів керування робочими процесами.

Об’єкт дослідження - процеси розробки програмного забезпечення на основі модельно-орієнтованої інженерії.

Предмет дослідження - методи та алгоритми генерації прототипів

програмних систем із UML-моделей із використанням механізмів керування робочими процесами та архітектур, керованих даними.

Завдання дослідження:

1. проаналізувати теоретичні засади модельно-орієнтованої інженерії;

2. дослідити можливості застосування UML для генерації програмного забезпечення;

3. розробити концептуальну модель відображення UML на програмну архітектуру;

4. обґрунтувати архітектуру механізму керування робочими процесами;

5. розробити методологію генерації прототипів програмних систем;

6. реалізувати алгоритмічне забезпечення генерації та виконання робочих процесів;

7. провести тестування та оцінку ефективності розробленої системи.

Методи дослідження

У роботі використано методи системного аналізу для дослідження існуючих підходів, методи модельно-орієнтованого проєктування для побудови архітектури системи, методи формалізації та трансформації моделей для генерації програмного коду, алгоритмічні методи для реалізації механізмів керування робочими процесами, а також методи експериментального дослідження для верифікації та тестування розробленої системи.

Наукова новизна

Наукова новизна одержаних результатів полягає у розробці методу генерації прототипів програмних систем, який поєднує UML-моделювання з механізмами керування робочими процесами, що функціонують у режимі реального часу. Запропоновано підхід до розширення метаданих UML-моделей для опису поведінкової логіки системи. Удосконалено методи відображення моделей на архітектуру веб-застосунків, що забезпечує динамічність і адаптивність систем. Отримали подальший розвиток методи

автоматизованої генерації інтерфейсів користувача.

Практичне застосування

Практичне значення роботи полягає у можливості використання розроблених методів і програмних рішень для створення прототипів інформаційних систем, зокрема систем керування бізнес-процесами та веб-застосунків. Запропонований підхід дозволяє скоротити час розробки, підвищити якість програмного забезпечення та забезпечити гнучкість адаптації систем до змін вимог.

Бакалаврська робота містить 81 сторінку, 18 рисунків, 3 розділи список використаних джерел із 36 найменуваннями і 1 додаток.

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ ВИКОРИСТАННЯ МОДЕЛЬНО-ОРІЄНТОВАНОЇ ІНЖЕНЕРІЇ ДЛЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1. Основні концепції модельно-орієнтованої інженерії

Модельно-орієнтована інженерія (Model-Driven Engineering, MDE) спрямована на розробку та застосування моделей з метою автоматизації та оптимізації процесів створення програмного забезпечення. Актуальною науково-практичною проблемою як для академічних інструментаріїв, так і для комерційних low-code платформ залишається автоматизована генерація виконуваних робочих процесів (workflows) на основі діаграм діяльності UML.

Модельно-орієнтована інженерія - це методологія розробки програмного забезпечення, де головним фокусом є створення та експлуатація моделей, а не написання коду вручну.

Розглянемо основні концепції, на яких тримається цей підхід:

1. Моделі як першокласні сутності

В традиційному програмуванні модель (наприклад, UML-схема) — це лише документація, яка швидко застаріває. В MDE модель є живим активом, з якого генерується код, конфігурація або навіть документація. Моделі дозволяють розробникам працювати на рівні бізнес-логіки, ігноруючи технічні деталі реалізації (наприклад, особливості конкретної бази даних).

2. Рівні абстракції (MDA)

Архітектура, керована моделями (Model Driven Architecture, MDA), визначає три основні рівні:

- CIM (Computation Independent Model) - описує бізнес-контекст і вимоги. Зрозуміла бізнес-аналітикам.

- PIM (Platform Independent Model) - описує логіку системи без

прив'язки до технологій (наприклад, Java, Python чи .NET).

- PSM (Platform Specific Model) - модель, адаптована під конкретну технологічну платформу.

3. Метамоделювання

Щоб комп'ютер розумів модель, вона повинна бути побудована за суворими правилами. Метамодель — це «модель моделі». Вона визначає правила: які елементи можна використовувати (наприклад, «Клас», «Атрибут», «Зв’язок») та як вони взаємодіють. Найвідомішим стандартом є MOF (Meta-Object Facility).

4. Трансформації моделей

Це «двигун» MDE. Процес перетворення однієї моделі в іншу або в код. M2M (Model-to-Model) - перетворення PIM у PSM, наприклад, абстрактна схема бази даних перетворюється на схему для PostgreSQL. M2T (Model-to-Text) - Генерація вихідного коду (наприклад, Java-класів) або документації з моделі.

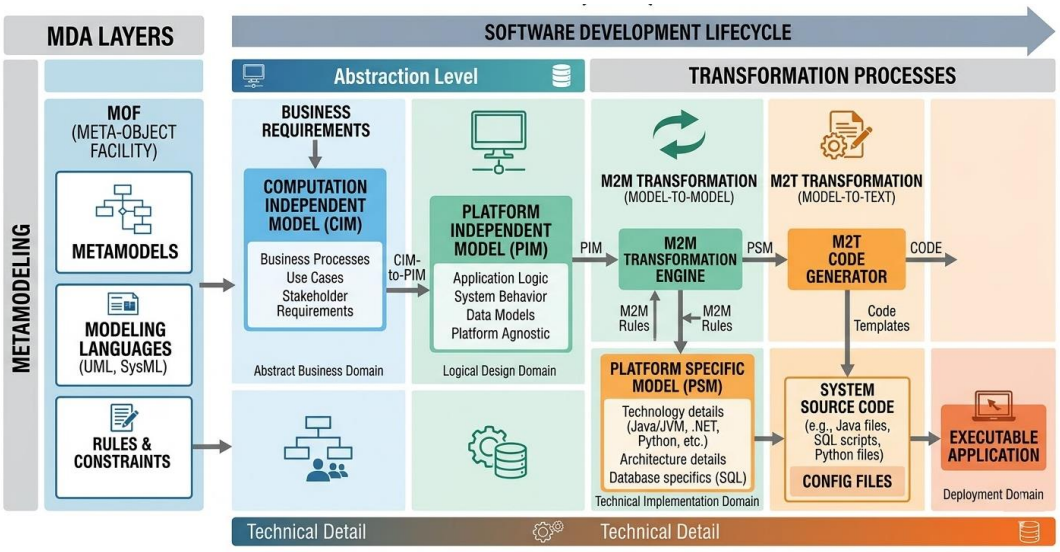

Рисунок 1.1 – Схема, що представляє основні концепції модельно-орієнтованої інженерії

Таблиця 1.1

Переваги MDE

Перевага	Опис
Підвищення продуктивності	Автоматизація рутинного написання коду.
Якість та узгодженість	Менше людських помилок при реалізації складної архітектури.
Довговічність	Бізнес-логіка (PIM) залишається актуальною, навіть якщо технології (PSM) змінюються.
Інтероперабельність	Легша інтеграція між різними системами завдяки спільним моделям.

Попри переваги, MDE вимагає високої кваліфікації архітекторів для створення якісних метамоделей та складних інструментів трансформації. Також часто виникає проблема синхронізації, коли розробники вносять зміни безпосередньо в код, ігноруючи модель (так званий «code-model drift»).

1.2. Методологічні засади архітектури, керованої моделями, у парадигмі модельно-орієнтованої інженерії

Модельно-орієнтована розробка (Model-Driven Development, MDD) постає як фундаментальна парадигма інженерії програмного забезпечення, що ґрунтується на концепції використання абстрактних моделей як першочергових артефактів для подальшої автоматизованої трансформації у функціональні програмні системи [1]. На відміну від традиційних підходів, де моделі виконують допоміжну документарну роль, у межах MDD вони є операційними одиницями, що визначають життєвий цикл розробки.

Однією з найбільш репрезентативних реалізацій цього підходу є Архітектура на основі моделей (Model Driven Architecture, MDA) — концептуальний фреймворк, розроблений консорціумом Object Management Group (OMG). Стратегічна мета MDA полягає у максимізації доданої вартості від процесів моделювання для ефективного управління складністю та взаємозалежністю компонентів у гетерогенних програмних середовищах [2].

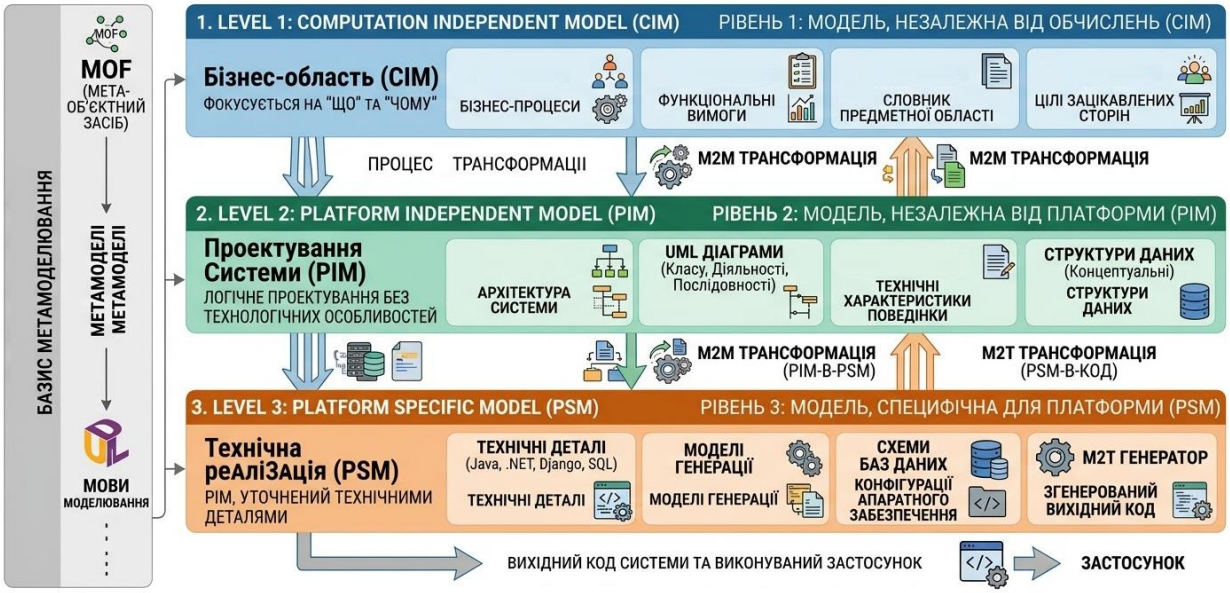

Рисунок 1.2 - Трирівнева ієрархія абстракції базису MDA

Методологічний базис MDA спирається на трирівневу ієрархію абстракції:

1. Модель, незалежна від обчислень (Computation Independent Model, CIM) - описує систему в контексті її бізнес-середовища та функціональних вимог. CIM фокусується на предметній області (domain), ігноруючи технічні аспекти автоматизації, що дозволяє встановити спільний концептуальний апарат між розробниками та зацікавленими сторонами.

2. Модель, незалежна від платформи (Platform Independent Model, PIM) - визначає архітектурну структуру та динамічну поведінку системи без прив'язки до конкретних технологічних стеків. PIM забезпечує високий рівень інваріантності логіки системи щодо можливих змін у технологіях реалізації.

3. Модель, специфічна для платформи (Platform Specific Model, PSM) - є результатом уточнення PIM шляхом інтеграції технічних специфікацій конкретної платформи (наприклад, J2EE, .NET, Django). PSM слугує безпосередньою основою для генерації вихідного коду та конфігураційних файлів.

Ключовим механізмом переходу між зазначеними рівнями є

автоматизовані трансформації (Model Transformations), які базуються на формальних правилах відображення (mapping rules). Це стає можливим завдяки використанню єдиного стандарту метамоделювання — Meta-Object Facility (MOF), що забезпечує семантичну цілісність моделей на різних етапах проектування.

Впровадження MDA у практику розробки забезпечує низку детермінованих переваг:

- Оптимізація часових витрат - значне прискорення виходу продукту на ринок (Time-to-Market) за рахунок автоматичної генерації коду.

- Технологічна стійкість - спрощення процесів міграції та оновлення систем завдяки збереженню бізнес-логіки на рівні PIM.

- Верифікованість та узгодженість - мінімізація розриву між дизайном та імплементацією, що гарантує відповідність фінального коду початковим специфікаціям.

- Покращення комунікації - підвищення рівня розуміння архітектурних рішень усіма учасниками проєкту через візуалізацію складних процесів.

1.3. Специфікація та роль уніфікованої мови моделювання у процесах автоматизованої генерації ПЗ

1.3.1. Загальна характеристика та класифікація UML

Уніфікована мова моделювання (Unified Modeling Language, UML) постає як де-факто стандарт у галузі програмної інженерії, забезпечуючи універсальний графічний апарат для специфікації, візуалізації, проєктування та документування артефактів складних систем. UML була офіційно прийнята консорціумом Object Management Group (OMG) у 1997 році. Поточна ревізія стандарту — 2.5.1 — відображає еволюцію парадигм об'єктно-орієнтованого аналізу та дизайну (OOAD).

Відповідно до специфікації OMG [10], таксономія UML охоплює 14

типів діаграм, класифікованих за двома фундаментальними категоріями:

- Структурні діаграми (7 типів) - визначають статичну архітектуру системи, її компоненти, класи, об’єкти та зв’язки між ними. Прикладами, що вже інтегровані в архітектуру генерації проєкту AI4MDE, є діаграми класів (визначають схему даних) та діаграми варіантів використання (окреслюють межі системи та функціональні ролі).

- Поведінкові діаграми (7 типів) - описують динамічні аспекти функціонування системи, взаємодію об’єктів у часі та реакцію на події. До цієї категорії належить діаграма діяльності (Activity Diagram), яка є об'єктом даного дослідження для розширення можливостей AI4MDE.

1.3.2. Моделювання бізнес-процесів за допомогою діаграм діяльності

Діаграма діяльності в контексті UML призначена для формалізованого опису алгоритмічної логіки та робочих процесів (workflows). На відміну від діаграм послідовності, які фокусуються на обміні повідомленнями між об'єктами, діаграма діяльності акцентує увагу на потоках управління (control flows) та даних [4].

Основними семантичними одиницями моделі діяльності є:

- Дії (Action Nodes) - неподільні кроки виконання завдання в межах процесу.

- Вузли управління (Control Nodes) - елементи, що детермінують маршрутизацію потоку:

  - Вузли рішення та злиття - реалізація умовної логіки (if-then-else).

  - Вузли розгалуження та об'єднання - ініціалізація та синхронізація паралельних обчислювальних потоків.

- Доріжки - механізм логічного групування дій за зонами відповідальності, що дозволяє чітко розмежувати обов'язки в процесах.

1.3.3. Сучасний стан та проблематика генерації коду на основі UML

Інтеграція UML у процеси автоматизованої генерації коду (Model-to-Code, M2C) є об'єктом численних досліджень, проте досягнення повної функціональності генерованих систем залишається складним завданням.

Аналіз наукових праць [5 - 9] демонструє такі тенденції:

1. Обмеженість семантичного охоплення/ Дослідження, присвячені генерації Java-коду на основі діаграм класів та послідовностей [5], підтверджують можливість автоматичного створення структурних каркасів. Проте діаграми послідовностей часто виявляються недостатніми для опису повної реалізації методів, оскільки вони орієнтовані на виклики інтерфейсів, а не на внутрішню логіку обробки даних.

2. Проблема моно-діаграмного підходу - vетоди, що базуються на перетворенні лише одного типу діаграми [6], за своєю природою не здатні генерувати цілісні системи. Це зумовлено тим, що одна модель (наприклад, структурна) не містить інформації про поведінкові аспекти, і навпаки.

3. Концептуальний розрив. У роботі [7], де проаналізовано 40 методологій, констатується наявність значного розриву в автоматичному перетворенні моделей у вихідний код. Автори підкреслюють, що домінування Java як цільової мови та фокус на ізольованих діаграмах перешкоджають створенню «end-to-end» рішень.

1.4. Концептуальна модель відображення елементів UML на архітектуру Django

Для забезпечення функціонування застосунків, орієнтованих на робочі процеси (workflow-aware applications), кожен елемент діаграми діяльності має бути інтерпретований як компонент архітектури MVT (Model-View-Template).

Таблиця 1.2

Порівняльна таблиця відображення

Елемент UML	Програмна конструкція Django / Python	Функціональне призначення
Action Node	View (або метод у Service Layer)	Виконання конкретної операції або рендеринг форми для введення даних.
Control Flow	URL Routing / redirect()	Визначає наступний крок процесу та маршрутизацію користувача.
Decision Node	if/else логіка у View	Перевірка умов (наприклад, валідація форми) для вибору шляху.
Merge Node	Спільний URL або точка входу	Об'єднання різних потоків в один логічний етап.
Swimlane	Permissions / User Groups	Обмеження доступу до конкретних дій на основі ролі користувача.
Initial/Final Node	Landing Page / Success View	Точки ініціалізації процесу та його логічного завершення.

Наведемо методику технічної реалізації ключових вузлів.

1. Трансформація вузлів дії (Action Nodes)

Кожна дія в UML відображається на окремий Django View. Якщо дія передбачає взаємодію з користувачем (наприклад, "Заповнити заявку"), генератор створює FormView. Якщо дія є автоматизованою (наприклад, "Надіслати сповіщення"), вона імплементується як фонове завдання або метод у контролері.

2. Реалізація вузлів рішень (Decision Nodes)

Вузли рішень трансформуються в умовні оператори всередині методу form_valid() або dispatch().

Метадані UML: Умова переходу (Guard condition).

Реалізація в Django: Логіка перевірки атрибутів моделі або даних сесії, що визначає, на який URL буде перенаправлено користувача (HttpResponseRedirect).

3. Рольова модель на основі доріжок (Swimlanes)

Використання доріжок дозволяє автоматично генерувати декоратори доступу. Наприклад, якщо дія знаходиться в доріжці "Менеджер", до відповідного View у Django додається міксин PermissionRequiredMixin або

декоратор @permission_required. Це гарантує, що лише користувачі з відповідними правами зможуть виконати даний крок робочого процесу.

4. Управління станом (Workflow State)

Для відстеження прогресу користувача в межах діаграми діяльності, в базу даних (Django Models) додається службове поле current_step або створюється окрема модель ProcessInstance. Це дозволяє системі "пам'ятати", на якому етапі зупинився користувач, та запобігати несанкціонованим переходам між вузлами.

Запропонований підхід до мапінгу дозволяє перетворити статичну візуалізацію UML у динамічну систему навігації. Використання Django-шаблонів (Jinja2) для генерації цих конструкцій забезпечує високу швидкість створення прототипів, де логіка переходів повністю відповідає бізнес-моделі, описаній на рівні PIM.

1.5. Архітектурна концепція керованих даними механізмів робочих процесів у контексті динамічних інформаційних систем

1.5.1. Визначення та функціональна архітектура механізму робочого процесу

Згідно з дефініціями Коаліції з управління робочими процесами (Workflow Management Coalition, WfMC), механізм робочого процесу (workflow engine) інтерпретується як сервіс програмного забезпечення, що забезпечує середовище виконання для екземплярів бізнес-процесів.

Відповідно до еталонної моделі Workflow Management Coalition представленої в [11], такий механізм є центральним компонентом системи управління робочими процесами (WfMS) і відповідає за реалізацію таких критичних функцій:

- Управління життєвим циклом: ініціалізація, призупинення, відновлення та термінація екземплярів процесів.

- Навігація та маршрутизація: забезпечення логічної послідовності виконання дій, включаючи обробку паралельних потоків, циклів та умовних розгалужень.

- Інтерпретація визначень: обробка метаданих процесу для виклику відповідних програмних ресурсів або взаємодії з користувачем.

- Підтримка консистентності даних: збереження та передача контексту даних робочого процесу між етапами його виконання.

1.5.2. Аналіз екосистеми Django та обмеження статичних підходів

У межах розробки проєкту, що базується на фреймворку Django, було проаналізовано спеціалізовані рішення для управління потоками робіт, такі як django-viewflow, django-joeflow та django-fsm-2.

Попри високу надійність цих інструментів у керуванні переходами станів (FSM), вони характеризуються високим рівнем зв’язності (tight coupling) із програмним кодом. Зокрема:

- Статичність дефініцій: Робочі процеси жорстко зафіксовані в коді моделей або класів переходів.

- Складність модифікації: Будь-яка зміна бізнес-логіки вимагає редагування вихідного коду, проведення міграцій бази даних та повторного розгортання застосунку.

- Обмеження гнучкості: У динамічних середовищах, де процеси еволюціонують швидше за цикли релізу ПЗ, такий підхід стає критичним бар'єром для масштабування.

1.5.3. Парадигма керованих даними систем

Для вирішення проблеми статичності пропонується перехід до керованої даними архітектури, де структура робочого процесу зберігається у вигляді метаданих у реляційній базі даних. Це дозволяє абстрагувати логіку процесу від програмної імплементації.

Переваги керованого даними підходу:

- Динамічна реконфігурація: Можливість модифікації графів процесів у реальному часі без втручання у вихідний код.

- Адаптивність: Швидка адаптація системи до нових вимог шляхом оновлення записів у БД.

- Еволюція систем: Підтримка версійності процесів, що дозволяє паралельно виконувати старі та нові екземпляри процесів.

Ця концепція безпосередньо корелює з теорією динамічних спроможностей (dynamic capabilities) [12]. Вона визначає здатність організації інтегрувати, розбудовувати та реконфігурувати внутрішні компетенції для адекватної відповіді на виклики високотурбулентного середовища. Впровадження керованого даними механізму дозволяє генерувати програмні продукти, що не лише автоматизують поточні операції, а й забезпечують стратегічну гнучкість інформаційної інфраструктури підприємства.

Отже, перехід від жорсткого кодування логіки до її моделювання в БД є ключовим фактором створення життєздатних прототипів. Це дозволяє подолати "семантичний розрив" між бізнес-моделлю та її реалізацією, забезпечуючи високу відповідність системи реальним потребам користувачів у довгостроковій перспективі.

1.6. Еволюція методів автоматизованої генерації інтерфейсів користувача у системах модельно-орієнтованої інженерії

1.6.1. Аналіз ранніх ітерацій та обмеження візуального проектування

На початкових етапах розвитку інструментів генерації програмного забезпечення значна увага приділялася редакторам типу WYSIWYG («що бачиш, те й отримуєш»). Такі рішення дозволяли розробникам створювати інтерфейсні сторінки з полями введення, що безпосередньо корелювали з

атрибутами відповідних класів моделей. Проте практична експлуатація подібних інструментів виявила суттєві методологічні недоліки:

- Необхідність ручного проектування кожної окремої сторінки нівелювала переваги автоматизації, що є фундаментальною метою модельно-орієнтованого підходу.

- Ранні реалізації часто обмежувалися візуальною частиною, не забезпечуючи автоматичного формування логіки обробки запитів (Views) та механізмів маршрутизації (URLs), необхідних для повноцінного функціонування застосунку.

Для подолання зазначених обмежень у сучасних архітектурах генерації було впроваджено розширену модель метаданих, що базується на трирівневій ієрархії компонентів: категорії, сторінки та секції.

Секційні компоненти - призначені для реалізації базових операцій маніпулювання даними (CRUD: створення, читання, оновлення, видалення) над конкретними сутностями моделі.

Сторінки - виступають агрегаторами для одного або декількох секційних компонентів, формуючи логічно завершений вузол інтерфейсу.

- Категорії - забезпечують вищий рівень абстракції, дозволяючи групувати споріднені сторінки в межі єдиної навігаційної структури.

Такий підхід дозволяє формалізувати опис інтерфейсу на рівні моделі, що є передумовою для його подальшої автоматичної імплементації.

1.6.2. Шаблонізація та трансформація моделей у рамках архітектурного патерну MVT

Процес перетворення високорівневих метаданих у виконуваний код реалізується за допомогою механізмів шаблонізації. Ця трансформація забезпечує генерацію артефактів, що відповідають архітектурному шаблону Model-View-Template (MVT):

- Рівень моделей - описи класів, визначені користувачем, трансформуються в уніфікований файл дефініцій моделей для запобігання

дублюванню структур даних.

- Рівень контролерів - для кожного компонента системи автоматично генеруються класи або функції обробки (Views), що забезпечують бізнес-логіку операцій CRUD та управління відображенням сторінок.

- Рівень представлення - на основі заданих шаблонів створюються HTML-файли та відповідні стилі, що формують фронтенд-частину застосунку.

Перехід від мануального візуального проектування до генерації на основі формалізованих метаданих дозволив мінімізувати потребу у глибокій технічній експертизі та значно скоротити обсяг ручного написання коду. Сучасні системи, що використовують даний метод, здатні створювати функціональні прототипи з розвиненою логікою обробки даних, забезпечуючи високу відповідність між початковою моделлю та кінцевим програмним продуктом.

1.7 Висновки до розділу

У першому розділі здійснено системний аналіз теоретичних засад модельно-орієнтованої інженерії та визначено її роль у сучасних процесах розробки програмного забезпечення. Встановлено, що використання моделей як первинних артефактів проєктування сприяє підвищенню рівня абстракції та узгодженості між вимогами і реалізацією систем. Досліджено методологію архітектури, керованої моделями, що дозволило обґрунтувати доцільність багаторівневого підходу до представлення програмних систем. Проаналізовано можливості та обмеження застосування UML у задачах автоматизованої генерації коду, зокрема у контексті відображення моделей на архітектуру веб-фреймворків. Узагальнення отриманих результатів дозволило сформувати концептуальні передумови для розробки методів генерації прототипів програмних систем.

РОЗДІЛ 2. МЕТОДОЛОГІЧНЕ ТА АЛГОРИТМІЧНЕ ОБҐРУНТУВАННЯ ПРОЄКТУВАННЯ СИСТЕМ ГЕНЕРАЦІЇ ПРОТОТИПІВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У даному розділі представлено концептуальний підхід до розробки архітектури генератора, здатного трансформувати діаграми діяльності UML у функціональні програмні застосунки з підтримкою складних робочих процесів. Методологічний базис дослідження становить синергія парадигми дизайн-дослідження та ітеративно-інкрементної моделі розробки, що реалізується через послідовний аналіз трьох тематичних кейсів зростаючої складності.

2.1. Методологія дизайн-дослідження

В основі даної роботи лежить метод дизайн-дослідження, де безпосередній процес проєктування артефакту слугує ключовим інструментом отримання наукового знання. Згідно з визначенням дослідницького комітету EAAE, цей підхід охоплює наукову діяльність, у якій дизайн є суттєвою складовою дослідницького процесу, що дозволяє формувати нові теоретичні та практичні інсайти безпосередньо через акт створення системи [10].

У контексті даної роботи процес розширення функціональних можливостей генератора прототипів розглядається одночасно як об'єкт вивчення та як процесуальний метод. Валідація отриманих результатів базується на принципі експертної оцінки (peer review), що передбачає аналіз розробки групою фахівців із відповідними дисциплінарними компетенціями. Емпіричне підтвердження гіпотез здійснюється шляхом ітеративної апробації та користувацького тестування, детальний опис яких наведено в наступному підрозділі.

Дослідження реалізується в межах ітеративного життєвого циклу, де архітектура генератора еволюціонує через серію інкрементальних циклів вдосконалення. Кожна ітерація спрямована на вирішення специфічних задач у межах обраного кейсу, що забезпечує поступове нарощування функціональної потужності системи.

Процес розробки структуровано за принципом від простого до складного:

1. Початкова фаза. Моделювання базових потоків управління UML між вузлами дій, що дозволяє перевірити коректність лінійної маршрутизації.

2. Проміжна фаза. Інтеграція складних структур управління, зокрема вузлів прийняття рішень (decisions), логічних злиттів, а також механізмів паралельного виконання та синхронізації потоків.

3. Фінальна фаза. На основі зворотного зв'язку, отриманого під час попередніх циклів, здійснюється комплексне вдосконалення алгоритмів генерації.

Завершальним етапом дослідження є проведення тестування. Це дозволяє провести об'єктивну оцінку функціональної повноти, юзабіліті-характеристик та загальної ефективності генератора при вирішенні практичних завдань. Такий підхід гарантує, що результуючий артефакт відповідає вимогам як технічної коректності, так і користувацької зручності.

2.2. Верифікація архітектури генератора прототипів через тематичні сценарії (Case Studies)

Для оцінки функціональної спроможності розробленого генератора щодо створення workflow-орієнтованих застосунків на основі діаграм діяльності UML застосовано метод тематичних досліджень, що виконують роль комплексного інтеграційного тестування. Зазначені тестування спрямовані на верифікацію системи в цілому, зокрема на аналіз її взаємодії з

існуючою архітектурою генерації та оцінку цілісності результуючого програмного продукту.

Кожне дослідження розпочинається з етапу проєктування формалізованої UML-моделі діяльності, яка визначає межі та логіку досліджуваного сценарію. Процес ітеративного вдосконалення системи базувався на результатах сесій експертного оцінювання за участю технічних архітекторів та наукових консультантів. Порівняльний аналіз проводився шляхом зіставлення детермінованої поведінки, закладеної в UML-діаграмах, із фактичною логікою функціонування згенерованого прототипу.

Обидва тематичні дослідження базуються на єдиному функціональному сценарії — процесі обробки замовлення в системі електронної комерції. Рольова модель сценарію охоплює трьох ключових акторів: Клієнт, Менеджер та Комплектувальник замовлень.

2.2.1. Сценарій базового виконання робочого процесу

Перший етап верифікації зосереджений на встановленні та перевірці базового лінійного потоку управління (control flow), де переходи між діями відбуваються послідовно без використання логічних розгалужень або паралельних ділянок.

Сценарій передбачає таку послідовність операцій:

- ініціалізація замовлення клієнтом.
- верифікація та затвердження замовлення менеджером.
- формування замовлення комплектувальником.
- фінальна перевірка та відправлення замовлення.

Основним завданням даного етапу є апробація механізмів інтеграції двигуна робочих процесів (workflow engine) у користувацький інтерфейс згенерованого прототипу. Графічне представлення UML-моделі для цього сценарію наведено на рисунку 13.

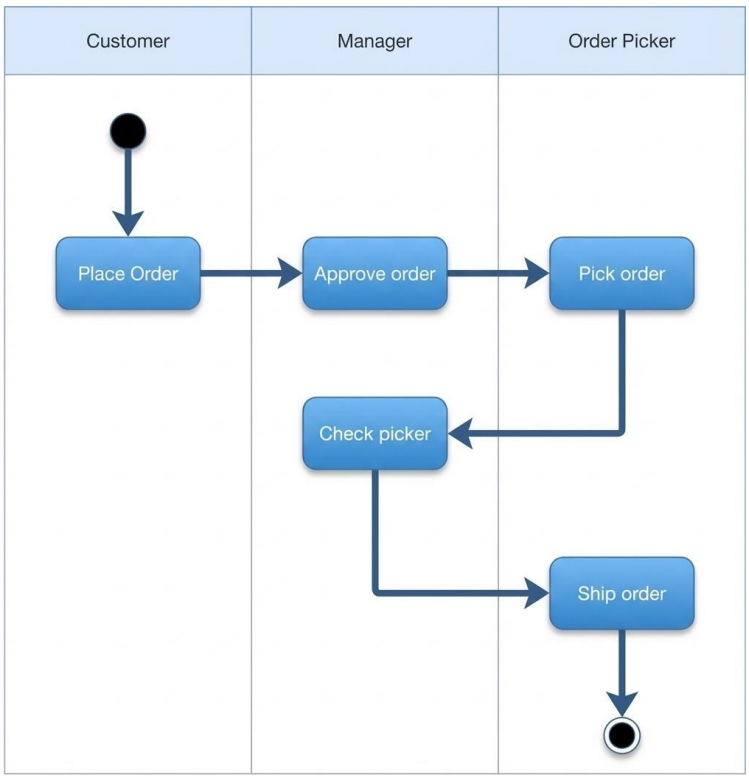

Рисунок 2.1 – UML-модель сценарію обробки замовлення (сценарій 1)

2.2.2. Сценарій умовної маршрутизації та паралельного виконання

Другий сценарій спрямований на розширення функціональних можливостей системи шляхом впровадження механізмів прийняття рішень та паралелізму. У цьому контексті переходи між вузлами діяльності стають детермінованими та залежать від специфічних умов або виборів, зроблених суб'єктами процесу.

Ключові аспекти реалізації включають:

1. Логічні розгалуження. У разі відхилення замовлення ініціюється автоматичне надсилання електронного повідомлення про скасування, що веде до термінації процесу.

2. Паралельне виконання. Після затвердження замовлення процеси комплектування та підготовки логістичної документації (створення відправлення) виконуються одночасно в окремих потоках.

3. Циклічні залежності. У випадку незадовільного результату перевірки замовлення передбачено механізм повернення процесу до етапу комплектації

для виправлення помилок.

Основним викликом даного етапу є забезпечення коректної синхронізації кількох одночасно активних задач. Для моделювання цього сценарію було використано дві спеціалізовані UML-діаграми діяльності, представлені на рисунках 2.2 та 2.3.

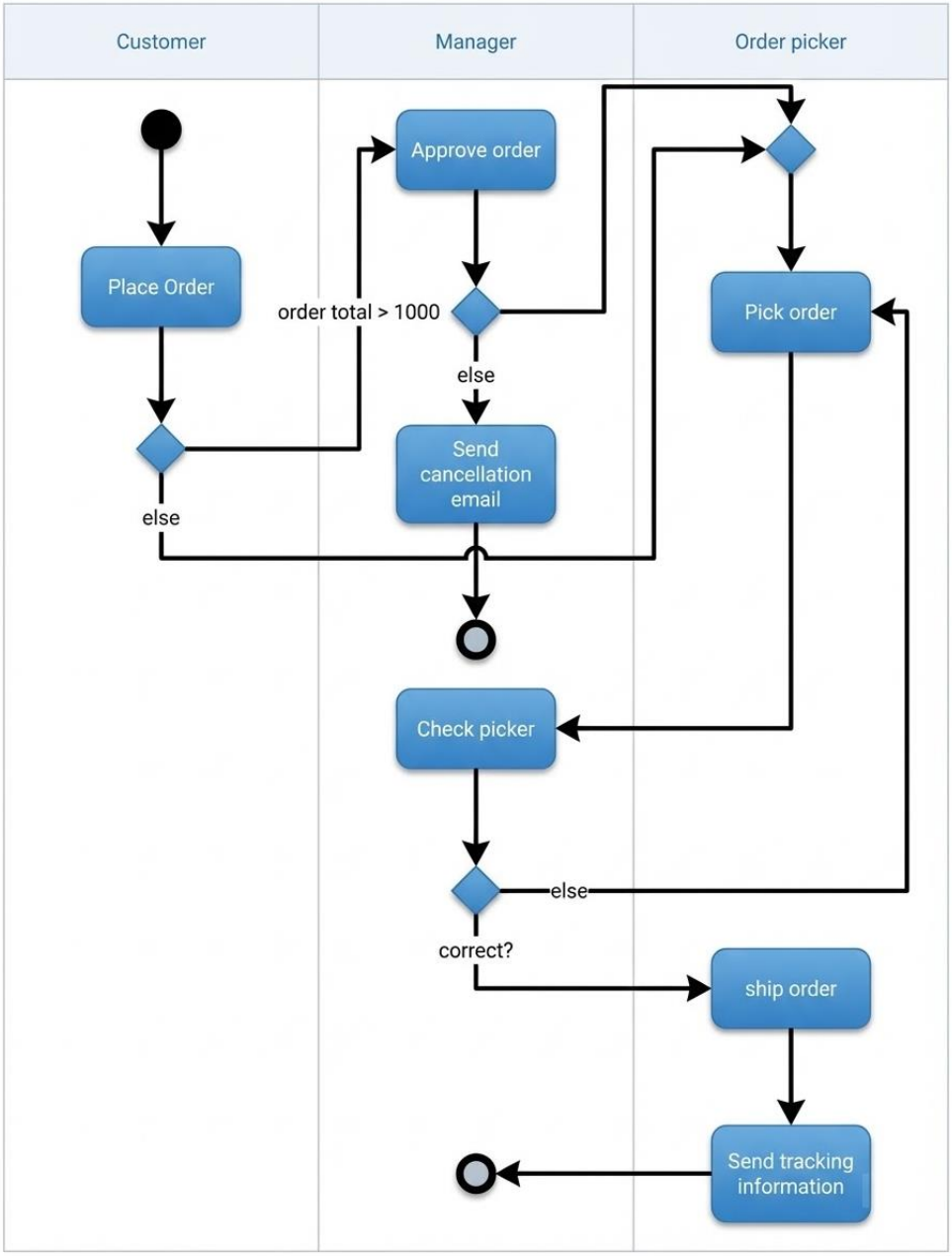

Рисунок 2.2 - UML-модель сценарію обробки замовлення (сценарій 2)

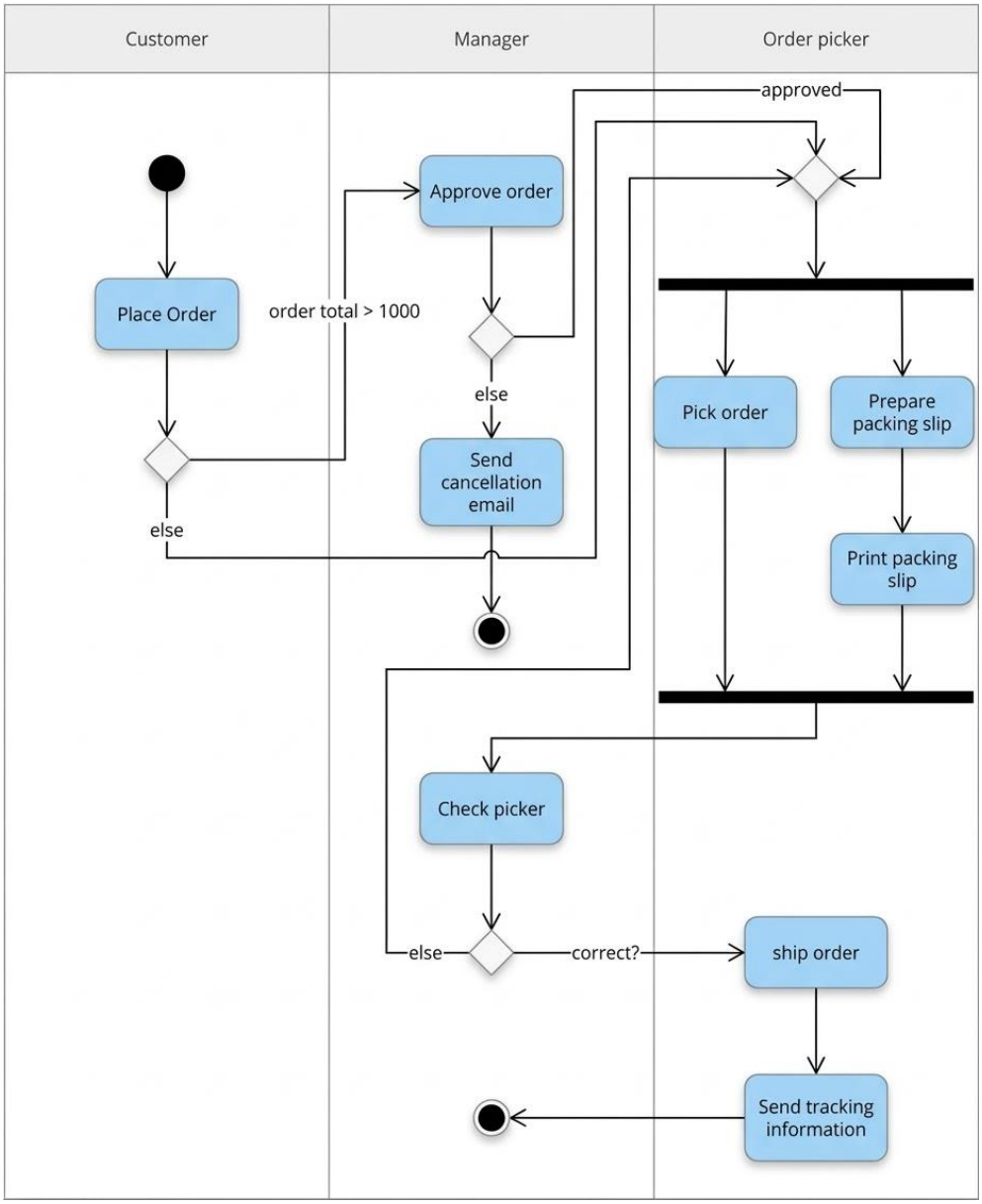

Рисунок 2.3 - UML-модель сценарію обробки замовлення (сценарій 2)

2.3. Проектування архітектури механізму робочих процесів

Для забезпечення генерації програмних застосунків із підтримкою функціоналу робочих процесів необхідна розробка та імплементація спеціалізованого програмного модуля — механізму робочих процесів (двигуна), з яким базовий застосунок зможе здійснювати взаємодію. Розроблюваний прототип системи повинен забезпечувати можливість ініціалізації нових примірників процесів, а також керування переходами між

етапами виконання поточних активностей за допомогою зазначеного механізму. Крім того, архітектурою має бути передбачено функціонал для зчитування стану активних процесів та моніторингу прогресу їх виконання.

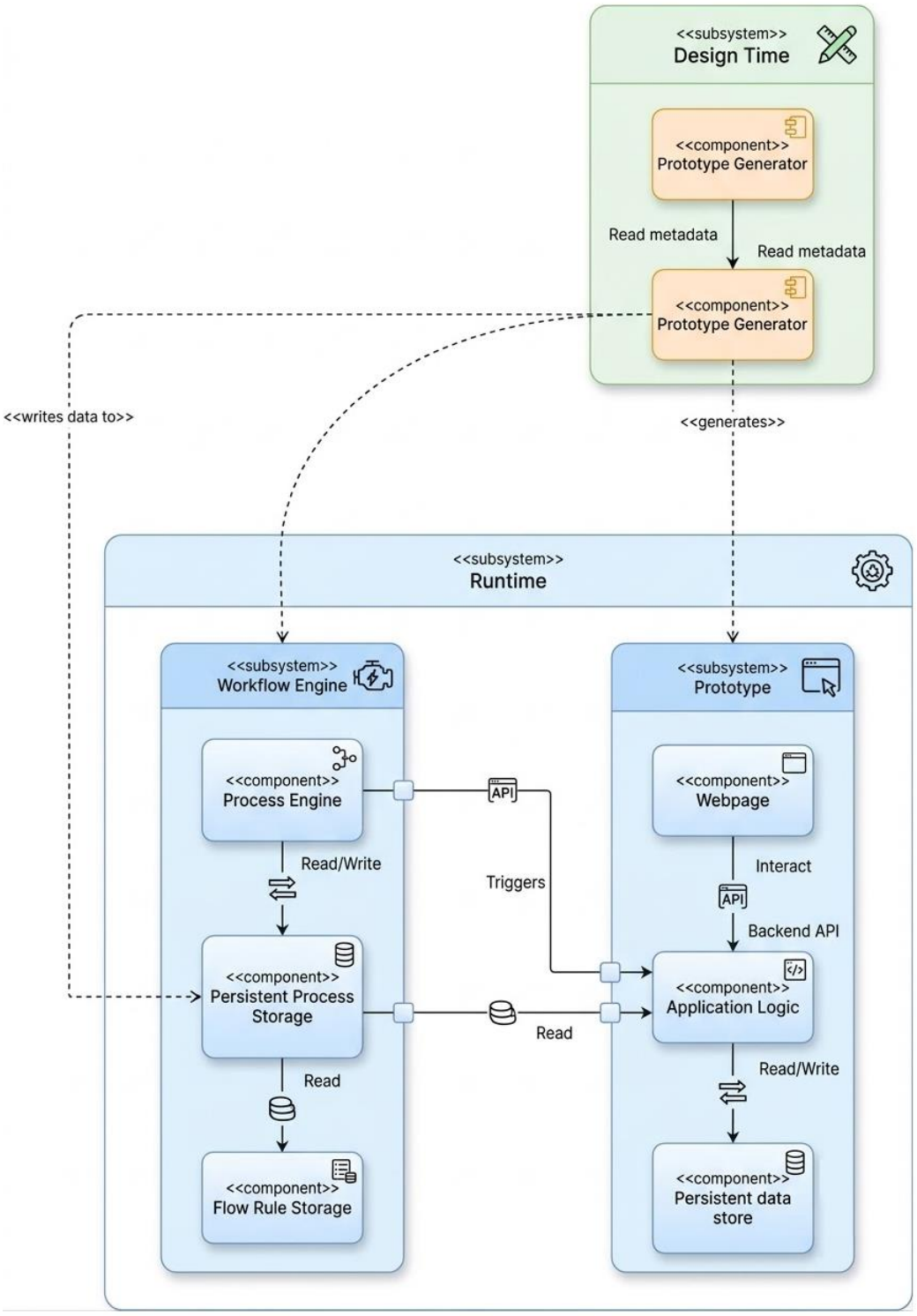

Рисунок 2.4 - Діаграма компонентів інтегрованої системи

Загальна схема інтеграції механізму робочих процесів у структуру базової системи відображена на діаграмі компонентів (рисунок 2.4). Детальний опис інтерфейсів взаємодії між двома підсистемами наведено в наступному підрозділі. Методика формування моделей робочих процесів та пов'язаних із ними структур даних розглядається нижче.

Як показано на рисунку 2.4, функціонування механізму робочих процесів базується на використанні підсистем персистентного зберігання примірників процесів та зберігання правил керування потоками виконання. Зазначені підсистеми забезпечують зберігання даних та відстеження станів процесів. У запропонованій архітектурі вихідні метадані середовища моделювання не використовуються безпосередньо. Замість цього передбачено процедуру екстракції метаданих та їхнього відображення (трансформації) у спеціалізований виконуваний формат, як проілюстровано на рисунку 2.5.

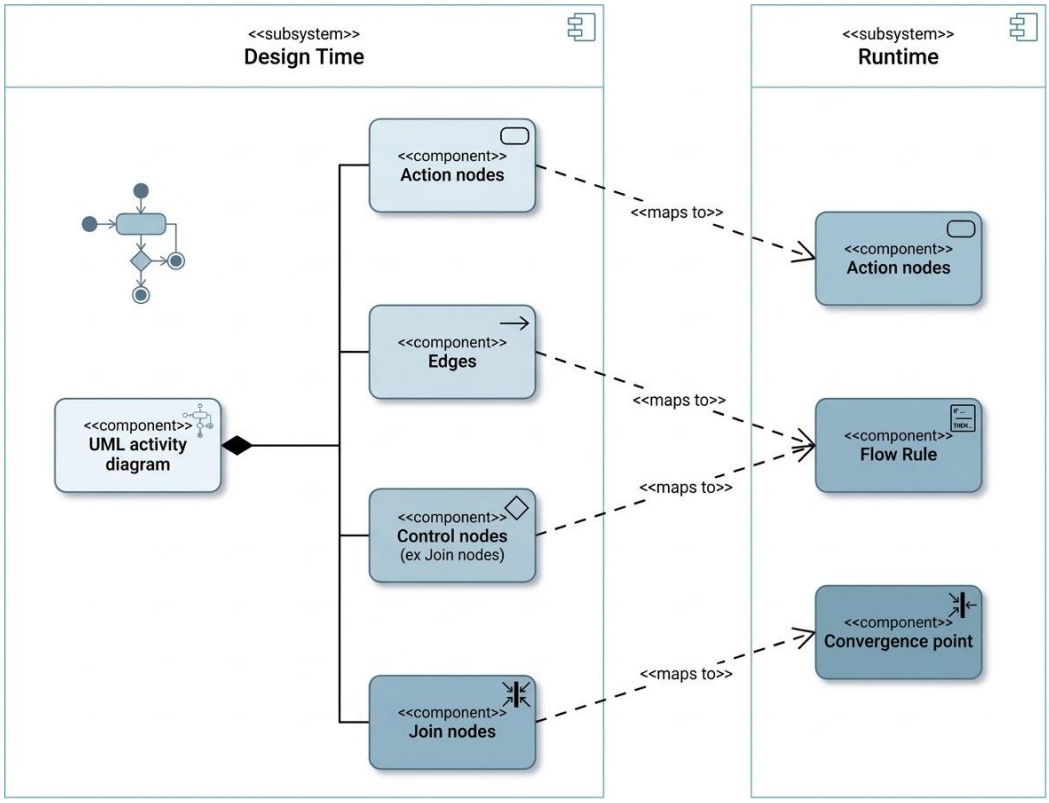

Рисунок 2.5 – Процес трансформації (відображення) метаданих

Трансформація вихідних моделей у проміжний формат, окрім архітектурних міркувань, забезпечує наступні переваги:

1. Можливості аналітики та моніторингу через застосування адаптованого формату дозволяє здійснювати автоматизований збір та аналіз даних щодо виконуваних і завершених процесів у режимі реального часу. На основі цих даних стає можливою оптимізація бізнес-процесів, що сприяє підвищенню ефективності функціонування системи та покращенню якості прийняття управлінських рішень.

2. Оптимізація швидкодії на етапі виконання, а саме пряме використання вихідних метаданих механізмом робочих процесів вимагало б повної інтерпретації та верифікації моделі на кожному кроці виконання процесу. Запропонований підхід із трансформацією моделей переносить обчислювально витратні операції з оцінювання структури процесу на етап генерації системи. У результаті, під час виконання механізм оперує лише релевантними для поточного етапу даними, що суттєво прискорює швидкість просування за етапами робочого процесу.

2.4. Методологія генерації програмних прототипів

Трансформація метаданих діаграм діяльності, отриманих із моделювального середовища, у структурований формат, придатний для обробки двигуном робочих процесів (workflow engine), є критичним етапом циклу генерації. Метадані є машинним відображенням візуальних UML-моделей, які, попри високу ефективність для концептуального проектування, не є безпосередньо виконуваними. У цьому розділі розглядається механізм відображення (mapping) метаданих у попередньо детермінований формат, що забезпечує коректну інтерпретацію та виконання змодельованих активностей програмним двигуном.

2.4.1. Архітектурна структура двигуна робочих процесів

Двигун робочих процесів функціонує як системний компонент, відповідальний за персистентне зберігання визначених процесів та моніторинг станів їх виконання. Для управління цими операціями архітектура системи використовує об'єктно-орієнтовану модель, що складається з восьми базових класів, які інтегруються у вихідний код на етапі генерації:

1. Process - агрегатор усіх вузлів дій, що складають цілісну модель. Містить посилання на початковий вузол як точку входу в алгоритм.

2. Action Node - представляє атомарну операцію або функціональне завдання в межах процесу.

3. Flow Rule - детермінує логічні умови, за яких відбувається перехід між вузлами. Детальна специфікація наведена в наступному підрозділі.

4. Active Process - забезпечує динамічне відстеження примірників процесів, що перебувають у стані виконання, та контролює їхній прогрес через асоційовані активні вузли.

5. Associated Model - абстрактний інтерфейс для зв’язку процесу з доменними об’єктами системи.

6. Associated Model Instance - зберігає унікальний ідентифікатор конкретного об'єкта даних, з яким взаємодіє активний процес.

7. Convergence Point - визначає статичні параметри синхронізації, де декілька паралельних потоків управління об'єднуються в єдину магістраль.

8. Active Convergence Point - контролює стан виконання умов синхронізації в режимі реального часу.

Взаємозв'язки між зазначеними класами ілюструються на рисунку рисунку 2.6. Відповідно до функціонального призначення, класи розподіляються на дві категорії: компоненти зберігання (позначені синім) та компоненти виконання (позначені червоним). Дана диференціація представлена на рисунку 2.7.

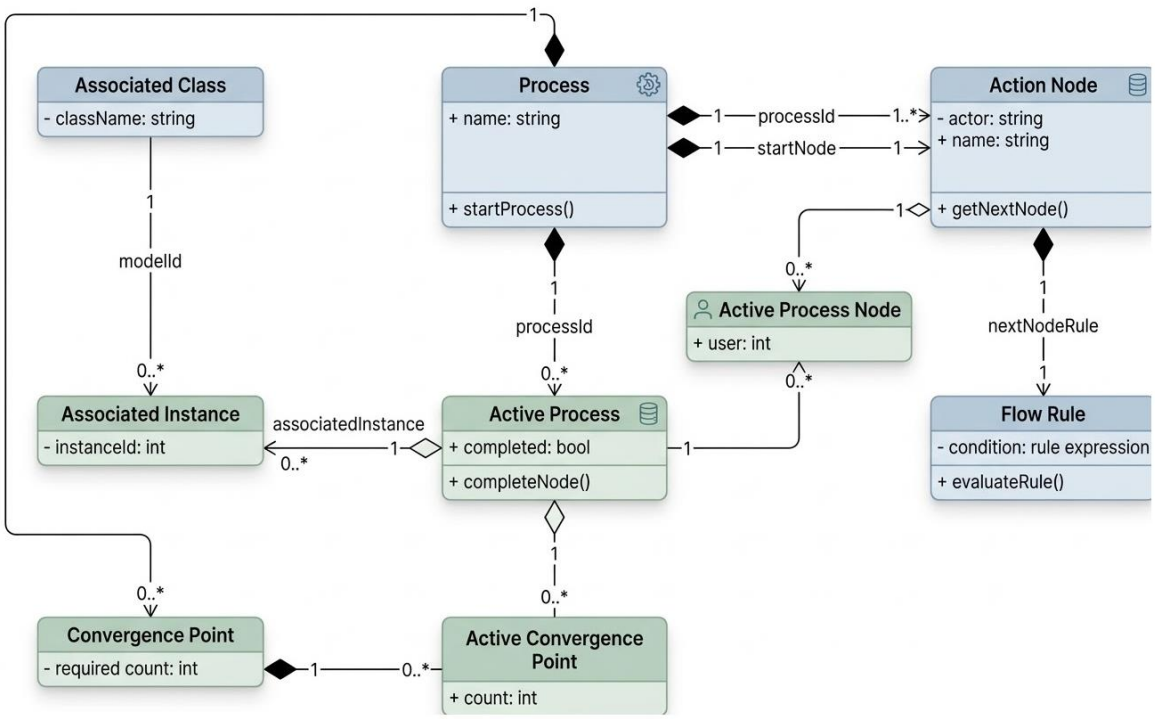

Рисунок 2.6 - Діаграма класів архітектури двигуна робочих процесів

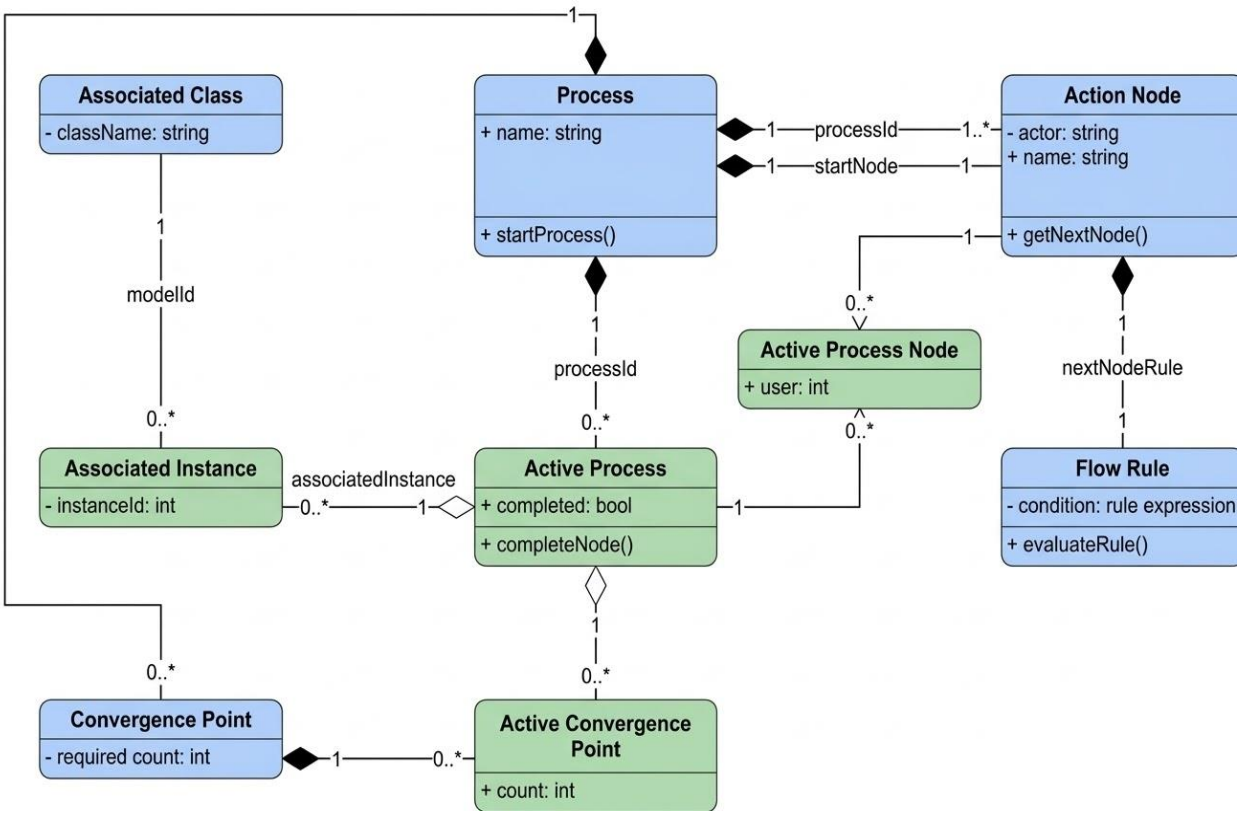

Рисунок 2.7 - Функціональний поділ компонентів між підсистемами зберігання (сині блоки) та виконання (зелені блоки)

2.4.2. Процедура відображення метаданих

Після визначення архітектурної структури здійснюється трансформація вихідних метаданих діаграми діяльності у структури підсистеми зберігання (синій блок на рисунку 2.7).

Процес розпочинається з ініціалізації асоційованих моделей на основі статичних діаграм класів.

Алгоритм відображення виконується ітеративно для кожної моделі діяльності:

1. Створюється запис у класі процес.

2. Здійснюється перенос вузлів дій із метаданих до відповідного класу системи.

3. Вузли управління (крім вузлів об'єднання) обробляються опосередковано: вони не створюють окремих дій, а трансформуються у логіку переходів.

4. Для вузлів об'єднання (Join nodes) розраховується кількість вхідних ребер та створюється об'єкт у класі Точка збіжності.

5. Правила потоку формуються рекурсивним обходом графів: для кожного вузла дії створюється правило, цільовим об'єктом якого стає наступний логічний вузол. Якщо на шляху зустрічається вузол управління, правило розширюється відповідно до його типу.

Результатом виконання даного алгоритму є генерація повнофункціональної моделі процесу, де кожен крок відповідає проектній документації UML і керується правилами, що враховують контекст виконання. Загальна схема трансформації представлена на рисунку 2.8.

Наведена схема класів UML ілюструє концептуальну архітектуру системи автоматизованої трансформації візуальних моделей діяльності UML у виконувані програмні компоненти. Вона чітко розмежовує два функціональні рівні: рівень метамоделювання (Design Time) та рівень виконання (Runtime).

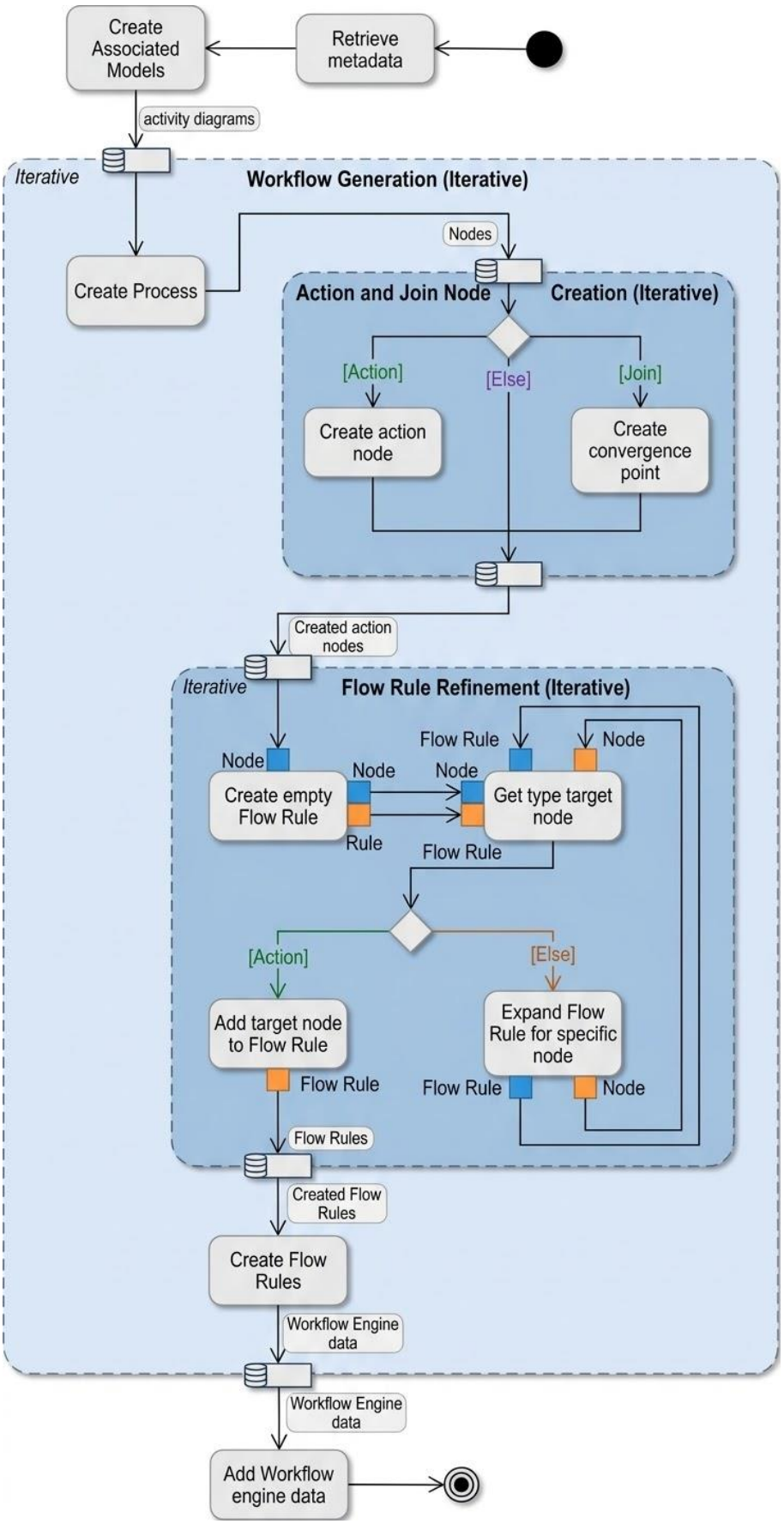

Рисунок 2.8. - Діаграма активностей процесу відображення мета даних

На рівні Design Time ключовим компонентом виступає клас <<component>> UML activity diagram. Цей компонент є центральним агрегатором (зв’язок спільної агрегації з боку Activity Diagram) для набору структурних елементів, що формально визначають логіку бізнес-процесу. До цих агрегованих елементів належать:

- Action nodes - класи, що моделюють окремі виконувані завдання або етапи діяльності.

- Edges - класи, що описують переходи та потоки управління між вузлами.

- Control nodes - класи, що відповідають за логіку розгалуження потоків (наприклад, логічні рішення).

- Join nodes: класи, які є специфічними вузлами управління для синхронізації та об'єднання паралельних потоків виконання.

Центральним архітектурним елементом, що пов’язує два рівні, є залежності <<maps to>>. Ці зв’язки відображають процес автоматизованої генерації (трансформації) метаданих з описового рівня (Design) на виконуваний (Runtime), детермінуючи, які технічні конструкти Runtime-сервісу будуть згенеровані на основі елементів UML-моделі:

- статичні Action nodes моделі автоматично імплементуються як виконувані Action nodes середовища Runtime, що виконують конкретні бізнес-завдання.

- Логіка переходів (Edges) та вузлів управління (Control nodes) трансформується у набір правил виконання (Flow Rule). Це перетворює візуальні стрілки та ромби в умовну логіку (як-от IF...THEN), що автоматично керує потоком застосунку.

- Точки об'єднання паралельних процесів (Join nodes) реалізуються як точки збіжності (Convergence point), що забезпечують синхронізацію та очікування завершення всіх паралельних гілок перед переходом до наступного кроку.

Таким чином, на рівні Runtime формується трикомпонентна виконувана архітектура, що є результатом трансформації: виконувані Action nodes, сервіс Flow Rule для управління логікою потоків та контролер Convergence point для синхронізації паралелізму. Дана схема демонструє цілісний підхід до Model-Driven розробки (MDD), де чітке розділення на описову та виконувану частини дозволяє автоматизувати створення складного програмного забезпечення на основі бізнес-моделей, забезпечуючи високий рівень абстракції та автоматизовану реалізацію технічних деталей застосунку.

2.4.3. Операційне виконання в режимі реального часу

Двигун робочих процесів виступає центральним координатором, відповідальним за управління життєвим циклом кожного примірника процесу. Його основна функція полягає у забезпеченні консистентного просування за етапами алгоритму відповідно до формалізованих правил.

Ключові аспекти функціонування в режимі runtime:

1. Ініціалізація та активація.

При старті процесу двигун ідентифікує початкову точку та активує відповідний робочий потік.

2. Управління контекстом.

Система безперервно актуалізує контекст процесу, пов'язуючи динамічні дані, генеровані користувачем (наприклад, параметри замовлення), із поточним станом виконання.

3. Динамічна інтерпретація.

Контекстні дані використовуються для обчислення логічних умов у правилах потоку, що дозволяє динамічно визначати наступний крок.

4. Адаптивність інтерфейсу.

Двигун обмежує видимість даних, забезпечуючи відображення лише тієї інформації, що є релевантною для поточного стану процесу.

5. Синхронізація паралелізму.

Механізм контролює точки збіжності, блокуючи подальше виконання до завершення всіх необхідних паралельних гілок.

Процес вважається завершеним при досягненні кінцевого вузла за умови відсутності активних паралельних задач. Детальна специфікація взаємодії компонентів під час виконання буде наведена в наступному розділі з посиланням на архітектурну схему на рисунку 2.6.

2.5 Висновки до розділу

У другому розділі розроблено методологічні та алгоритмічні засади побудови системи генерації прототипів програмного забезпечення на основі модельно-орієнтованого підходу. Запропонована методологія дизайн-дослідження забезпечила послідовне формування архітектури системи з урахуванням практичних сценаріїв її використання. Проведена верифікація архітектурних рішень на основі сценаріїв підтвердила їхню коректність та здатність підтримувати складні робочі процеси. Сформовано архітектуру механізму керування робочими процесами, яка забезпечує гнучке управління станами, переходами та логічними умовами виконання. У результаті обґрунтовано ефективність запропонованих підходів для реалізації динамічних систем генерації програмних прототипів.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ МЕТОДІВ ГЕНЕРАЦІЇ ПРОТОТИПІВ ПРОГРАМНИХ СИСТЕМ

3.1. Специфікація розширень метаданих для генерації програмних систем на основі UML-моделей

Для забезпечення повноцінної генерації застосунків, що підтримують складні робочі процеси (workflows), існуючий інструментарій візуального моделювання потребує розширення атрибутивного складу метаданих. У межах даної методології метадані визначаються як структуроване представлення елементів UML-діаграм та специфікація інтерфейсів, необхідних для трансформації моделей у виконуваний код (наприклад, на базі фреймворку Django).

1. Рольова модель та детермінація відповідальності

Реальні операційні процеси передбачають участь декількох суб'єктів, що обумовлює необхідність закріплення дій за конкретними ролями користувачів. Відповідно до стандартів UML, для візуалізації розподілу відповідальності використовуються доріжки (swimlanes). Для підтримки цієї функціональності метамодель діаграми діяльності розширюється атрибутом actorNode. Даний атрибут виконує роль посилання на вузол актора, визначений у діаграмі варіантів використання, що дозволяє встановити однозначну відповідність між дією в робочому процесі та правами доступу конкретної ролі.

2. Автоматизація системних операцій

Сучасні бізнес-процеси характеризуються високим рівнем автоматизації, де значна частина завдань (наприклад, надсилання повідомлень або взаємодія з зовнішніми API) виконується без участі оператора. Для підтримки такої диференціації у метадані вузла дії впроваджуються наступні атрибути:

- isAutomatic: логічний прапорець, що визначає тип виконання дії (ручний або системний).

- customCode: атрибут, що містить виконуваний блок коду (наприклад, мовою Python), який ініціюється двигуном робочих процесів при досягненні відповідного вузла.

Для реалізації часових тригерів у метадані початкового вузла інтегровано атрибути scheduled та schedule. Останній використовує вирази CRON для налаштування автоматичного запуску процесів за розкладом, що дозволяє інтегрувати систему з планувальниками завдань (наприклад, пакет django-crontab).

3. Контекстно-залежна генерація інтерфейсів користувача

Традиційні підходи до генерації інтерфейсів часто надають глобальний доступ до екземплярів даних, що може бути надлишковим у межах конкретного кроку процесу. Для створення інтерфейсів, що враховують контекст виконання, схема метаданих сторінки доповнюється атрибутом type із класіфікацією:

- Normal: сторінки загального доступу до даних.

- Activity: контекстно-залежні сторінки, що обмежують видимість об'єктів лише тими екземплярами, які безпосередньо пов'язані з поточною активністю.

Це дозволяє на етапі генерації адаптувати представлення (Views), маршрути (URLs) та шаблони (Templates) під специфічні потреби поточного стану процесу.

4. Адміністрування та управління доступом

Управління життєвим циклом процесів потребує спеціалізованих інтерфейсів, доступ до яких має бути обмежений групою користувачів із підвищеними повноваженнями. Для цього до метаданих інтерфейсу додається атрибут managerAccess. Наявність даного прапорця детермінує права актора на перегляд та маніпулювання адміністративними сторінками

управління.

5. Формалізація умовної логіки та потоків управління

Для забезпечення коректного розгалуження процесів у вузлах прийняття рішень (Decision Nodes) необхідно мати інструментарій визначення умов переходу. Метадані ребер потоку управління розширюються двома параметрами:

- condition: формалізована умова переходу.

- isElse: атрибут, що позначає альтернативний шлях виконання, якщо жодна з основних умов не є істинною.

6. Оптимізація візуальної ергономіки моделей

Зі збільшенням складності діаграм виникає потреба в точнішому управлінні геометрією зв'язків для уникнення перетинів та накладань. Метадані ребер доповнюються обробниками позиціонування (PositionHandlers). Ці компоненти фіксують координати проміжних точок, дозволяючи формувати ламані лінії та кутові з'єднання, що суттєво підвищує когнітивну зрозумілість візуальних моделей.

3.2. Функціональна специфікація інтерфейсів користувача у системах управління робочими процесами

Реалізація взаємодії між механізмом робочих процесів та кінцевим користувачем здійснюється через набір спеціалізованих інтерфейсних елементів, що забезпечують повний життєвий цикл операційної діяльності.

1. Централізована панель керування

Після автентифікації в системі користувачу надається доступ до головної сторінки (Dashboard), яка виконує роль інтегрованого вузла управління. Функціонал цієї сторінки розширено для відображення двох критичних інформаційних масивів:

- Реєстр доступних процесів - перелік бізнес-процесів, ініціацію яких

може здійснити поточний користувач згідно з його правами доступу.

- Черга активних завдань - перелік операцій, що очікують на виконання у межах поточних екземплярів процесів та потребують безпосередньої уваги користувача.

Активація елемента в будь-якому з перелічених списків ініціює маршрутизацію користувача на відповідну контекстно-залежну сторінку активності для виконання конкретного завдання. Візуальна структура даного інтерфейсу представлена на рисунку 3.1.

Рисунок 3.1 - Головна сторінка інтерфейсу програмного прототипу

2. Інтерфейс виконання операційних завдань

Сторінка активності призначена для безпосередньої реалізації кроку робочого процесу. Після завершення необхідних маніпуляцій з даними користувач здійснює фіналізацію завдання за допомогою елемента управління «Завершити завдання» (розташованого у нижньому правому сегменті інтерфейсу).

Ця дія виконує роль тригера, який позначає операцію як виконану в базі даних та ініціює перехід механізму робочих процесів до наступного вузла згідно з визначеною логікою (Flow Rules). Типовий приклад структури такої сторінки наведено на рисунку 3.2.

Рисунок 3.2 - Спеціалізована сторінка виконання активності

3. Адміністративні інтерфейси моніторингу та управління

Для забезпечення гнучкості управління ресурсами та прозорості виконання процесів у систему інтегровано додаткові модулі, орієнтовані на користувачів із правами адміністратора або менеджера:

- Модуль аудиту та журналювання - сторінка журналу дій забезпечує комплексний моніторинг усіх подій у системі. Вона дозволяє відстежувати часові мітки завершення завдань, ідентифікувати поточний стан активних процесів та аналізувати історію змін.

- Модуль делегування та перепризначення. Даний інтерфейс дозволяє динамічно змінювати виконавців для активних операцій, що забезпечує адаптивність системи до змін у розподілі трудових ресурсів або виникнення позаштатних ситуацій.

Відповідні програмні реалізації цих модулів продемонстровані на рисунках 3.3 та 3.4.

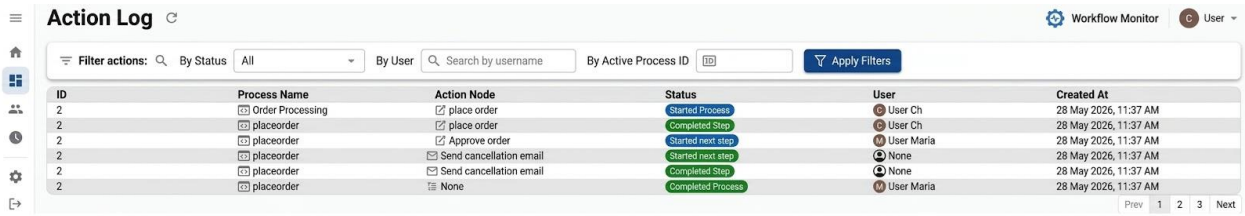

Рисунок 3.3 - Інтерфейс системного журналу подій та аудиту активностей

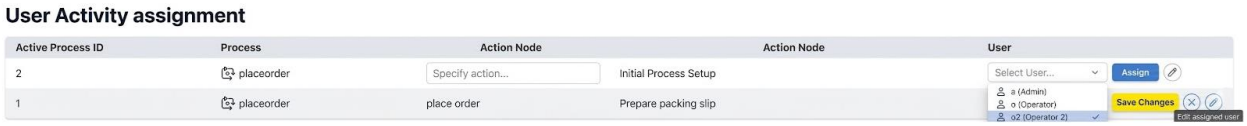

Рисунок 3.4 - Модуль управління призначеннями та розподілу ролей користувачів

3.3. Реалізація архітектури та механізмів заповнення даних модуля керування робочими процесами

У цьому розділі наведено детальну специфікацію технічної реалізації класів, концептуально визначених на рисунку 2.6. Також описано методологію використання метаданих діаграм діяльності UML для автоматизованого заповнення відповідних структур даних.

Для централізованого управління зберіганням моделей робочих процесів та моніторингу стану їх виконання в архітектурі системи передбачено створення виділеного модуля (додатка) фреймворку Django. Таке архітектурне рішення забезпечує універсальний доступ до функціоналу механізму робочих процесів з боку будь-яких периферійних модулів системи, специфічних для конкретних акторів.

3.3.1. Технічна імплементація класів

Переважна більшість проектованих класів реалізується у файлі дефініції моделей модуля робочих процесів у вигляді стандартних моделей Django ORM. У ході технічної реалізації було встановлено недоцільність прямої імплементації класу Асоційована модель. Його функціонал ефективно заміщується стандартним фреймворком ContentType [15], який забезпечує універсальний інтерфейс для роботи з усіма моделями, встановленими у проекті. Завдяки використанню механізму GenericForeignKey, що надається цим фреймворком, екземпляр класу ActiveProcess може бути динамічно пов'язаний з будь-якою іншою моделлю даних без необхідності попереднього

статичного визначення відносин на рівні схеми бази даних.

Додатково до базової структури було впроваджено клас ActionLog. Його призначення полягає у веденні детального журналу дій користувачів та автоматизованих системних подій, що відбуваються під час виконання примірника процесу. Остаточна технічна діаграма класів реалізованого модуля представлена на рисунку 3.5.

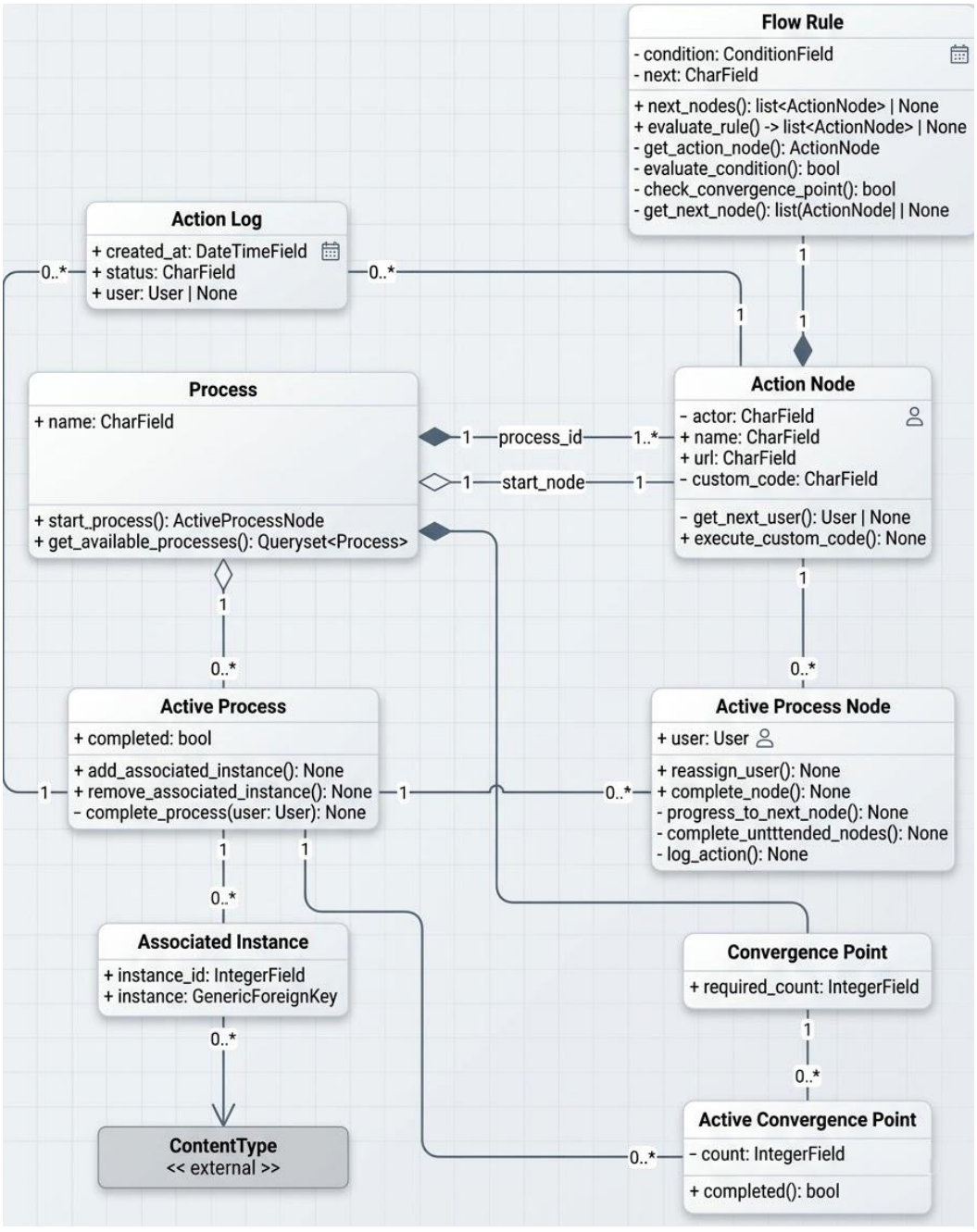

Рисунок 3.5 - Технічна діаграма класів модуля робочих процесів

Наведена технічна діаграма класів (рис. 3.5) відображає архітектуру програмної реалізації механізму робочих процесів, адаптовану для середовища розробки на базі фреймворку Django. Систему структуровано за принципом розділення статичних дефініцій процесів та їх динамічних примірників у режимі реального часу.

1. Рівень дефініцій та статичних метаданих

Фундаментом архітектури є клас Process, який слугує агрегатором для опису логіки бізнес-процесу. Він містить метадані (наприклад, name) та керує композицією вузлів через зв'язки з класом Action Node. Кожен вузол дії інкапсулює параметри конкретного етапу: роль виконавця (actor), цільову URL-адресу для інтерфейсу та блоки спеціалізованого коду (custom_code) для автоматизованих операцій.

Логіка переходів між вузлами делегована класу Flow Rule. Цей компонент відповідає за обчислення умов (evaluate_rule) та визначення наступного кроку в графі процесу. Для сценаріїв паралельного виконання передбачено клас Convergence Point, який детермінує необхідну кількість вхідних потоків (required_count) для синхронізації процесу в точці збіжжя.

2. Рівень виконання та управління станами (Runtime)

Динамічний аспект системи представлений класами з префіксом Active, що забезпечують персистентність станів для кожного окремого запуску процесу:

- Active Process представляє конкретний примірник виконуваного процесу. Він відстежує загальний статус завершення та забезпечує зв'язок із прикладними даними.

- Active Process Node фіксує поточний стан виконання конкретного вузла, асоціюючи його з реальним користувачем системи (user). Методи класу дозволяють здійснювати перепризначення виконавців, логування дій та просування за графом процесу.

- Active Convergence Point виконує роль лічильника завершених

паралельних гілок, блокуючи виконання до досягнення цільового показника, встановленого у статичній дефініції.

3. Механізми інтеграції та аудиту

Особливістю даної архітектури є використання класу Associated Instance у поєднанні із зовнішнім модулем ContentType. Це реалізує паттерн «Generic Foreign Key», що дозволяє динамічно пов'язувати процеси з будь-якими бізнес-сутностями системи без жорсткої фіксації зв'язків на рівні схеми бази даних. Така гнучкість є критичною для універсальних workflow-рішень.

Для забезпечення прозорості та можливості ретроспективного аналізу впроваджено клас Action Log. Він здійснює автоматичну реєстрацію всіх подій (зміна статусів, завершення кроків, системні помилки) із прив'язкою до часових міток та конкретних користувачів.

3.3.2. Алгоритмічний протокол взаємодії компонентів при переході між етапами робочого процесу

Процес переходу від однієї активності до іншої в межах двигуна робочих процесів базується на динамічній взаємодії між об'єктами стану (Active Process Node) та об'єктами логіки (Flow Rule). Цей механізм забезпечує детермінованість виконання бізнес-логіки, закладеної в UML-моделі.

Нижче наведено покроковий опис алгоритму ініціації та виконання переходу:

1. Тригеризація завершення поточної активності

Процес активується, коли користувач або автоматизована система викликає метод complete_node() у класі Active Process Node.

- Система фіксує часову мітку завершення в Action Log.

- Об'єкт Active Process Node звертається до свого статичного прототипу — класу Action Node, щоб отримати асоційоване з ним Flow Rule (правило

потоку).

2. Евалюація логічних умов (Evaluation Phase)

На цьому етапі керування передається об'єкту класу Flow Rule. Виконується метод evaluate_rule(), який працює за наступним принципом:

- Аналіз контексту. Правило зчитує поточні дані процесу через Associated Instance.

- Обчислення виразів. Перевіряються логічні умови, задані в атрибуті condition. Якщо умова істинна (або активовано шлях else), визначається ідентифікатор наступного цільового вузла.

- Рекурсивне розширення. Якщо цільовим об'єктом є вузол управління (наприклад, Decision Node), правило рекурсивно обчислює наступні сегменти графа, доки не досягне наступного об'єкта типу Action Node або термінального вузла.

3. Синхронізація та обробка точок збіжності

Якщо наступним кроком у моделі є точка збіжжя (Join Node), алгоритм взаємодіє з компонентами синхронізації:

- Викликається метод check_convergence_point().

- Об'єкт Active Convergence Point інкрементує лічильник завершених вхідних потоків (count).

- Перехід до наступної дії блокується, доки count не зрівняється з required_count. Тільки після виконання цієї умови ініціюється створення наступної активності.

4. Ініціалізація нового стану виконання

Після успішного визначення наступного кроку механізм робочих процесів виконує фінальні операції:

- Створення нового вузла. Генерується новий екземпляр Active Process Node для наступної дії.

- Призначення виконавця. Викликається метод get_next_user() для ідентифікації відповідального актора на основі ролі, визначеної в метаданих.

- Активація системних тригерів. Якщо дію позначено як автоматичну (isAutomatic), негайно запускається метод execute_custom_code().

Описана архітектурна взаємодія дозволяє відокремити жорстку логіку переходів від динамічного стану системи. Клас Flow Rule виступає «інтелектуальним центром» системи, тоді як Active Process Node забезпечує персистентність та контекст виконання, що гарантує цілісність бізнес-процесу навіть за умов високої складності та паралелізму.

3.3.3. Методика заповнення даних механізму робочих процесів

Процес ініціалізації даних механізму робочих процесів базується на аналізі метаданих діаграм діяльності, які первинно подаються у форматі JSON. На першому етапі здійснюється десеріалізація JSON-даних та їх перетворення у графову структуру. Ця структура релевантно відображає взаємозв'язки між вузлами та переходами, що є оптимальним для подальшої алгоритмічної обробки.

Алгоритм аналізу передбачає ітеративну обробку кожної специфікації діаграми діяльності. Точкою входу для рекурсивного обходу графа слугує початковий вузол. Процес обходу фіксує всі переходи між вузлами, вибудовуючи цілісну топологію потоку управління. Рекурсивний алгоритм завершується за виконання однієї з наступних умов:

- Досягнення термінального стану. Виявлення кінцевого вузла або вузла, який не має вихідних ребер (дуг).

- Запобігання циклічності. Виявлення вузла, який вже був відвіданий під час поточного обходу. Це забезпечує коректну обробку циклів у процесах та вузлів логічного злиття (merge/join), запобігаючи повторній обробці одних і тих самих сегментів графа.

Після формування повної графової структури здійснюється її трансформація у персистентні записи бази даних. Послідовність створення записів є критичною: спочатку формується запис для об'єкта Процес, що

необхідно для забезпечення цілісності посилань (зовнішніх ключів) у наступних записах. Вузли дій створюються шляхом рекурсивного обходу графа з ігноруванням вузлів управління.

Наступним етапом є генерація правил потоку, що детермінують переходи між створеними вузлами дій. Формування логічних умов відбувається шляхом рекурсивного аналізу вихідних потоків управління вузлів. Рекурсія розширення умови триває до моменту досягнення наступного вузла дії або термінального вузла.

Результатом описаного процесу для всієї сукупності діаграм діяльності є формування результуючого JSON-об'єкта, що містить повний набір записів для ініціалізації механізму робочих процесів. Цей об'єкт разом із відповідним файлом міграції інтегрується у структуру прототипу, забезпечуючи автоматизоване заповнення бази даних релевантними процесовими даними під час розгортання системи.

3.3.4. Оптимізація генерації об'єктів представлення

У ході розширення підсистеми генерації переглядів для підтримки функціоналу робочих процесів було виявлено недоліки попередніх методів, зокрема значне дублювання коду та функціоналу. Наприклад, для операції створення екземпляра моделі генерувалася надлишкова кількість окремих переглядів (для відображення форми, для обробки даних, для контролю відображення форми). Аналогічна ситуація спостерігалася при генерації операцій оновлення та виклику спеціальних методів моделей.

Дана проблема набула особливої актуальності при інтеграції переглядів, специфічних для конкретних активностей робочого процесу. Для таких представлень критично важливим є застосування контекстної фільтрації: вони повинні повертати не повну сукупність екземплярів моделі, а лише ті, що асоційовані з поточною активністю. Реалізація такої фільтрації за умов дублювання коду вимагала б внесення змін у велику кількість

ізольованих дефініцій переглядів.

Для вирішення цієї проблеми парадигму генерації переглядів було змінено на користь використання переглядів на основі класів (Class-Based Views, CBV) Django [20].

Застосування CBV дозволило інкапсулювати загальну логіку (наприклад, управління контекстом відображення форм або застосування фільтрів) у базових класах. Це забезпечило можливість успадкування функціоналу та скоротило кількість необхідних переглядів для однієї сторінки до максимум двох, незалежно від складності бізнес-логіки. Також було задіяно механізм загальних переглядів (generic views), що додатково мінімізувало обсяг коду. Задля забезпечення повторного використання базові класи переглядів було імплементовано у загальному модулі моделей (shared_models).

3.3.5. Впровадження обов'язкової аутентифікації

Інтеграція механізмів робочих процесів в архітектуру генерації системи зумовила перехід від необов'язкової до обов'язкової аутентифікації користувачів. Це обумовлено введенням функціоналу призначення завдань конкретним суб'єктам, що вимагає надійної ідентифікації користувача.

Призначення завдань персоніфікованим користувачам, а не абстрактним ролям, є необхідним для забезпечення підзвітності та раціонального розподілу навантаження. Відсутність механізму аутентифікації перетворила б управління робочими процесами на непрозору систему («чорний ящик»), унеможливлюючи моніторинг відповідальності за виконання конкретних етапів процесу.

3.4. Функціональна логіка операційного циклу механізму керування робочими процесами в режимі реального часу

Процес функціонування механізму керування робочими процесами в режимі реального часу (runtime) базується на динамічному управлінні станами та виконанні визначених бізнес-алгоритмів. Технічна реалізація цього процесу здійснюється шляхом маніпулювання примірниками класів підсистеми виконання (виділені зеленим кольором на рисунку 2.7).

Архітектурно механізм реалізує дві ключові точки доступу (endpoints), відповідальні за життєвий цикл процесу:

- Ініціалізація процесів забезпечує створення нового запису в класі «Активний процес» та призначення йому початкового «Активного вузла процесу», що відповідає вхідній точці, детермінованій у статичній конфігурації моделі.

- Просування виконання. Дана точка доступу приймає ідентифікатор існуючого активного вузла як вхідний параметр та ініціює евалюацію правила потоку (Flow Rule), асоційованого з відповідним вузлом дії. Усі нові вузли, отримані в результаті обчислення логічних умов, набувають статусу нових активних вузлів системи.

3.4.1. Управління операційним контекстом та зв'язування даних

Для коректного обчислення логічних умов механізм повинен мати безперервний доступ до контексту конкретного примірника процесу. Ця задача вирішується за допомогою класу «Примірник асоційованої моделі». Використовуючи механізми універсального зв'язування об'єктів (зокрема фреймворк ContentType), система забезпечує динамічну кореляцію будь-якого екземпляра моделі, створеного на певному етапі, з відповідним активним процесом. Така архітектура дозволяє правилам потоку динамічно отримувати доступ до актуальних даних для їх подальшої оцінки. Процес

зв'язування інтегрується як додатковий крок у логіку представлень (Views), що відповідають за генерацію даних у межах поточного кроку процесу.

3.4.2. Синхронізація паралельних потоків та завершення процесу

У складних моделях, що передбачають паралельне виконання завдань, активація наступного вузла можлива лише за умови повної конвергенції всіх вхідних гілок управління. Цей механізм реалізується через взаємодію класів «Точка збіжності» та «Активна точка збіжності».

- Кожного разу, коли потік управління досягає вузла збіжжя, система перевіряє його поточний статус.

- Клас моніторингу відстежує кількість вхідних сигналів, що надійшли до даної точки.

- Після того, як фактична кількість вхідних потоків збігається з очікуваною (прописаною в метаданих), етап вважається завершеним, що дозволяє подальше просування робочого процесу.

Завершальна фаза ітерації передбачає аудит активних вузлів. Якщо після оцінки всіх правил та оновлення статусів у системі не залишається жодного активного вузла, примірник процесу автоматично маркується як завершений. Концептуальна ілюстрація описаної логіки виконання наведена на рисунку 3.6.

Наведена UML-діаграма активностей (Activity Diagram) ілюструє структурований, ітеративний та паралельний процес генерації робочих процесів (workflows) на основі вхідних метаданих.

Процес охоплює етапи від початкового вилучення даних до фінального збереження згенерованої конфігурації у сховищі даних двигуна робочих процесів.

Нижче наведено детальний розповідний опис алгоритму виконання, представленого на діаграмі:

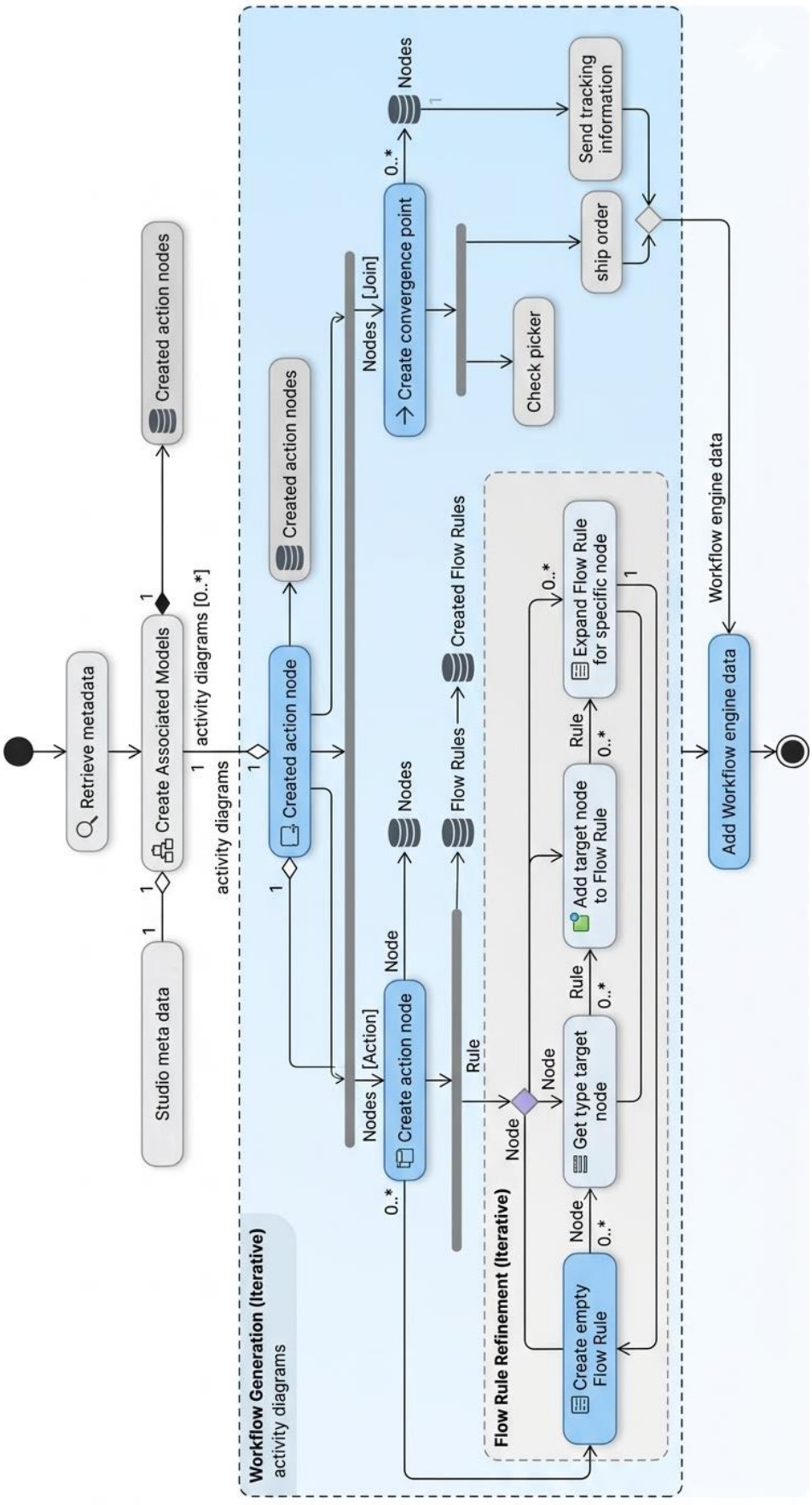

Рисунок 3.6 - Модель логіки виконання механізму робочих процесів у режимі реального часу

1. Ініціалізація та підготовка моделей

Процес розпочинається з початкового вузла (initial node) і переходить до виконання дії «Вилучення метаданих» (Retrieve metadata). На цьому етапі система отримує необхідну конфігураційну інформацію.

Наступним кроком є дія «Створення асоційованих моделей» (Create Associated Models). Ця операція є комплексною і взаємодіє з декількома сутностями:

- Вона приймає як вхідні дані об'єкт Studio meta data (у відношенні 1:1).

- Результатом її виконання є створення множини об'єктів у сховищі даних Created action nodes.

- Також формується потік об'єктів (object flow), що представляє множину діаграм активностей (activity diagrams [0..*]), які підлягають подальшій обробці.

2. Основний цикл: генерація робочого процесу

Вихідний потік діаграм активностей (activity diagrams [1]) надходить на вхід великої ітеративної області «Генерація робочого процесу (ітеративна)» (Workflow Generation (Iterative)). Ця область виконується циклічно для кожної вхідної діаграми активностей.

Логіка всередині цієї області розгалужується для обробки різних типів вузлів, що містяться в діаграмі:

- Гілка обробки вузлів дій (Actions): Контрольний потік проходить через вузол прийняття рішення (decision node) за умовою [Action]. Виконується дія «Створення вузла дії» (Create action node). Ця дія використовує дані з потоку Nodes, що надходить на вхід ітеративної області, і оновлює сховище даних Created action nodes.

- Гілка обробки вузлів об'єднання (Joins): За умовою [Join] контроль переходить до дії «Створення точки збіжності» (Create convergence point). Ця дія також взаємодіє зі сховищем Created action nodes.

Паралельно з описаними вище діями (що підтверджується лініями

синхронізації/розгалуження — fork bars), виконується специфічний підпроцес, пов'язаний з обробкою логіки вибору та доставки (ймовірно, прикладний сценарій, що використовується як приклад):

- Виконується дія «Перевірка підбірника» (Check picker).

- Далі слідує дія «Відправлення замовлення» (ship order).

- Останньою в цьому ланцюгу є дія «Надіслати інформацію про відстеження» (Send tracking information), яка оновлює дані у сховищі Created action nodes.

Усі паралельні потоки управління всередині області «Генерація робочого процесу» зливаються на вузлі злиття (merge node) перед виходом з ітеративної області.

3. Вкладений цикл - уточнення правил потоку

Всередині основної ітеративної області виділено ще одну, вкладену ітеративну область — «Уточнення правил потоку (ітеративне)» (Flow Rule Refinement (Iterative)). На її вхід надходять об'єкти типу Вузол (Node) (зі сховища Nodes) та Правило (Rule) (зі сховища Flow Rules).

Логіка цієї області спрямована на детальне налаштування логіки переходів між вузлами:

- Процес починається з вузла прийняття рішення, який аналізує вхідне Rule.

- Виконується дія «Створення порожнього правила потоку» (Create empty Flow Rule).

- Наступна дія — «Отримати тип цільового вузла» (Get type target node), яка приймає створене правило та вузол для визначення типу наступного кроку.

- На основі отриманого типу відбувається розгалуження (за логікою іконок на ребрах):

- Якщо цільовий вузол є звичайним вузлом дії, виконується дія «Додати цільовий вузол до правила потоку» (Add target node to Flow Rule).

- Якщо логіка переходу складніша (іконка "діамант"), виконується дія «Розширити правило потоку для специфічного вузла» (Expand Flow Rule for specific node).

- Обидва шляхи ведуть до створення оновлених об'єктів Node та Rule, які повертаються на вхід першого вузла прийняття рішення всередині цієї вкладеної області, утворюючи цикл уточнення. Результати обробки (множина Created Flow Rules [0..*]) акумулюються у відповідному сховищі даних.

4. Фіналізація та завершення

Після завершення виконання основної ітеративної області «Генерація робочого процесу», сформований потік об'єктів «Дані двигуна робочих процесів» (Workflow engine data [0..*]) надходить на вхід фінальної дії.

Дія «Додати дані двигуна робочих процесів» (Add Workflow engine data) приймає цей потік і здійснює запис фінальної конфігурації у сховище даних Workflow engine data.

Після успішного виконання цієї дії процес завершується у фінальному вузлі активності (final activity node).

3.5. Алгоритмічне забезпечення детермінованих переходів та оцінки логічних умов у системах керування робочими процесами

Процес просування активного вузла в межах виконуваного робочого процесу базується на евалюації правил потоку (Flow Rules), що асоційовані з конкретним вузлом дії. Структурно кожне правило детерміноване двома основними атрибутами: next та condition.

Атрибут next призначений для опису прямих переходів, що не мають зовнішніх логічних залежностей і не потребують аналізу контексту під час виконання (наприклад, лінійний перехід між діями або вихід до термінального вузла). Натомість атрибут condition застосовується для

моделювання складних розгалужень, де вибір наступного кроку залежить від актуального стану даних активного процесу. У межах бази даних умови зберігаються у форматі JSON. Для оптимізації обробки цих даних на рівні абстракції бази даних реалізовано спеціалізоване поле, яке автоматично трансформує сирі JSON-дані у структуровані об’єкти мови програмування під час їх зчитування.

Нижче наведено алгоритм, що описує процедуру ідентифікації наступних вузлів для активації.

Алгоритм 3.1. Детермінація наступного вузла в робочому процесі

Вхідні дані: active_process, список потенційних переходів next_nodes.

Вихідні дані: Список вузлів дії для активації або None.

1. Впорядкувати next_nodes за пріоритетом: переходи з визначеними умовами обробляються першими.

2. Для кожного вузла node у списку next_nodes виконати:

3. Якщо вузол має логічну умову ТА результат функції _evaluate_condition є False:

4. Перейти до наступного ітератора (пропустити вузол).

5. Кінець якщо

6. Якщо присутня перевірка точки збіжності ТА результат _check_convergence_point є False:

7. Повернути None (очікування завершення паралельних гілок).

8. Кінець якщо

9. Визначити значення цільового переходу next_value = node.next.

10. Якщо next_value є масивом ідентифікаторів:

11. Повернути список об'єктів вузлів дії для кожного ID.

12. Інакше якщо next_value є вкладеною структурою вузлів:

13. Повернути результат рекурсивного виклику _get_next_node для цієї структури.

14. Інакше якщо next_value є одиничним ідентифікатором:

15. Повернути список, що містить один відповідний вузол дії.

16. Інакше якщо next_value має значення "END":
17. Повернути None (термінація процесу).
18. Кінець
Повернути None.

Згідно з Алгоритмом 3.1, вибір вектору руху в робочому процесі вимагає одночасної оцінки предикатних умов та логіки синхронізації. Перехід вважається валідним лише за умови істинності пов'язаної з ним логічної умови. Для цього поточний стан даних процесу аналізується за допомогою процедури, описаної в Алгоритмі 3.2.

Алгоритм 3.2. Оцінка логічної умови на основі контексту процесу

Вхідні дані: active_process, об'єкт умови condition.

Вихідні дані: True, якщо умова задоволена; інакше False.

1. Отримати цільовий набір об'єктів з active_process, використовуючи специфікацію класу в condition.target_class_name.

2. Якщо в умові визначено агрегувальну функцію:

- Обчислити агреговане значення (sum,average,max,min або count) для атрибута condition.target_attribute.

3. Інакше:

- Отримати значення цільового атрибута з першого об'єкта в наборі.

4. Кінець якщо

5. Привести порогове значення умови до типу даних, що відповідає condition.target_attribute_type.

6. Виконати операцію порівняння отриманого значення з порогом за допомогою оператора condition.operator.

7. Повернути результат порівняння.

Окрім предикатної логіки, система забезпечує цілісність паралельних обчислень через механізми збіжності. Це гарантує, що наступні етапи процесу не будуть активовані до моменту завершення всіх необхідних попередніх гілок.

Алгоритм 3.3. Верифікація завершення точки збіжності (Join Node)

Вхідні дані: active_process, ідентифікатор точки збіжності convergence_point_id.

Вихідні дані: True, якщо умови збіжності виконані; інакше False.

1. Отримати об'єкт активної точки збіжності для поточного процесу за його ідентифікатором.

2. Якщо об'єкт відсутній:

- Створити новий запис активної точки збіжності для даного процесу.

- Повернути False (ініціалізація очікування).

3. Інакше:

- Інкрементувати лічильник (count) точки збіжності, фіксуючи досягнення її черговим вхідним потоком.

- Зберегти оновлений стан об'єкта в базі даних.

- Якщо поточне значення лічильника менше за необхідну кількість вхідних потоків:

- Повернути False.

- Інакше:

- Видалити об'єкт активної точки збіжності (звільнення ресурсів після виконання умови).

- Повернути True.

- Кінець якщо

4. Кінець якщо

3.6. Аналіз стійкості алгоритмів управління робочими процесами до виняткових ситуацій

Забезпечення надійності механізму робочих процесів вимагає не лише виконання детермінованих сценаріїв, а й коректної обробки аномальних станів даних або логічних розривів у моделі. Нижче наведено аналітичний огляд механізмів обробки виняткових ситуацій у межах раніше описаних алгоритмів.

1. Обробка відсутності цільових об'єктів (Алгоритм 3.2)

Однією з найбільш розповсюджених виняткових ситуацій є відсутність екземплярів даних, на основі яких має бути оцінена логічна умова переходу. Це часто трапляється на початкових етапах процесу або при некоректному завершенні попередніх кроків.

Стратегія порожнього набору (Null-Set Policy): Якщо функція екстракції об'єктів за target_class_name повертає порожній набір, Алгоритм 3.2 ініціює захисний механізм. Для агрегувальних функцій, таких як count (підрахунок), повертається значення 0. Для функцій sum, average, max або min система повертає нейтральне значення (зазвичай 0 або None залежно від конфігурації), що запобігає виникненню помилок переповнення або ділення на нуль.

Безпека атрибутивного доступу: У разі прямого порівняння без агрегації, спроба доступу до атрибута першого об'єкта в порожньому наборі блокується перевіркою на існування. У такому випадку умова автоматично набуває значення False, що зупиняє просування за даним вектором переходу та вимагає втручання оператора або ініціалізації альтернативного шляху else.

2. Валідація типів та безпека порівняння

Невідповідність типів даних між пороговим значенням у метаданих та фактичним атрибутом у базі даних може призвести до системних збоїв на етапі виконання.

Динамічне приведення типів: Перед виконанням операції порівняння механізм здійснює примусову конвертацію порогового значення до типу target_attribute_type. Якщо конвертація є неможливою (наприклад, спроба порівняти текстовий рядок із числовим порогом), алгоритм генерує внутрішній виняток, який реєструється в Action Log, а поточний крок процесу маркується як такий, що потребує технічного аудиту.

3. Запобігання логічним тупикам у точках збіжності (Алгоритм 3.3)

Синхронізація паралельних шляхів через Join-вузли є критичною

точкою, де можливе виникнення «стану очікування» (deadlock), якщо одна з паралельних гілок не була завершена.

Моніторинг життєвого циклу точок збіжності: Алгоритм 3.3 забезпечує стійкість шляхом персистентного зберігання стану очікування. Якщо через помилку в логіці моделі один із потоків управління ніколи не досягне точки збіжності, механізм залишає Active Convergence Point у стані «очікування».

Механізм примусової деблокації: Для запобігання вічному очікуванню адміністративні інтерфейси дозволяють менеджеру процесу вручну завершувати або ігнорувати заблоковані точки збіжності, що забезпечує просування процесу в екстрених ситуаціях.

4. Цілісність маршрутизації (Алгоритм 3.1)

Якщо в результаті оцінки всіх можливих переходів (next_nodes) жодна умова не повертає True і не визначено шлях за замовчуванням (else), процес опиняється в стані невизначеності.

Статус «Очікування рішення»: В Алгоритмі 1 передбачено, що повернення None без маркування процесу як «завершений» переводить його в статус очікування. Це дозволяє уникнути некоректного закриття процесу та дає можливість розробнику або аналітику скоригувати дані процесу для виконання однієї з існуючих умов.

Отже, описана архітектура алгоритмів побудована на принципі «відмови в безпеці» (fail-safe). Будь-яка непередбачувана ситуація (відсутність даних, помилка типу або логічний розрив) призводить до зупинки автоматичного просування процесу та фіксації інциденту в журналі аудиту, замість виконання некоректних дій або аварійного завершення всієї системи.

3.7. Адаптивність та архітектурна розширюваність механізмів керування робочими процесами

У даному розділі розглядаються принципи забезпечення гнучкості систем автоматизованої генерації програмного забезпечення через механізми динамічного налаштування правил та стратегій розподілу ресурсів.

3.7.1. Конфігурація та екстенсивність предикатної логіки

Архітектура механізму робочих процесів базується на принципах керування даними (data-driven design). Це дозволяє здійснювати модифікацію логіки переходів між активностями безпосередньо під час виконання системи (runtime) шляхом внесення змін до параметрів правил у базі даних. Типовим прикладом такої адаптації є коригування числових порогів у JSON-структурах умов, що детермінують необхідність додаткової верифікації бізнес-об'єктів.

Окрім параметричного налаштування, система забезпечує високий рівень розширюваності мови правил на рівні програмного коду. Додавання нових логічних операторів або функцій агрегації здійснюється шляхом оновлення відповідних словників у класі дефініції правил.

Лістинг 3.1 Фрагмент технічної реалізації класу FlowRule

```
# Реалізація у файлі моделі механізму робочих процесів
class FlowRule(models.Model):
    # Словник конвертерів для приведення рядкових даних до складних типів
    type_converters = {
        "tuple_int": lambda x: tuple(map(int, x.split("-"))),
    }

    # Реєстр операторів порівняння
    operators = {
        "between": lambda value, threshold: threshold[0] <= value <= threshold[1
    }

    # Реєстр функцій агрегації для QuerySet
    aggregators = {
        "exists": lambda qs, _: qs.exists(),
    }
    # Подальша реалізація логіки евалюації...
```

Вище наведено фрагмент технічної реалізації класу FlowRule, що демонструє методику інтеграції нового оператора діапазону (between) та агрегатора існування (exists).Python

Процес обробки умови передбачає попереднє застосування конвертера типів перед викликом цільового оператора. Це гарантує консистентність даних, оскільки порогові значення зберігаються в персистентному сховищі у рядковому форматі. Наприклад, конвертер перетворює рядок "1-100" у кортеж (1, 100), що є необхідною умовою для коректного функціонування оператора between. Така архітектура дозволяє аналітикам впроваджувати складні математичні та логічні моделі шляхом простої зміни атрибута target_attribute_type у базі даних.

3.7.2. Динамічна детермінація виконавців та стратегії призначення

Гнучкість механізму робочих процесів також проявляється у можливості адаптації алгоритмів призначення завдань конкретним суб'єктам системи. Поточна реалізація базується на івристичній стратегії забезпечення неперервності контексту: система намагається призначити наступну активність тому ж користувачу, який завершив попередній етап, за умови відповідності його прав заданій ролі актора.

Лістинг 3.2. Налаштування стратегії призначення користувачів

```
def _get_next_user(self, current_user: User | None) -> User | None:
    """Алгоритм ідентифікації виконавця для вузла дії."""
    # Відсутність призначення для автоматизованих/системних вузлів
    if not self.url:
        return None

    # Пріоритет збереження поточного виконавця для мінімізації втрати контексту
    if current_user and self.actor in current_user.roles:
        return current_user

    # Базова стратегія вибору за роллю.
    # Може бути розширена алгоритмами Round Robin або аналізом поточної черги завдань.
    users = User.objects.filter(**{f"is_{self.actor}": True})
    return users.first() if users.exists() else None
```

У разі неможливості дотримання принципу неперервності, алгоритм здійснює вибір першого доступного користувача з відповідною рольовою кваліфікацією. Проте, архітектура методу _get_next_user() у класі ActionNode спроектована з урахуванням подальшої інтеграції складніших стратегій розподілу навантаження.

Запропонована структура дозволяє легко інтегрувати методи балансування навантаження (load balancing) або аналіз доступності персоналу в режимі реального часу. Це забезпечує масштабованість системи та її здатність адаптуватися до мінливих організаційних умов без докорінної зміни архітектури ядра робочих процесів.

3.8. Тестування функціональної придатності системи генерації прототипів програмного забезпечення

Як було обґрунтовано, методологія кейс-стаді була застосована для комплексної оцінки продуктивності та верифікації архітектурних рішень моделі на ітераційних етапах її розробки.

3.8.1. Верифікація базової логіки та архітектурна стабілізація

Перший етап емпіричної оцінки був зосереджений на верифікації базових функціональних можливостей діаграм активностей UML. Попри фокусування на фундаментальних операціях механізму робочих процесів, дана стадія вимагала найбільшої концентрації ресурсів. Це зумовлено необхідністю розробки та впровадження системних функцій, що забезпечують життєздатність механізму, включаючи глибокий рефакторинг об'єктів представлення (Views) програмного фреймворку та проектування динамічних сторінок активностей.

У ході проведення дослідження було виділено декілька критичних аспектів оптимізації системи:

1. Семантична прозорість та людино-машинна взаємодія (HCI).

Аналіз виявив, що на початкових етапах розробки термінологія інтерфейсу була надмірно технічно орієнтованою, що створювало когнітивні бар'єри для кінцевих користувачів. З метою підвищення інклюзивності системи було проведено перегляд найменувань елементів на головній сторінці та ідентифікаторів кнопок завершення завдань. Крім того, результати зворотного зв'язку вказали на недостатню структурованість сторінок активностей. Для усунення цього недоліку було інтегровано контекстні інструкції, які інформують користувача про послідовність необхідних кроків у межах вибраної операції.

2. Формалізація функціональних вимог та безпека.

Впровадження механізмів робочих процесів виявило неможливість збереження факультативної аутентифікації. Було встановлено, що інтеграція Workflow-логіки вимагає переходу до обов’язкової ідентифікації та автентифікації суб'єктів у межах рольової моделі доступу (RBAC) для забезпечення підзвітності. Також було ідентифіковано дефіцит функцій життєвого циклу процесів, зокрема відсутність механізму видалення існуючих екземплярів. Відповідний інструментарій було додано до інтерфейсу домашньої сторінки.

3. Системна стійкість та обробка виняткових ситуацій.

У процесі тестування було виявлено критичну вразливість: виникнення системної помилки (runtime error) при спробі призначення завдання ролі, для якої у системі відсутні активні користувачі. Для забезпечення безперервності процесу було впроваджено алгоритм деградації функціоналу: у разі відсутності доступного виконавця завдання переходить у статус «непризначеного», а менеджер системи отримує автоматичне сповіщення для ручного втручання та перерозподілу ресурсів.

Візуальні матеріали та знімки екранів інтерфейсів, отримані під час проведення даного кейс-стаді, представлені на рисунках 3.7 – 3.9.

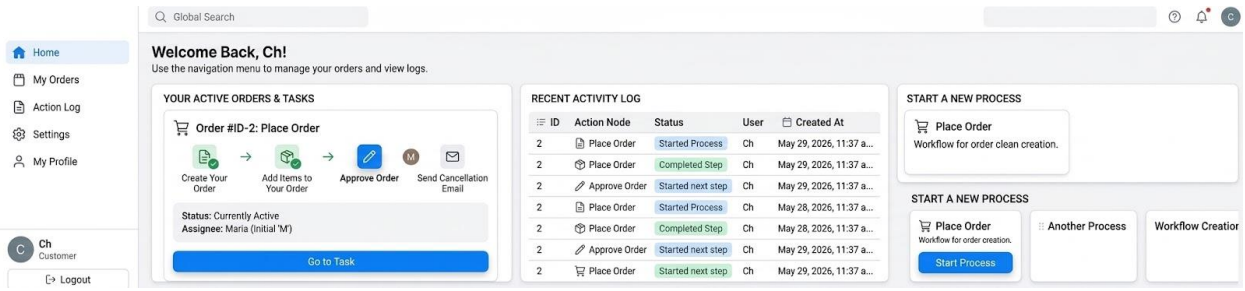

Рисунок 3.7 – Сценарій 1. Головна сторінка

Рисунок 3.8 – Сценарій 1. Кнопка завершення етапу активності

Рисунок 3.9 – Сценарій 1. Сторінка активності

3.8.2. Апробація розширеного функціоналу та оптимізація середовища візуального моделювання

На другому етапі основну увагу було приділено інтеграції та верифікації розширених функціональних можливостей механізму робочих процесів. Зі збільшенням складності модельованих бізнес-процесів виникли нові виклики, пов'язані з масштабованістю графічного представлення та ергономікою розробки.

1. Оптимізація візуальної топології діаграм

Зростання кількості вузлів та зв'язків у межах однієї моделі призвело до критичного ускладнення візуальної структури діаграм активностей. Було

ідентифіковано обмеження, пов'язане з використанням виключно прямолінійних ребер (control flows), що спричиняло численні перетини графічних елементів та погіршувало когнітивне сприйняття логіки процесу.

Для усунення цієї проблеми в архітектуру редактора було впроваджено компоненти PositionHandlers. Ці інструменти дозволяють визначати проміжні координати для ребер, забезпечуючи можливість створення ламаних ліній та обходу перешкод, що суттєво підвищує читабельність складних графів.

2. Еволюція семантичної класифікації завдань

На даному етапі було впроваджено механізми автоматизованих завдань, що ініціюються системою без участі оператора. Початкова диференціація завдань на «ручні» та «автоматичні» виявилася недостатньо чіткою для кінцевих користувачів.

В результаті проведеного аналізу термін «ручні завдання» було замінено на більш точну дефініцію — «завдання інтерфейсу користувача» (User Interface Tasks). Така зміна термінології дозволила чітко розмежувати операції, що вимагають взаємодії з UI-компонентами, та фонові системні процеси, спрощуючи процес концептуального моделювання для аналітиків.

3. Контекстуалізація середовища розробки код-блоків

Важливим аспектом дослідження став аналіз ефективності вбудованого редактора коду для автоматизованих сценаріїв. Первинна апробація показала, що користувачі стикалися з труднощами при реалізації логіки через дефіцит контекстної інформації безпосередньо в інтерфейсі програмування.

З метою вирішення цієї проблеми функціонал редактора було розширено:

- Впроваджено систему автоматичного додавання коментарів, що описують доступні змінні процесу.

- Інтегровано шаблони (boilerplate) коду, які надають необхідний структурний контекст для виконання конкретної операції.

Це дозволило суттєво знизити поріг входу для розробників та

зменшити кількість синтаксичних і логічних помилок при створенні автоматизованих дій. Детальні візуальні докази впроваджених змін та знімки екрана робочого інтерфейсу подано на рисунках 3.10 - 3.13.

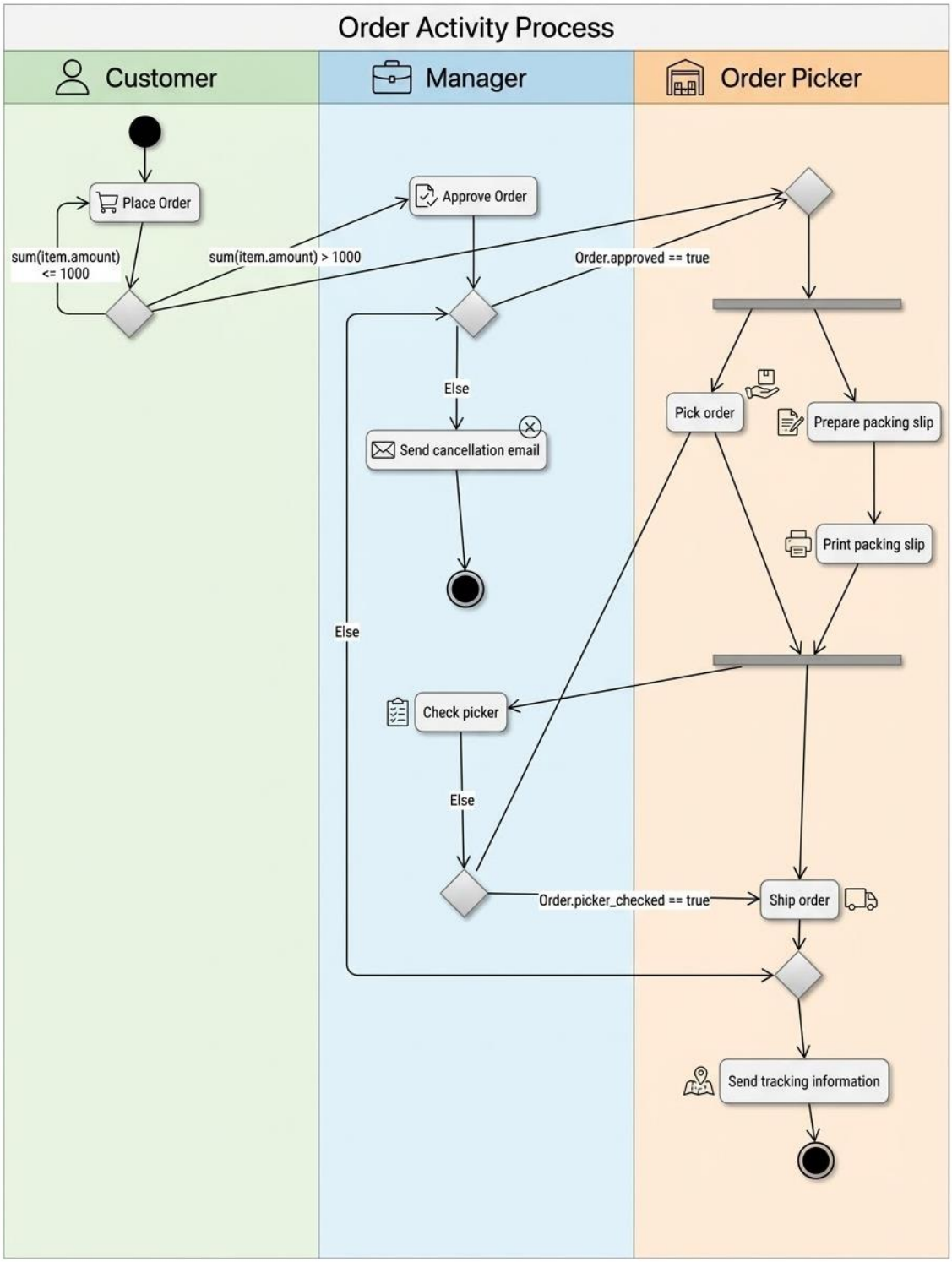

Рисунок 3.10 – Сценарій 2. Діаграма потоків керування

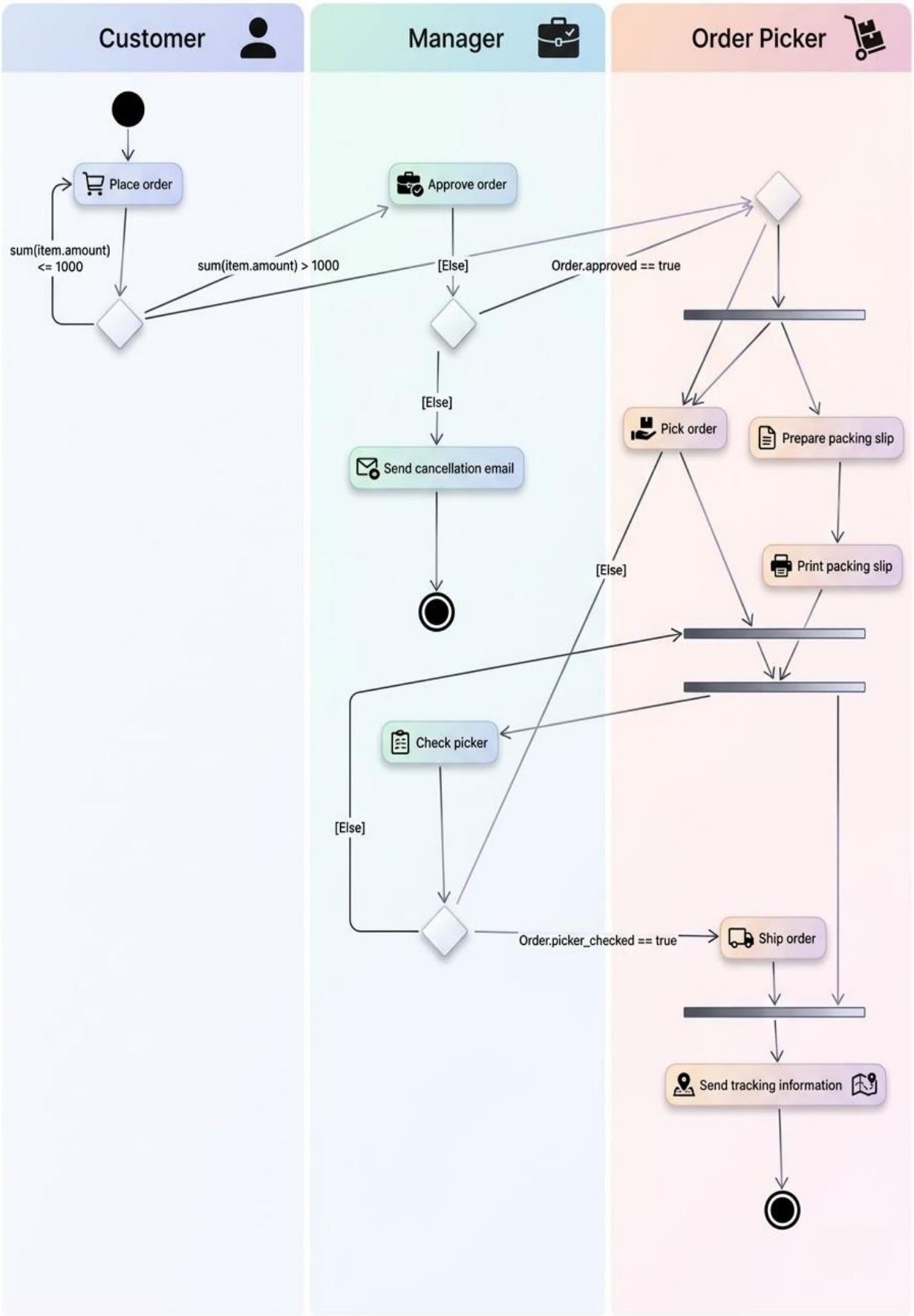

Рисунок 3.11 – Сценарій 2. Послідовність процесу

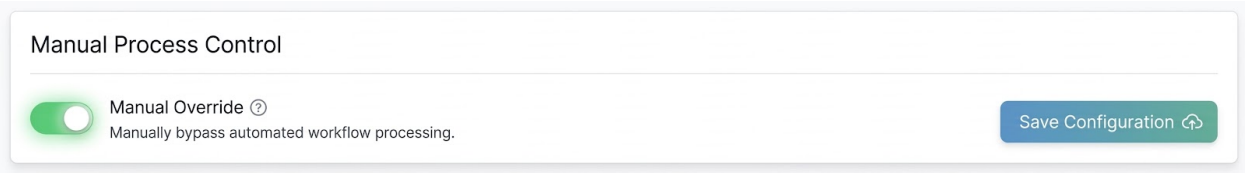

Рисунок 3.12 – Сценарій 2. Мітка типу сторінки

Рисунок 3.13 – Сценарій 2. Редактор коду

Попри функціональну повноту ядра системи, було ідентифіковано ряд технічних аспектів, що потребують подальшої оптимізації:

1. Ергономіка середовища моделювання.

Інтерфейсний шар, призначений для побудови діаграм UML, потребує впровадження додаткових інструментів візуалізації та інтелектуальних підказок. Це є критичним для мінімізації помилок при проектуванні, особливо в частині написання спеціалізованого коду (custom code) користувачами з обмеженим досвідом програмування.

2. Диспропорція функціональних рівнів.

Емпіричне тестування виявило певний розрив між можливостями серверної частини (backend) та згенерованим клієнтським інтерфейсом (frontend). Поточна потужність логіки обробки процесів на серверному рівні дещо випереджає можливості її візуальної репрезентації та взаємодії на стороні користувача.

3. 9 Висновки по розділу

У третьому розділі здійснено практичну реалізацію методів генерації прототипів програмних систем із використанням розроблених моделей та алгоритмів. Розроблено розширення метаданих і функціональні специфікації, що забезпечують інтеграцію UML-моделей із механізмами виконання робочих процесів. Реалізовано програмну архітектуру системи, включаючи механізми обробки даних, управління переходами та взаємодії компонентів у режимі реального часу. Проведений аналіз стійкості та тестування підтвердили надійність і коректність функціонування системи за різних умов експлуатації. Отримані результати засвідчили практичну придатність запропонованого підходу та його здатність до адаптації й розширення в межах сучасних інформаційних систем.

ВИСНОВКИ

У результаті виконання дипломної роботи на тему «Методи генерації прототипів програмних систем засобами модельно-орієнтованої інженерії» було здійснено теоретико-методологічне та прикладне дослідження підходів до автоматизації процесів проєктування та розробки програмного забезпечення на основі моделей.

У першому розділі проведено ґрунтовний аналіз сучасного стану та проблематики застосування модельно-орієнтованої інженерії у розробці програмного забезпечення. Встановлено, що ключовою перевагою цього підходу є підвищення рівня абстракції, що дозволяє відокремити бізнес-логіку від технічної реалізації та забезпечити повторне використання моделей. Дослідження методологічних засад архітектури, керованої моделями, підтвердило доцільність використання багаторівневого представлення систем із чітким розмежуванням платформно-незалежних та платформно-специфічних моделей. Особливу увагу приділено ролі уніфікованої мови моделювання UML як універсального інструменту формалізації вимог і структури системи, що виступає основою для автоматизованої генерації програмного коду.

У другому розділі розроблено методологічне та алгоритмічне обґрунтування системи генерації прототипів програмного забезпечення. Запропонована методологія дизайн-дослідження забезпечила систематичний підхід до побудови архітектури генератора, орієнтований на ітеративну перевірку рішень у реалістичних сценаріях використання. Верифікація архітектури на основі тематичних сценаріїв дозволила підтвердити її коректність у контексті як базових, так і складних процесів, включаючи умовну маршрутизацію та паралельне виконання.

Сформовано архітектуру механізму керування робочими процесами, яка базується на принципах модульності, розширюваності та динамічного

зв’язування компонентів. Розроблено методологію генерації програмних прототипів, що включає процедури відображення метаданих, формування структур даних і забезпечення виконання процесів у режимі реального часу. Запропоновано архітектурну структуру двигуна робочих процесів, яка забезпечує обробку станів, керування переходами та підтримку складних логічних умов.

У третьому розділі здійснено практичну реалізацію запропонованих методів генерації прототипів програмних систем. Розроблено специфікацію розширень метаданих, що дозволяє розширити можливості UML-моделей для опису поведінки та логіки системи. Сформовано функціональну специфікацію інтерфейсів користувача, орієнтовану на інтеграцію з механізмами керування робочими процесами.

Реалізовано архітектуру програмної системи, включаючи класи, модулі та алгоритмічні протоколи взаємодії компонентів. Особливу увагу приділено реалізації механізмів переходів між етапами робочого процесу, заповнення даних та оптимізації генерації об’єктів представлення. Забезпечено впровадження механізмів аутентифікації та контролю доступу, що підвищує рівень безпеки системи. Розроблено алгоритмічне забезпечення для детермінованих переходів та оцінки логічних умов, що гарантує коректність виконання процесів. Проведено аналіз стійкості алгоритмів до виняткових ситуацій, що дозволило підвищити надійність системи в умовах невизначеності. Проведене тестування підтвердило функціональну придатність розробленої системи генерації прототипів програмного забезпечення, її здатність до коректного виконання як базових, так і розширених сценаріїв, а також ефективність інтеграції з середовищем візуального моделювання.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Schmidt, D. C. Model-Driven Engineering. IEEE Computer, Los Alamitos: IEEE Computer Society, 2006, Vol. 39(2), pp. 25–31.
2. France, R., Rumpe, B. Model-Driven Development of Complex Software: A Research Roadmap. Proceedings of FOSE 2007, Minneapolis: IEEE, 2007, pp. 37–54.
3. Mellor, S. J., Scott, K., Uhl, A., Weise, D. Model-Driven Architecture. IEEE Software, Los Alamitos: IEEE, 2003, Vol. 20(5), pp. 14–18.
4. Object Management Group. Unified Modeling Language (UML) Specification, Version 2.5.1. Needham: OMG, 2017, 796 p.
5. Stahl, T., Völter, M. Model-Driven Software Development: Technology, Engineering, Management. Chichester: Wiley, 2006, pp. 1–45.
6. Brambilla, M., Cabot, J., Wimmer, M. Model-Driven Software Engineering in Practice. San Rafael: Morgan & Claypool, 2017, pp. 50–95.
7. Selic, B. The Pragmatics of Model-Driven Development. IEEE Software, Los Alamitos: IEEE, 2003, Vol. 20(5), pp. 19–25.
8. Kent, S. Model Driven Engineering. Integrated Formal Methods Conference, Canterbury: Springer, 2002, pp. 286–298.
9. Bézivin, J. Model Driven Engineering: An Emerging Technical Space. Generative and Transformational Techniques in Software Engineering, Braga: Springer, 2006, pp. 36–64.
10. Hailpern, B., Tarr, P. Model-Driven Development: The Good, the Bad, and the Ugly. IBM Systems Journal, Armonk: IBM, 2006, Vol. 45(3), pp. 451–461.
11. Fowler, M. Domain-Specific Languages. Boston: Addison-Wesley, 2010, pp. 10–60.
12. Greenfield, J., Short, K., Cook, S., Kent, S. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.

Indianapolis: Wiley, 2004, pp. 75–120.

13. Atkinson, C., Kuhne, T. Model-Driven Development: A Metamodeling Foundation. IEEE Software, Los Alamitos: IEEE, 2003, Vol. 20(5), pp. 36–41.

14. Czarnecki, K., Eisenecker, U. Generative Programming: Methods, Tools, and Applications. Boston: Addison-Wesley, 2000, pp. 120–180.

15. Sendall, S., Kozaczynski, W. Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software, Los Alamitos: IEEE, 2003, Vol. 20(5), pp. 42–45.

16. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Völkel, S. Design Guidelines for Domain Specific Languages. OOPSLA Workshop, Montreal: ACM, 2009, pp. 1–6.

17. Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1995, pp. 15–40.

18. Richardson, C. Microservices Patterns. Shelter Island: Manning, 2018, pp. 35–80.

19. Pautasso, C., Zimmermann, O., Leymann, F. RESTful Web Services vs. Big Web Services. WWW Conference, Beijing: ACM, 2008, pp. 805–814.

20. Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures. Irvine: University of California, 2000, pp. 76–106.

21. Van der Aalst, W. M. P. Workflow Management: Models, Methods, and Systems. Cambridge: MIT Press, 2004, pp. 1–50.

22. Russell, N., van der Aalst, W. M. P., ter Hofstede, A. Workflow Control-Flow Patterns. Data & Knowledge Engineering, Amsterdam: Elsevier, 2006, Vol. 60(1), pp. 1–22.

23. Dumas, M., La Rosa, M., Mendling, J., Reijers, H. Fundamentals of Business Process Management. Berlin: Springer, 2018, pp. 100–150.

24. Weske, M. Business Process Management: Concepts, Languages,

Architectures. Berlin: Springer, 2012, pp. 55–110.

25. Django Software Foundation. Django Documentation. Lawrence: DSF, 2024, pp. 1–200.

26. Holovaty, A., Kaplan-Moss, J. The Definitive Guide to Django. Berkeley: Apress, 2009, pp. 30–85.

27. Brownlee, J. Software Architecture for Developers. Raleigh: Leanpub, 2015, pp. 20–60.

28. Bass, L., Clements, P., Kazman, R. Software Architecture in Practice. Boston: Addison-Wesley, 2013, pp. 75–140.

29. Sommerville, I. Software Engineering. Boston: Pearson, 2016, pp. 100–180.

30. Pressman, R. S., Maxim, B. R. Software Engineering: A Practitioner's Approach. New York: McGraw-Hill, 2020, pp. 120–200.

31. Ambler, S. W. Agile Modeling. New York: Wiley, 2002, pp. 40–90.

32. Beck, K. Extreme Programming Explained. Boston: Addison-Wesley, 2000, pp. 10–50.

33. Fowler, M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2003, pp. 55–120.

34. Evans, E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston: Addison-Wesley, 2004, pp. 80–150.

35. Newcomer, E., Lomow, G. Understanding SOA with Web Services. Boston: Addison-Wesley, 2005, pp. 60–110.

36. Papazoglou, M. P. Service-Oriented Computing: Concepts, Characteristics and Directions. Proceedings of WISE 2003, Rome: Springer, 2003, pp. 3–12.

ДОДАТКИ

Додаток А

Фрагмент коду інтерфейсу

```
import React from 'react';
import {
  Home, ClipboardList, History, Settings, User,
  Search, Bell, HelpCircle, LogOut, ShoppingCart,
  CheckCircle2, Box, PenLine, Mail
} from 'lucide-react';

const Dashboard = () => {
  return (
    <div  className="flex  h-screen  bg-gray-50  text-slate-800  font-
sans">
      {/* Sidebar */}
      <aside  className="w-64  bg-white  border-r  border-gray-200  flex
flex-col">
        <div className="p-6 flex flex-col flex-grow">
          <nav className="space-y-1">
            <NavItem icon={<Home size={20} />} label="Home" active />
            <NavItem  icon={<ClipboardList  size={20}  />}  label="My
Orders" />
            <NavItem  icon={<History  size={20}  />}  label="Action  Log"
/>
            <NavItem icon={<Settings size={20} />} label="Settings" />
            <NavItem icon={<User size={20} />} label="My Profile" />
          </nav>
        </div>

        <div className="p-4 border-t border-gray-100">
          <div className="flex items-center gap-3 px-2 py-3">
            <div  className="w-10  h-10  rounded-full  bg-slate-200  flex
items-center justify-center font-bold text-slate-600">
              C
            </div>
            <div>
              <p className="font-semibold text-sm">Charlie</p>
```

```
              <p className="text-xs text-gray-500">Customer</p>
            </div>
          </div>
          <button className="flex items-center gap-2 w-full px-2 py-2 mt-2 text-sm text-gray-600 hover:bg-red-50 hover:text-red-600 rounded-lg transition-colors">
            <LogOut size={18} /> Logout
          </button>
        </div>
      </aside>

      {/* Main Content */}
      <main className="flex-1 flex flex-col overflow-hidden">
        {/* Topbar */}
        <header className="h-16 bg-white border-b border-gray-200 flex items-center justify-between px-8">
          <div className="relative w-96">
            <Search  className="absolute  left-3  top-1/2  -translate-y-1/2 text-gray-400" size={18} />
            <input
              type="text"
              placeholder="Global Search"
              className="w-full bg-gray-100 rounded-md py-2 pl-10 pr-4 text-sm focus:outline-none focus:ring-2 focus:ring-blue-500/20"
            />
          </div>
          <div className="flex items-center gap-4 text-gray-500">
            <HelpCircle       size={20}       className="cursor-pointer hover:text-blue-600" />
            <div className="relative">
              <Bell  size={20}  className="cursor-pointer  hover:text-blue-600" />
              <span  className="absolute  -top-1  -right-1  w-2  h-2  bg-red-500 rounded-full border-2 border-white"></span>
            </div>
            <div  className="w-8  h-8  rounded-full  bg-slate-400  flex items-center justify-center text-white text-xs font-bold">C</div>
          </div>
```

```
        </header>

        {/* Dashboard Body */}
        <div className="flex-1 overflow-y-auto p-8">
          <div className="mb-8">
            <h1 className="text-2xl font-bold">Welcome Back, Charlie!</h1>
            <p className="text-gray-500 text-sm">Use the navigation menu to manage your orders and view logs.</p>
          </div>

          <div className="grid grid-cols-12 gap-6">
            {/* Active Orders Card */}
            <div className="col-span-12 lg:col-span-4 bg-white rounded-xl shadow-sm border border-gray-100 flex flex-col">
              <div className="p-5 border-b border-gray-50">
                <h2 className="font-bold text-xs uppercase tracking-wider text-gray-500">Your Active Orders & Tasks</h2>
              </div>
              <div className="p-6 flex-1">
                <div className="flex items-center gap-3 mb-6">
                  <ShoppingCart className="text-slate-700" size={22} />
                  <h3 className="font-bold">Order #ID-2: Place Order</h3>
                </div>

                {/* Stepper */}
                <div className="flex items-center justify-between mb-8 px-2">
                  <Step icon={<CheckCircle2 className="text-green-500" />} label="Create Your Order" active />
                  <div className="h-px bg-green-200 flex-1 mx-2" />
                  <Step icon={<Box className="text-green-500" />} label="Add Items to Your Order" active />
                  <div className="h-px bg-gray-200 flex-1 mx-2" />
                  <Step icon={<PenLine className="text-blue-600" />} label="Approve Order" current />
```

```
                <div className="h-px bg-gray-200 flex-1 mx-2" />
                <Step icon={<Mail className="text-gray-300" />}
label="Send Cancellation Email" />
              </div>

            {/* Recent Activity Log */}
            <div  className="col-span-12  lg:col-span-5  bg-white
rounded-xl shadow-sm border border-gray-100 overflow-hidden">
              <div className="p-5 border-b border-gray-50">
                <h2 className="font-bold text-xs uppercase tracking-
wider text-gray-500">Recent Activity Log</h2>
              </div>
              <div className="overflow-x-auto">
                <table className="w-full text-left text-sm">
                  <thead  className="bg-gray-50  text-gray-500  text-xs
uppercase">
                    <tr>
                      <th className="px-6 py-3 font-semibold">ID</th>
                      <th  className="px-6  py-3  font-semibold">Action
Node</th>
                      <th       className="px-6       py-3       font-
semibold">Status</th>
                      <th       className="px-6       py-3       font-
semibold">User</th>
                      <th  className="px-6  py-3  font-semibold">Created
At</th>
                    </tr>
                  </thead>
                  <tbody className="divide-y divide-gray-100">
                    <LogEntry     id="2"     action="Place     Order"
status="Started  Process"  color="blue"  user="Charlie"  date="May  29,
2025" />
                    <LogEntry     id="2"     action="Place     Order"
status="Completed  Step"  color="green"  user="Charlie"  date="May  28,
2025" />
                    <LogEntry     id="2"     action="Approve     Order"
status="Started  next  step"  color="blue"  user="Maria"  date="May  29,
2025" />
```

```
                </tbody>
              </table>
            </div>
          </div>

          {/* Start New Process */}
          <div className="col-span-12 lg:col-span-3 space-y-6">
            <div  className="bg-white  rounded-xl  shadow-sm  border
border-gray-100 p-6">
              <h2  className="font-bold  text-xs  uppercase  tracking-
wider text-gray-500 mb-4">Start a New Process</h2>
              <div  className="p-4  border  border-gray-100  rounded-lg
hover:border-blue-200 transition-colors cursor-pointer group">
                <div className="flex items-center gap-3 mb-2">
                  <ShoppingCart  size={18}  className="text-slate-600"
/>
                  <p className="font-bold text-sm">Place Order</p>
                </div>
                <p  className="text-xs  text-gray-400">Workflow  for
order clean creation.</p>
              </div>
            </div>

            <div  className="bg-white  rounded-xl  shadow-sm  border
border-gray-100 p-6">
              <h2  className="font-bold  text-xs  uppercase  tracking-
wider text-gray-500 mb-4 text-center">--- OR ---</h2>
              <button  className="w-full  bg-blue-600  hover:bg-blue-
700 text-white font-semibold py-2 rounded-lg text-sm">
                Start Process
              </button>
            </div>
          </div>
        </div>
      </main>
    </div>
  );
```

```
};

const LogEntry = ({ id, action, status, color, user, date }) => {
  const colors = {
    blue: "bg-blue-50 text-blue-600 border-blue-100",
    green: "bg-green-50 text-green-600 border-green-100"
  };
  return (
    <tr className="hover:bg-gray-50/50 transition-colors">
      <td className="px-6 py-4 text-gray-400">{id}</td>
      <td className="px-6 py-4 font-medium flex items-center gap-2">
        <ClipboardList size={14} className="text-gray-400" /> {action}
      </td>
      <td className="px-6 py-4">
        <span  className={`px-2  py-1  rounded  text-[10px]  font-bold
border uppercase tracking-tighter ${colors[color]}`}>
          {status}
        </span>
      </td>
      <td className="px-6 py-4 flex items-center gap-2">
        <div className="w-6 h-6 rounded-full bg-slate-200 flex items-
center justify-center text-[10px] font-bold">
          {user[0]}
        </div>
        {user}
      </td>
      <td className="px-6 py-4 text-gray-400 text-xs">{date}</td>
    </tr>
  );
};

export default Dashboard;
```

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: “ Методи генерації прототипів програмних систем засобами модельно-орієнтованої інженерії ”

Обсяг пояснювальної записки: 81 аркуші.

Дата закінчення роботи: 9 червня 2026 р.

Підпис студента _____