***БАКАЛАВРСЬКА РОБОТА***

БР.ІП – 38.00.00.000 ПЗ

**Група ІП-22-2**

**Крошний Іван**

**2026**

**Івано-Франківський національний технічний університет нафти і газу**

**Факультет інформаційних технологій**

**Кафедра інженерії програмного забезпечення**

**Крошний Іван Васильович**

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

# БАКАЛАВРСЬКА РОБОТА

**Інженерія програмного забезпечення для ВЕБ-сервісів**
(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 ступеня _____Крошний І.В._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник ____Яцишин Микола Миколайович, доцент, к.т.н._____
(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**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

# ЗАВДАННЯ
## НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

**Крошному Івану Васильовичу**

(прізвище, ім’я, по-батькові)

**1. Тема проекту (роботи) "Інженерія програмного забезпечення для ВЕБ-сервісів "**

керівник проекту (роботи) доц. , к.т.н., Яцишин М.М.

затверджені наказом вищого навчального закладу від **“** 21 ” квітня 2026 р. № 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 вимог до програмного забезпечення
2. Аналіз вимог і постановка задачі
3. Проектування системи
4. Реалізація проекту
5. Тестування та впровадження

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

1 Таксономія композиції веб-сервісів (рис. 1.2, ст. 18)

2 Загальна схема підходу (рис. 2.1, ст. 21)

3 Робочий процес, що відображає взаємодію запущеного прикладу (рис. 2.2, ст. 23)

4 Фази модельно-орієнтованої архітектури (рис. 3.2, ст. 44)

5 Базова метамодель CIM для MDE в S-CASE (рис. 3.4, ст. 49)

**6. Консультанти розділів проекту (роботи)**

| Розділ | Консультант | Підпис, дата | |
|---|---|---|---|
| | | Завдання видав | Завдання прийняв |
| | | | |
| | | | |

**2. Дата видачі завдання** **2026 р.**

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

# КАЛЕНДАРНИЙ ПЛАН

| № з/п | Назва етапів дипломного проекту (роботи) | Строк виконання етапів проекту | Примітка |
|---|---|---|---|
| 1 | Визначення та обгрунтування теми роботи | 17.03.2026 | виконано |
| 2 | Огляд існуючих концепцій, рішень та сервісів в даній області | 21.04.2026 | виконано |
| 3 | Побудова моделі або алгоритму власного рішення | 01.05.2026 | виконано |
| 4 | Документування реалізації власного оригінального рішення вибраними засобами | 21.05.2026 | виконано |
| 5 | Оформлення пояснювальної записки бакалаврської роботи | 09.06.2026 | виконано |

Студент _____Крошний І.В
(підпис)

Керівник роботи _____
(підпис) Яцишин М.М.

# АНОТАЦІЯ

Бакалаврська робота містить 77 сторінок, 28 рисунків, 12 таблиць, список використаних джерел із 40 найменувань.

**Метою роботи** є дослідження принципів інженерії програмного забезпечення для веб-сервісів та розроблення програмного рішення.

**Об'єкт дослідження:** процеси розробки та функціонування веб-сервісів.

**Предмет дослідження**: методи, технології та інструменти інженерії програмного забезпечення для проектування та реалізації веб-сервісів.

**Результати дослідження**: розроблено веб-сервіс, який демонструє використання сучасних підходів до проєктування програмного забезпечення, композиції сервісів, моделювання RESTful-архітектури та реалізації механізмів взаємодії між програмними компонентами.

**У першому розділі** проведено аналіз теоретичних основ інженерії програмного забезпечення для веб-сервісів, розглянуто підходи до композиції програмних компонентів та досліджено таксономію веб-сервісів.

**У другому розділі** розглянуто методи та інструменти розробки веб-сервісів, досліджено особливості проєктування застосунків на основі веб-сервісів, а також виконано проєктування цільового програмного рішення.

**У третьому розділі** реалізовано модуль моделювання та проєктування веб-сервісу, виконано моделювання RESTful-сервісів та продемонстровано функціональність розробленого механізму.

**Висновок:** у результаті виконання роботи було досліджено підходи до інженерії програмного забезпечення для веб-сервісів та реалізовано програмне рішення, що підтвердило ефективність використання сервісно-орієнтованих технологій для створення масштабованих, надійних і гнучких веб-систем.

**КЛЮЧОВІ СЛОВА:** ВЕБ-СЕРВІС, ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, REST, RESTFUL API, СЕРВІСНО-ОРІЄНТОВАНА АРХІТЕКТУРА, КОМПОЗИЦІЯ СЕРВІСІВ, ПРОЄКТУВАННЯ ПЗ, ВЕБ-ТЕХНОЛОГІЇ, API.

# ANNOTATION

The thesis consists of 78 pages, 28 figures, 12 tables, and a reference list containing 40 sources.

**The aim of the work** is to study software engineering principles for web services and develop a software solution demonstrating the design, modeling, and implementation of modern web services using the REST architectural style.

**Research object**: processes of web service development and operation.

**Research subject:** methods, models, technologies, and tools of software engineering used for designing, implementing, and testing web services.

**Research results:** a web service was developed demonstrating modern approaches to software design, service composition, RESTful architecture modeling, and interaction mechanisms between software components.

**The first section** analyzes the theoretical foundations of software engineering for web services, examines approaches to software composition, and studies the taxonomy of web services.

**The second section** discusses methods and tools for web service development, investigates application design based on web services, service description and composition languages, and presents the design of the target software solution.

**The third section** implements a web service modeling and design module, performs RESTful service modeling, and demonstrates the functionality of the developed mechanism.

**Conclusion:** the study investigated modern approaches to software engineering for web services and resulted in the implementation of a software solution that confirmed the effectiveness of service-oriented technologies for building scalable, reliable, and flexible web systems.

**KEY WORDS**: WEB SERVICE, SOFTWARE ENGINEERING, REST, RESTFUL API, SERVICE-ORIENTED ARCHITECTURE, SERVICE COMPOSITION, SOFTWARE DESIGN, WEB TECHNOLOGIES, API.

# ЗМІСТ

# ВСТУП

У сучасному світі веб-сервіси є однією з основних складових цифрової інфраструктури, забезпечуючи взаємодію між програмними системами, обмін даними та реалізацію бізнес-процесів у різних сферах діяльності. Стрімкий розвиток інформаційних технологій, збільшення обсягів даних та необхідність інтеграції різнорідних програмних рішень зумовлюють потребу у створенні надійних, масштабованих і гнучких веб-сервісів. Саме тому важливого значення набуває інженерія програмного забезпечення, яка забезпечує системний підхід до проєктування, реалізації та супроводу програмних систем.

**Актуальність теми** зумовлена широким використанням веб-сервісів у сучасних інформаційних системах та необхідністю застосування ефективних методів і технологій для їх розроблення. Існуючі програмні рішення потребують забезпечення високої продуктивності, безпеки, масштабованості та можливості інтеграції з іншими сервісами. Використання принципів інженерії програмного забезпечення дозволяє підвищити якість програмних продуктів, скоротити витрати на їх підтримку та забезпечити ефективне функціонування в умовах постійного розвитку інформаційних технологій.

**Метою роботи** є дослідження принципів інженерії програмного забезпечення для веб-сервісів та розроблення програмного рішення, що демонструє процес проєктування, моделювання та реалізації сучасних веб-сервісів.

Для досягнення поставленої мети необхідно виконати такі завдання: провести аналіз теоретичних основ інженерії програмного забезпечення для веб-сервісів, дослідити підходи до композиції програмного забезпечення та таксономії веб-сервісів, розглянути методи проєктування застосунків на основі веб-сервісів, проаналізувати мови опису та композиції сервісів, виконати проєктування цільового програмного рішення, реалізувати модуль моделювання та проєктування веб-сервісу, а також оцінити ефективність його функціонування.

**Об’єктом дослідження** є процеси розроблення та функціонування веб-сервісів у сучасних інформаційних системах.

**Предметом дослідження** є методи, моделі, технології та інструменти інженерії програмного забезпечення, що застосовуються для проєктування, реалізації та тестування веб-сервісів.

**Методи дослідження** включають аналіз наукових джерел, порівняльний аналіз сучасних технологій розробки веб-сервісів, методи моделювання програмних систем, принципи сервісно-орієнтованої архітектури та експериментальні методи перевірки працездатності програмного забезпечення.

**Практичним результатом** роботи є розроблення веб-сервісу, який демонструє використання сучасних підходів до проєктування та реалізації програмних систем на основі RESTful-архітектури. Особлива увага приділяється забезпеченню масштабованості, надійності, повторного використання програмних компонентів та ефективної взаємодії між сервісами.

Бакалаврська робота містить 78 сторінок, 28 рисунків, 12 таблиць, 3 розділи, список використаних джерел із 40 найменувань.

# РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ВЕБ-СЕРВІСІВ

## 1.1 Основи інженерії програмного забезпечення для веб-сервісів

Останнім часом пропонування програмного забезпечення у формі веб-сервісів набуло популярності завдяки розвитку хмарних архітектур та впровадженню концепцій Інтернету речей. Як і очікувалося, були розроблені веб-сервіси наступного покоління, які революціонізували спосіб розробки програмного забезпечення. З моменту його впровадження, архітектурний стиль *передачі репрезентативного стану (REST)* дедалі більше користується перевагою розробників завдяки своїй простоті та масштабованості, і практично став сучасним практикою для створення веб-сервісів. REST складається з набору правил та практик, що пропонують прості зрозумілі API, чіткі представлення та масштабовані сервіси.

Тим часом, з точки зору автоматизованої розробки програмного забезпечення, *модельно-керована інженерія (MDE)* набирає популярності. MDE та її найвідоміший приклад, *модельно-керована архітектура* (MDA), представлена *Open Management Group* (OMG) , *дійсно* приносять певні переваги своїм практикам. За даними, фахівці MDE повідомляють, що це призводить до вищого ступеня автоматизації та підвищення продуктивності, важливим аспектом якої є генерація коду. Крім того, вони повідомляють, що MDE призводить до покращення якості та зменшення кількості дефектів, які виявляються раніше на життєвому циклі проекту, отже, їх можна виправити з меншими витратами [1]. Нарешті, інші стверджують, що MDE прискорює впровадження нових вимог та підвищує зрозумілість. Завдяки своїй прозорій ресурсоорієнтованій моделі, парадигма RESTful була включена до кількох фреймворків розробки, деякі з яких вбудовують методології MDE, що дозволяють швидку розробку та прагнуть автоматизувати частини процесу розробки. Деякі з цих систем спрямовані на створення скелета, а в деяких

випадках і схеми бази даних для ресурсів, тоді як інші створюють повнофункціональні веб-сервіси. Однак більшість з них вимагають участі розробника в тій чи іншій формі моделі/мови, що наражає розробника на найпоширеніші недоліки MDE. Ці недоліки полягають у значних витратах на навчання, необхідних для успішного використання цієї методики, та складності процесу моделювання, який вважається складним і вимагає занадто багато зусиль.

У цій роботі ми намагаємося пом'якшити вищезгадані недоліки RESTful сервісних фреймворків, яким також бракує широко поширених методів інженерії вимог. Отже, ми представляємо синергію верхнього та нижнього CASE. Технології та інструменти штучного інтелекту, такі як *обробка природної мови (NLP),* онтології та логіка предикатів першого порядку з кількома основними засобами програмної інженерії верхнього/нижнього CASE, а саме: управління мультимодальними вимогами, предметно-орієнтований метамоделювання та ланцюги перетворення MDE [2]. Результатом є узгоджений механізм, який дозволяє швидке прототипування та розробку функціональних RESTful сервісів, тоді як розробник може моделювати свою уявну систему, використовуючи комплексні представлення вимог, тобто текстові вимоги та візуальні розкадровки, без необхідності подальшого навчання предметно-орієнтованим мовам або зіткнення зі складними діями моделювання, які вбудовуються в чисті методи MDE.

Зокрема, ми представляємо двоскладовий механізм, який допомагає розробникам у створенні RESTful сервісів. Його серверна частина з нижніми літерами використовує механізм MDE, який автоматизує генерацію RESTful сервісів, що відповідають 3-му рівню *моделі зрілості Річардсона (RMM* ). Він пропонує автоматичну функцію базової автентифікації, автоматизує процес - пошуку за популярними ключовими словами в базі даних та забезпечує сумісність передбачуваного RESTful сервісу з існуючими сторонніми сервісами.

Крім того, як уже згадувалося, на відміну від більшості механізмів перетворення, які отримують вхідні дані у вигляді моделі або мови, наш механізм

MDE поєднаний з інтерфейсом верхнього CASE, який дозволяє розробникам проектувати свою уявну систему за допомогою вимог до програмного забезпечення в мультимодальних форматах. Зокрема, верхній CASE ефективно моделює статичні та динамічні представлення уявної системи та використовує методи NLP та семантику для перетворення функціональних вимог та сценаріїв динамічної системи на специфікації системи [3]. Отримані представлення створюються в онтологіях програмного забезпечення за допомогою інструментів, спеціально розроблених та розроблених для побудови REST-сумісних моделей з функціональних вимог та розкадровок (тобто графічних - сценаріїв). Функціональні вимоги аналізуються та семантично анотуються для створення RESTful ресурсів та властивостей, тоді як розкадрові дошки використовуються для визначення гіпермедійних зв'язків між ресурсами.

Поєднання модуля *«вимоги до специфікацій» (Reqs2Specs)* з механізмом MDE призводить до створення комплексної та ефективної системи з численними перевагами. Вона впроваджує підвищену автоматизацію в розробці RESTful сервісів завдяки використанню NLP та семантики для побудови вхідної моделі механізму MDE [4]. Потрібно менше зусиль для перетворення вимог у специфікації, тоді як побудова прототипів веб-сервісів є простою, навіть з мінімальними знаннями MDE та парадигм RESTful. Крім того , наша методологія спрощує міграцію застарілих програмних систем, розроблених традиційними методами, до сфери MDE, оскільки вона здатна приймати як вхідні дані їхні текстові вимоги. Нарешті, зауважте, що зміни у вимогах до програмного забезпечення миттєво поширюються на базові програмні артефакти та моделі, таким чином забезпечуючи високу узгодженість та відстежуваність моделі.

Композиція сервісів охоплює всі процеси, які створюють сервіси з доданою вартістю, які називаються *композитними* або *агрегованими* сервісами, з існуючих сервісів. Необхідність інтеграції сервісів визнається як на корпоративному, так і на споживчому рівнях. У звіті Gartner зазначено, що «до 2014 року акт композиції буде кращою можливістю отримати цінність від програмного забезпечення, ніж акт розробки». Зовсім недавно акцент змістився

на більш конкретні типи композицій: звіт Gartner «10 головних стратегічних технологічних тенденцій на 2014 рік» підтверджує важливість композиції хмарних сервісів, а Forrester у своїх «Прогнозах на 2015 рік для фахівців з розробки та доставки додатків» виділяє композицію для мобільних додатків (тобто складання компонентів інтерфейсу) як наріжний камінь для підвищення продуктивності розробки, використовуючи веб-компоненти та платформи, такі як HTML5 та Google Polymer, бібліотеку для створення цих компонентів [5].

Композиція сервісів – це дуже активна галузь досліджень, і більше того, у багатьох випадках проводяться дослідження в галузі робочих процесів та програмних компонентів, які можна використовувати. Об'єднуючи веб-сервіси та технології семантичного веб-сервісу, парадигма семантичних веб-сервісів обіцяє зробити крок вперед у покращенні композиції веб-сервісів завдяки багатим та зрозумілим для машинного розуміння представленням властивостей та можливостей сервісів, а також механізмам міркування для вибору та агрегації сервісів.

Кілька варіацій моделей опису сервісів, протоколів взаємодії, а також мов і методів композиції. Загалом, з одного боку, основні методи композиції зосереджені переважно на сервісах додатків та даних. Логіка композиції зазвичай спирається на процедурний потік або мови сценаріїв [6]. Взаємодія з компонентними сервісами здійснюється через низькорівневі API та протоколи. Цікаво, що сучасні середовища композиції сервісів рідко надають інструменти підтримки продуктивності, подібні до тих, що надаються сучасними інтегрованими середовищами розробки (IDE). Наприклад, IDE підтримують розробників, забезпечуючи пошук коду, повторне використання, налагодження, генерацію тощо. Як і традиційні програмісти, композитори сервісів виграли б від середовищ, які об'єднують методи продуктивності, такі як виявлення сервісів, повторне використання, генерація коду, документація, інтеграція із середовищами композиції сервісів. Такі середовища можуть мати різні вимоги щодо моделей компонентів, протоколів взаємодії та конструкцій оркестрації; наприклад, оркестрація хмарних сервісів вимагає координації апаратних та

програмних ресурсів на різних рівнях. Окрім традиційних конструкцій потоку керування та потоку даних, також існує потреба в абстракціях для підтримки узгодженості на фізичних та логічних рівнях, обробки винятків, а також гнучкої, ефективної координації, вибору та планування ресурсів.

У цій роботі ми пропонуємо чітко сформульовану структуру для аналізу та порівняння підходів до композиції веб-сервісів. Попередні роботи в основному зосереджувалися на конкретних аспектах інженерії сервісів, включаючи виявлення сервісів та якість послуг [7]. Існуючі дослідження, що стосуються композиції веб-сервісів, здебільшого фрагментовані та не мають цілісного погляду на проблему. Ці дослідження досліджували конкретні аспекти композиції, такі як зручність використання, адаптивність та ефективність механізмів композиції сервісів, динамічна композиція сервісів (що полягає в автономному складанні програми, коли користувач запитує програму під час виконання), мови композиції сервісів на основі процесів, мови та моделі, веб-машапи, наукові робочі процеси, життєвий цикл, якість обслуговування (QoS) композиції, мови та протоколи опису послуг та взаємодії, а також семантична композиція послуг [8]. Очевидно, що попередні зусилля дослідницьких та практичних спільнот дали багатообіцяючі результати, які, безумовно, є корисними. Однак, потрібна більш консолідована та цілісна аналітична структура, яка б просунула фундаментальне розуміння структурних блоків композиції веб-сервісів з точки зору концепцій, моделей, мов, методів підтримки продуктивності та інструментів. Ця структура необхідна для ефективного дослідження, розуміння, оцінки, порівняння та вибору моделей, мов, методів, платформ та інструментів композиції послуг.

## 1.2 Розробка програмного забезпечення на основі композиції веб-сервісів

Базовим інгредієнтом будь-якої складеної програми є програмні компоненти (ми використовуємо терміни *«програмний компонент»* та

*«компонент»* як взаємозамінні). Вони інкапсулюють *функціональність, дані* та/або *інтерфейс користувача* (UI), який можна використовувати повторно, і надають набір *операцій,* що дозволяють програмно взаємодіяти з інкапсульованою функціональністю та/або інтерфейсом користувача. Прикладами типових компонентів, які ми маємо на увазі, є SOAP та RESTful веб-сервіси.

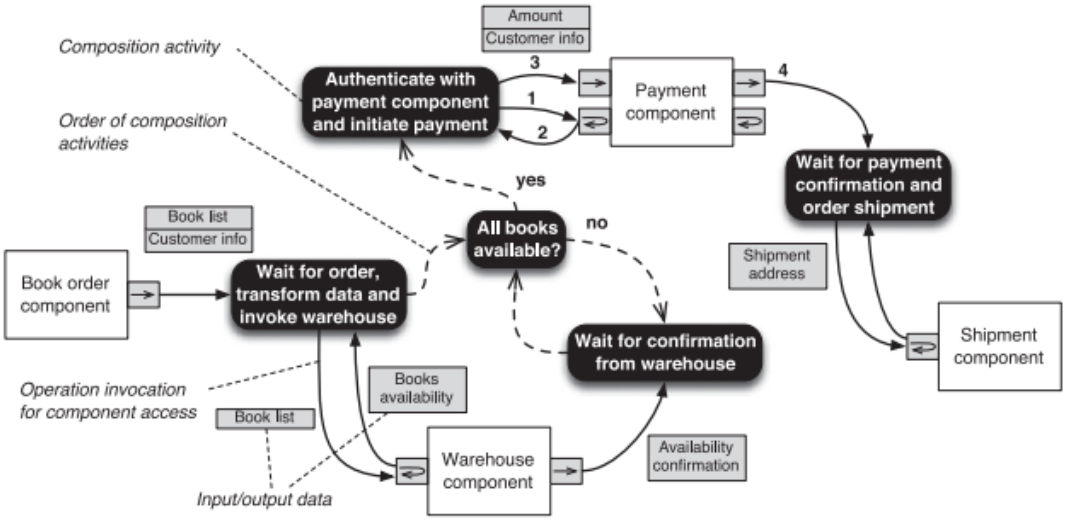


Рисунок 1.1 - Композитний додаток для обробки замовлень. Чорним шрифтом і жирним шрифтом виділено дії логіки композиції, що склеює компоненти.

На рисунку 1.1 показано внутрішню структуру програми для замовлення книг, що використовує чотири компоненти. Це дозволяє нам описати основні проблеми, які необхідно вирішити:

— *Доступ до компонентів:* Використання різних компонентів вимагає опанування різних моделей компонентів та підтримки механізмів їх доступу. Тобто, може знадобитися викликати операції, надсилати сповіщення та перехоплювати події [9].

—*Керування розмовами:* Компонент також може супроводжуватися бізнес-протоколом, який є специфікацією, що вказує, в якому порядку мають виконуватися операції компонента, щоб встановити правильну розмову з компонентом. Наприклад, компонент «Платіж» на рисунку 1.1 вимагає від своїх

клієнтів спочатку автентифікації в компоненті (кроки 1 та 2), потім ініціювання процесу оплати (3), а потім очікування події завершення (4).

— *Потік керування:* Композиційні дії, як правило, не можуть виконуватися випадковим чином; наприклад, немає сенсу викликати компонент «Склад» за відсутності впорядкованого списку книг. Для досягнення заданої мети необхідно впорядкувати композиційні дії. Наприклад, після отримання замовлення композиція на рисунку 1.1 негайно викликає компонент «Склад», формуючи так звану *послідовність* викликів. Замовлення дій також може вимагати прийняття *рішень* щодо того, яку дію виконувати далі. Наприклад, перевірка «Усі книги доступні?» або запускає процедуру оплати, або очікує підтвердження складом наявності всіх книг, залежно від наявності замовлених книг [10].

—*Потік даних:* У композиції вхідні дані одного компонента зазвичай виробляються іншим компонентом як вихідні (якщо вони не надаються самою логікою композиції). Наприклад, компонент «Склад» на рисунку 1.1 вимагає списку книг як вхідних даних, які надаються компонентом «Замовлення книг» як вихідні дані. Визначення того, як вхідні дані походять від виходів, визначає так званий потік *даних* між компонентами.

— *Трансформація даних:* У більшості випадків виходи та вхідні дані не одразу збігаються, наприклад, за форматами даних або розміром. Тому поширення даних від одного компонента до іншого може вимагати проміжного етапу *перетворення даних* (наприклад, переформатування , розділення або об’єднання), щоб зробити компоненти сумісними. Наприклад, список книг для компонента «Склад» є підмножиною даних, створених компонентом «Замовлення книг».

Вирішення вищезазначених проблем для всіх компонентів, що підлягають інтеграції, створює специфікацію того, як має поводитися композитний застосунок [11]. Окрім цих особливостей композиції, розробка композитного застосунку може також вимагати вирішення низки *перехресних питань,* таких як угоди про безпеку та рівень обслуговування.

*Оркестрування проти хореографії*

Приклад композиції, проілюстрований на рисунку 1.1, зосереджується на внутрішній роботі композитного застосунку та координації дій композиції та компонентів. Це робиться з *централізованої точки зору,* виражаючи, як має діяти композиція, щоб інтегрувати компоненти. Результатом є *виконуваний проект* композитного застосунку, який називається *оркестрацією* [12].

Композитні додатки також можуть бути розроблені з *розподіленої точки зору* , коли постачальники компонентів, що беруть участь у композиції, спільно домовляються про спільний протокол зв'язку для своїх компонентів. Результатом є контракт *(*протокол) між партнерами, що співпрацюють, який не є виконуваним і має бути реалізований всередині кожного компонента окремо. Цей контракт зазвичай відомий як *хореографія* [13]. Хореографії особливо корисні в тих ситуаціях, коли кілька сторін повинні співпрацювати, але жодна з них не хоче брати на себе відповідальність за управління централізованою оркестрацією (наприклад, у транзакціях B2B).

## 1.3 Таксономія та класифікація веб-сервісів

Основна увага в цьому огляді зосереджена на *композиції програмних сервісів повторного використання,* зокрема *оркестраціях* . Також, незважаючи на схожість з програмуванням, ми тут не вивчаємо придатність універсальних мов програмування для розробки композитних додатків. Наша увага зосереджена на всіх тих підходах, мовах та інструментах, які пропонують *композиційно-специфічні* абстракції та рішення для розробки.

Простір композиції сервісів, обмежений цими припущеннями, все ще величезний. Для полегшення цілеспрямованого аналізу ми розробили таксономію, проілюстровану на рисунку 1.2, яка пропонує *цілісний погляд* на проблему композиції на основі оркестрації та має на меті висвітлити ключові аспекти та варіанти, з якими доводиться стикатися під час створення програмного забезпечення. Таксономія базується на нашому власному досвіді в

композиції веб-сервісів, управлінні бізнес-процесами та мэшапах, а також на великому огляді літератури у суміжних галузях, обговореннях з колегами, експериментах з різними системами та прототипами, що дозволило нам визначити спільні будівельні блоки для різних варіацій систем композиції сервісів [14].

У попередньому підрозділі ми обговорили, *що* означає розробка композиції. Наша таксономія розглядає, *як* можна розробляти композиції незалежно від технологій чи цільових композитів, а також як можна *допомогти процесу композиції,* щоб спростити розробку. Частина цієї проблеми також вимагає розуміння *того, хто* насправді створює програмне забезпечення та які методи чи практики композиції підходять для якого профілю розробника. Відповідно, ми розділили проблему композиції на шість *вимірів,* які, у свою чергу, можна розділити на *підвиміри:*

— *Мова композиції* **:** це основний вимір нашого аналізу. Мова визначає, як відбувається композиція, які композиційні дії підтримуються та як. Згадуючи Рисунок 1, мова композиції - це формалізм, який дозволяє виражати логіку композиції та виконувати композитну програму. Враховуючи її вирішальну роль у процесі композиції, ми далі розділяємо вимір мови на підвиміри та розглядаємо різні *компоненти* та *цільові програми,* які може підтримувати мова, конкретні *нотації* та *парадигми,* що використовуються, а також *композиційні конструкції* та *наскрізні питання* .

— *Повторне використання знань* **:** Визнаючи природу композиції, що базується на повторному використанні (ця практика базується на повторному використанні існуючих програмних компонентів), ми детальніше аналізуємо, які *артефакти композиції* існують і можуть бути використані повторно. Розробка композитної програми може бути нетривіальною, а композиція ще не є такою поширеною практикою, як можна було б подумати. Ефективне повторне використання - і відповідні *методи повторного використання* –
таким чином відіграють особливу роль у контексті композиції.

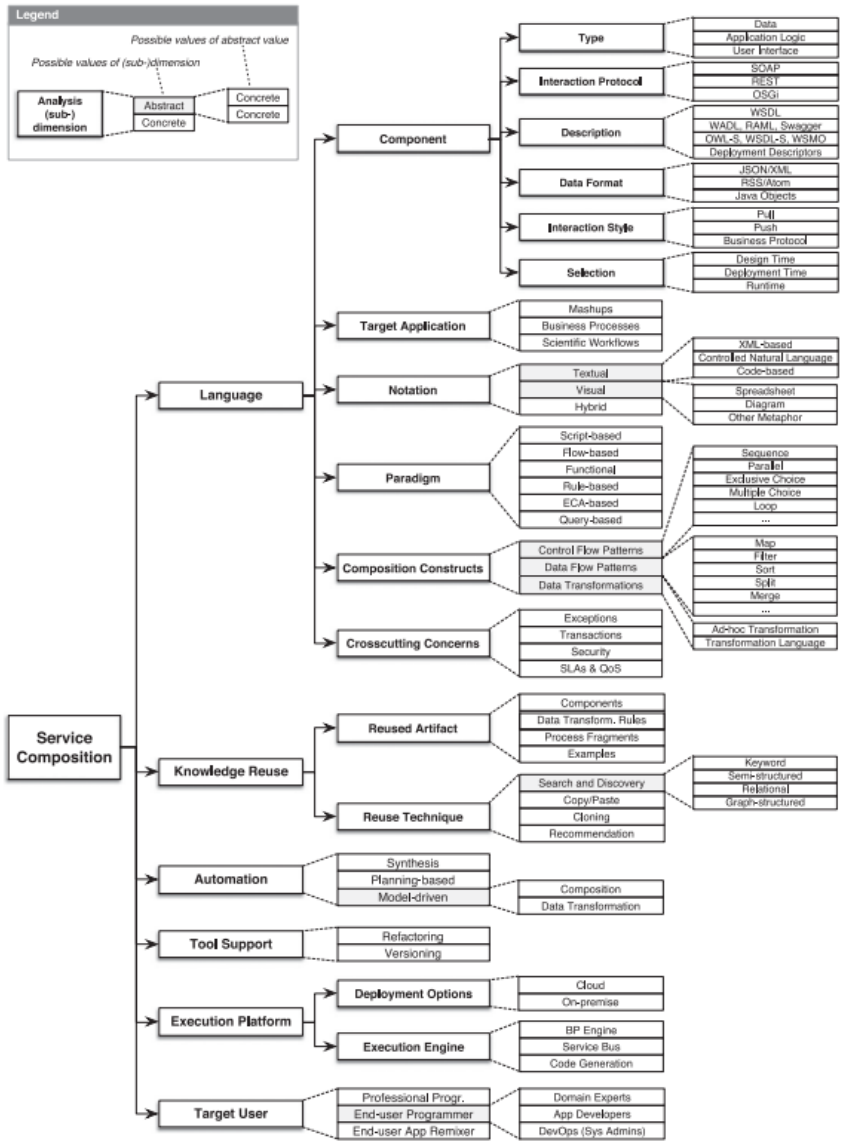


Рисунок 1.2 - Таксономія композиції веб-сервісів.

— *Автоматизація :* Завдяки наявності спеціалізованих мов композиції, які характеризуються відносно обмеженим набором композиційних конструкцій, та компонентів повторного використання, які інкапсулюють більшу частину складності композитного застосунку, простір рішень композиції зазвичай набагато менший порівняно, наприклад, з розробкою програмного забезпечення за допомогою універсальних мов програмування. Це спостереження надихнуло на розробку набору автоматизованих підходів до композиції та практик композиції на основі моделей, ролі та цінність яких важливо розуміти.

— *Підтримка інструментів* **:** Попередні три виміри конкретно аналізують, як можна розробити композитний додаток. В ідеалі, але не

обов'язково, цей процес підтримується відповідним інструментом розробки або інтегрованим середовищем розробки (IDE), яке допомагає розробнику протягом усього процесу розробки за допомогою інфраструктури або функціональності, незалежної від додатків.

— *Платформа виконання :* одним із ключових аспектів композиції є те, як розгортаються та виконуються готові композитні програми. Розгортання може підтримуватися за допомогою різних опцій , що спрощують чи ні роботу розробника (наприклад, подумайте про хмарні обчислення та парадигму «як-послуга»). Виконання може підтримуватися різними *механізмами виконання, що* характеризуються їхньою архітектурною структурою та мовами композиції, які вони підтримують.

— *Цільові користувачі* : Оскільки композиційні підходи спрямовані на спрощення розробки застосунків порівняно з генеричним програмуванням, незабаром з'явилися міркування щодо навичок, необхідних для успішного створення застосунку. За допомогою цього виміру ми прагнемо зрозуміти, які типи цільових користувачів (від професійних розробників до кінцевих користувачів) існують і які практики композиції вони здатні опанувати [15].

### 1.4 Висновки по розділу

У першому розділі досліджено теоретичні основи інженерії програмного забезпечення для веб-сервісів. Розглянуто принципи створення та функціонування веб-сервісів, особливості сервіс-орієнтованого підходу до розробки програмного забезпечення та можливості композиції сервісів для побудови складних інформаційних систем. Проведено аналіз таксономії та класифікації веб-сервісів, що дозволило визначити їх основні характеристики, переваги та сфери застосування. Отримані результати формують теоретичну базу для подальшого проєктування та реалізації веб-сервісу.

# РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ РОЗРОБКИ ВЕБ-СЕРВІСІВ

## 2.1 Проєктування застосунків на основі веб-сервісів

У цьому підрозділі описано нашу пропозицію щодо напівавтоматичної генерації семантичних специфікацій веб-застосунку, сумісних з WSMO. Наш підхід розширює методологію WebML, на проектування семантичних веб-сервісів та веб-застосунків. На рис. 2.1 підсумовано передбачуваний процес розробки. Основний процес проектування, що підтримується традиційною веб-технологією [16], безперешкодно веде дизайнера від моделювання процесу до запущеного веб-застосунку, створюючи деякі проміжні артефакти (моделі BPMN, скелети WebML, моделі даних, моделі гіпертексту) та делегуючи частину виконання середовищу семантичного виконання (наприклад, WSMX). Такі моделі збагачуються імпортованими онтологічними описами (у верхній частині рисунка) та використовуються для розробки набору специфікацій WSMO (внизу рисунка): онтологія походить від моделі BP, моделі даних та моделі гіпертексту; опис можливостей веб-сервісів походить від моделі гіпертексту; інформація про хореографію походить від моделі BP та моделі гіпертексту; цілі користувача походять від моделі BP.

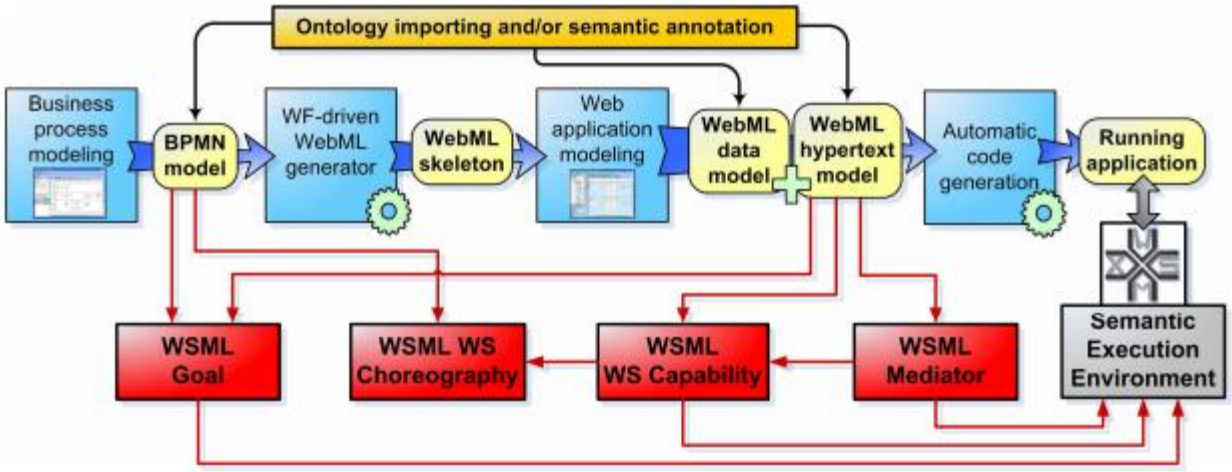


Рисунок 2.1 - Загальна схема підходу.

*Проектування бізнес-процесу*

Завдання проектування бізнес-процесів (BP), що зосереджується на високорівневій схематизації процесів, що лежать в основі застосунку, призводить до створення однієї або кількох діаграм BP. Читач може звернутися до [17] для отримання методології проектування веб-застосунків на основі бізнес-процесів. Діаграма BP для поточного випадку представлена на рис. 2.2 з чітко визначеною семантикою робочого процесу: для ясності процес розділено на два підпроцеси: частина (a) описує закупівлю, а частина (b) описує управління відвантаженнями. Далі ми наведемо приклади проектування медіатора частини (a) та вилучення онтології, можливостей та хореографії частини (b).

*Проектування моделі даних та вилучення онтологій*

Виявлення онтологій, що беруть участь у застосунку, здійснюється за допомогою чотирьох послідовних кроків, кожен з яких стосується різних аспектів онтології застосунку (див. рис. 2.1). Цей структурований підхід дозволяє системно перейти від загального аналізу домену до конкретної формалізованої моделі, яка може бути використана для подальшого проектування бази даних, семантичної інтеграції та забезпечення інтероперабельності системи.:

1. По-перше, можна імпортувати існуючі віддалені онтології, можливо, надані третіми сторонами.
2. Потім модель даних розглядається як частина онтології. Це означає, що відповідне перетворення моделі даних WebML перетворює її на онтологію, сумісну з WSMO, яка потім реєструється в менеджері ресурсів WSMX [18];
3. Потім онтологія процесу витягується зі специфікації BPMN. Елементи моделі робочого процесу (наприклад, назви дій, смуги руху) витягуються як семантичні концепції та використовуються як додаткова частина онтології, яка буде корисною для визначення сигнатури стану інтерфейсів хореографії веб-сервісів;
4. Нарешті, модель BPMN та модель даних WebML анотуються концепціями, імпортованими з існуючих онтологій.

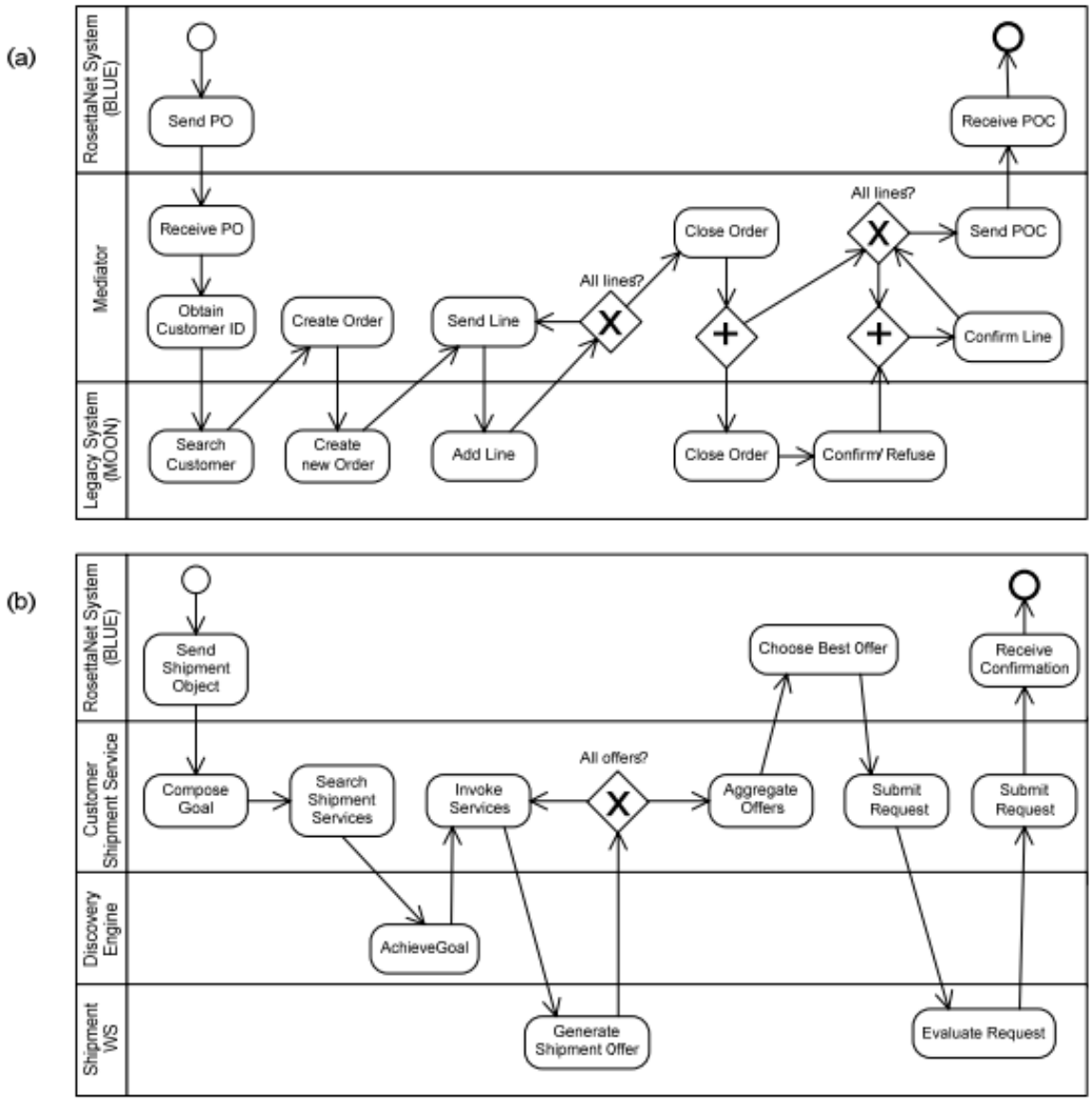


Рисунок 2.2 - Робочий процес, що відображає взаємодію запущеного прикладу.

На рис. 2.3 показано модель даних, що використовується веб-сервісом відвантаження. Вона включає три основні сутності домену: *Shipment*, *ShipmentService* (партнери з відвантаження) та *Location* (географічні місця). Діаграма включає сутності *Case* та *Activity*. Кожне *відвантаження* пов'язане з *ShipmentService*, з місцем відправлення та місцем призначення *Location*, а також з *Activity* , що вказує на його поточний стан. *ShipmentService* пов'язаний з *Location* через відношення *shipTo*, яке описує набір можливих місць відвантаження для кожного партнера; відношення *hasLocation* визначає набір дійсних точок отримання для кожного перевізника.

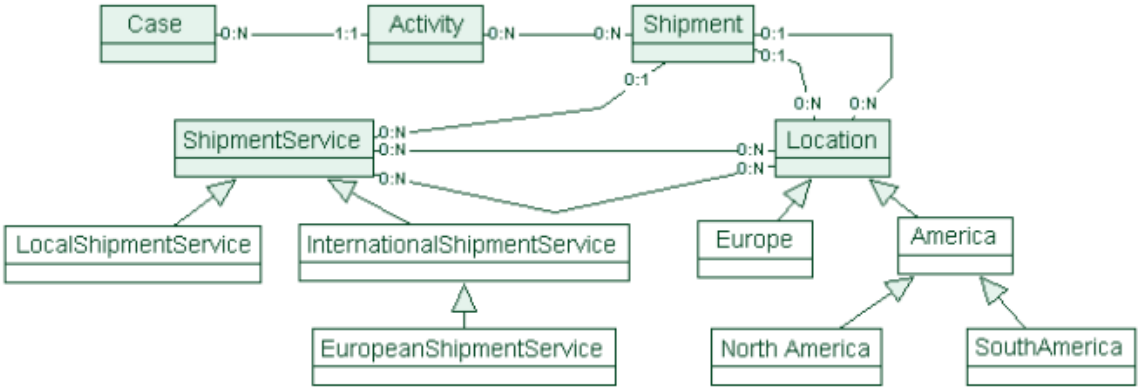


Рисунок 2.3 - Фрагмент моделі даних WebML, що використовується веб-службою відвантаження.

Модель даних WebML можна легко перетворити на онтологію WSML-Flight, зберігаючи всі її обмеження. Наприклад, сутність *EuropeanShipmentService* є підсутністю InternationalShipmentService , яка розташована в *Європі*. Ця підсутність описується в синтаксисі WebML-OQL так:

```
InternationalShipmentService(as SuperEntity) where
   InternationalShipmentService.hasLocation isa Europe.
```

Рисунок 2.4 - Сутність EuropeanShipmentService

Його переклад на WSML-Flight такий:

```
concept EuropeanShipmentService subConceptOf InternationalShipmentService
   nfp dc#relation hasValue { EuShipmentServiceDef } endnfp
axiom EuShipmentServiceDef
   definedBy
?x memberOf InternationlShipmentService
and hasLocation(?x,?nation) and ?nation memberOf Europe
implies ?x memberOf EuropeanShipmentService.
```

Рисунок 2.5 - Перетворення на онтологію WSML-Flight

Процес генерації онтологій WSML починається з імпорту зовнішніх онтологій, що використовуються в моделі даних WebML, для збагачення визначень типів даних WebML. Потім для кожної сутності в моделі даних

генерується відповідна концепція WSML з її безпосередньою суперконцепцією, атрибутами (також зв'язки відображаються на атрибути) та можливими аксіомами.

*Дизайн сервісу та інтерфейсів користувача у WebML*

Після розробки бізнес-процесу обмеження робочого процесу необхідно перетворити на обмеження навігації між сторінками дій гіпертексту та на запити даних до метаданих робочого процесу для перевірки стану процесу. Це стосується як елементів контенту, що споживається людиною (тобто інтерфейсів сайту), так і контенту, що споживається машиною (тобто взаємодій із семантичними веб-сервісами).

Для перетворення моделей робочих процесів на скелети гіпертекстових діаграм WebML було розроблено гнучке перетворення, яке залежить від кількох параметрів налаштування та стилізації [19]. Оскільки описи дій не передбачають апріорної семантики, згенерований скелет може бути реалізований лише з гіпертекстом та запитами, необхідними для забезпечення дотримання обмежень робочого процесу. Розробник залишається відповідальним за реалізацію внутрішніх механізмів кожної дії. Крім того, можна анотувати дії, що дозволяє автоматично генерувати грубий гіпертекст, який реалізує задану поведінку, яку потім потрібно уточнити розробнику.

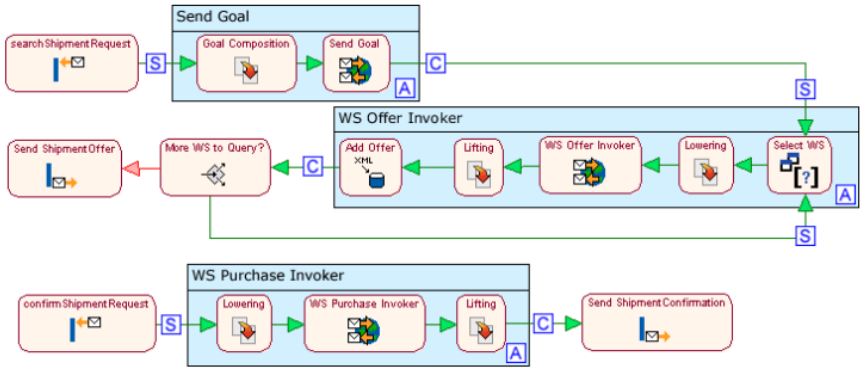


Рисунок 2.6 - Веб-сервіс Blue Shipment.

Наприклад, на рис. 2.6 показано можливу специфікацію WebML для сервісу Blue Shipment. У верхній частині рис. 2.6 представлено операцію

*searchShipmentRequest* : об'єкт ShipmentObject передається до *композиції цілі* , яка перетворює його на опис цілі для WSMX-сумісного механізму виявлення; отриманий опис цілі передається до функції *Send Goal,* яка надсилає ціль до веб-сервісу, що надається механізмом виявлення. Механізм виявлення повертає результат із набором веб-сервісів, сумісних з оригінальною метою відвантаження. Для кожного веб-сервісу застосовуються операції опускання та підйому за допомогою відповідного таблиці стилів XSLT. Потім для кожного повернутого веб-сервісу робиться запит на пропозицію відвантаження. Результати об'єднуються та перетворюються на модель даних Blue, а набір пропозицій повертається замовнику сервісу. Після того, як замовник сервісу вибирає одну з пропозицій та надсилає її до операції *confirmShipmentRequest* (нижня частина рис. 2.6), пропозиція купується шляхом виклику відповідного веб-сервісу, і повідомлення з підтвердженням надсилається назад.

*Вилучення опису веб-сервісів*

Ще одним важливим аспектом, який можна напівавтоматично отримати зі специфікації проекту, є опис веб-сервісів. Деяку інформацію про сервіси можна безпосередньо витягти з високорівневого опису взаємодій BPMN (зокрема, інформацію про можливу хореографію сервісу та базову специфікацію інтерфейсу та параметрів). Більш детальну інформацію можна отримати з діаграм WebML, які забезпечують більш чітке представлення специфікації застосунку.

Вилучення можливостей веб-сервісів. Моделі BPMN та WebML веб-сервісів надають достатньо інформації для опису їхньої поведінки. Припускаючи, що активність BPMN є атомарним викликом веб-сервісу, ми можемо використовувати *потік даних BPMN* для надання гарних підказок для вилучення вхідних та вихідних даних сервісу. Дійсно, потік даних визначає об'єкти, які передаються між різними активностями [20]. Виділяючи одну активність, можна автоматично вилучати попередні умови (входи) та пост-умови (виходи) WSML. Однак, зазвичай, потрібні уточнення дизайнером.

Попередні умови WSML отримуються з першого блоку ланцюжка

WebML, що описує операцію веб-сервісу (блок запиту), тоді як пост-умови отримуються з останнього (блок відповіді). Ці два блоки містять інформацію про точну структуру обмінюваного повідомлення та, зрештою, про відображення елементів повідомлення в моделі домену, а отже, і в видобутих онтологіях. Ефекти видобуваються шляхом пошуку блоків WebML, які змінюють або створюють екземпляри сутностей, пов'язаних з діями, що беруть участь у процесі, описаному у веб-сервісі WebML. Спільні змінні отримуються зі згенерованих умов шляхом групування всіх змінних, що беруть участь у потоці даних операцій [21].

Наведений нижче опис можливостей веб-сервісу у WSML генерується автоматично після повного визначення моделей WebML.

```
capability
   sharedVariables (?Req)
   precondition
      definedBy
         (?Req memberOf searchShipmentRequest) or
         (?Req memberOf ConfirmShipmentRequest).
   postcondition
      definedBy

                  (?Req[
                  pickupdate hasValue ?pkd, deliverydate hasValue ?dd,
                  start hasValue ?s, destination hasValue ?dest,
                  weight hasValue ?w, maxCost hasValue ?maxc
                  ] memberOf searchShipmentRequest)
                  implies
                  exists ?Res (
                     ?Res memberOf ShipmentOfferContainer and
                        forall ?offer (
                        ?Res [offers hasValue ?offer]
                        implies (
                           ?offer [
                              offerID hasValue ?OID, pickupdate hasValue ?pkd,
                              deliverydate hasValue ?dd, start hasValue ?s,
                              destination hasValue ?dest, weight hasValue ?w,
                              cost hasValue ?c] memberOf ShipmentOffer
                           and ?c<=?maxc
                  )))) and
                  (?Req[ offerID hasValue ?OID] memberOf ConfirmShipmentRequest)
                  implies
                     exists ?Confirmation (
                        ?Confirmation[
                           offerID hasValue ?OID, confirmationID hasValue ?CID
                        ] memberOf ShipmentConfirmation
                  ))
```

Рисунок 2.7 - Опис можливостей веб-сервісу у WSML.

Вилучення хореографії сервісу. Хореографія сервісу – це частина інформації, яка зазвичай вимагає певних анотацій від розробника, щоб встановити всі можливі послідовності взаємодії з сервісом. Однак, принаймні

одну з послідовностей хореографії можна вилучити з моделі BPMN, аналізуючи порядок виклику різних операцій сервісу. Очевидно, що це не гарантує, що всі можливі сценарії будуть розглянуті, оскільки можна проаналізувати лише один сценарій. Вилучення такої інформації досить просте: якщо доріжка описує один веб-сервіс, можна припустити, що всі посилання потоку керування, що перетинають її межі, сприяють визначенню можливого порядку виклику операцій, тобто інтерфейсу хореографії веб-сервісу. Автоматично згенерований WSML-опис хореографії веб-сервісу виглядає наступним чином:

```
interface
   choreography
      stateSignature
         in
            searchShipmentRequest withGrounding […]
            ConfirmShipmentRequest withGrounding […]
         out
            ShipmentOfferContainer withGrounding […]
            ShipmentConfirmation withGrounding […]
         controlled oasm#ControlState
         transitionRules
            forall {?x, ?state} with (
               ?state[oasm#value hasValue oasm#InitialState]
                  memberOf oasm#ControlState and
               ?x memberOf ShipmentRequest
            ) do
              add(?state[oasm#value hasValue ShipmentOfferRequested])
              delete(?state[oasm#value hasValue oasm#InitialState])
              add(_# memberOf ShipmentOfferContainer)
            endForall
            forall {?x, ?state} with (
              ?state[oasm#value hasValue ShipmentOfferRequested] and
              ?x memberOf ConfirmShipmentRequest) do
               add(_# memberOf ShipmentConfirmation)
            endForall
```

Рисунок 2.8 - WSML-опис хореографії веб-сервісу.

*Вилучення цілі користувача*

Вилучення цілей користувача може бути виконано шляхом об'єднання інформації, доступної на рівні BPMN, з інформацією, доступною на рівні WebML. Перший рівень виявлення цілей може бути досягнутий шляхом вилучення послідовності умов та об'єктів, що передаються веб-сервісам маршрутом користувача на діаграмі BPMN.

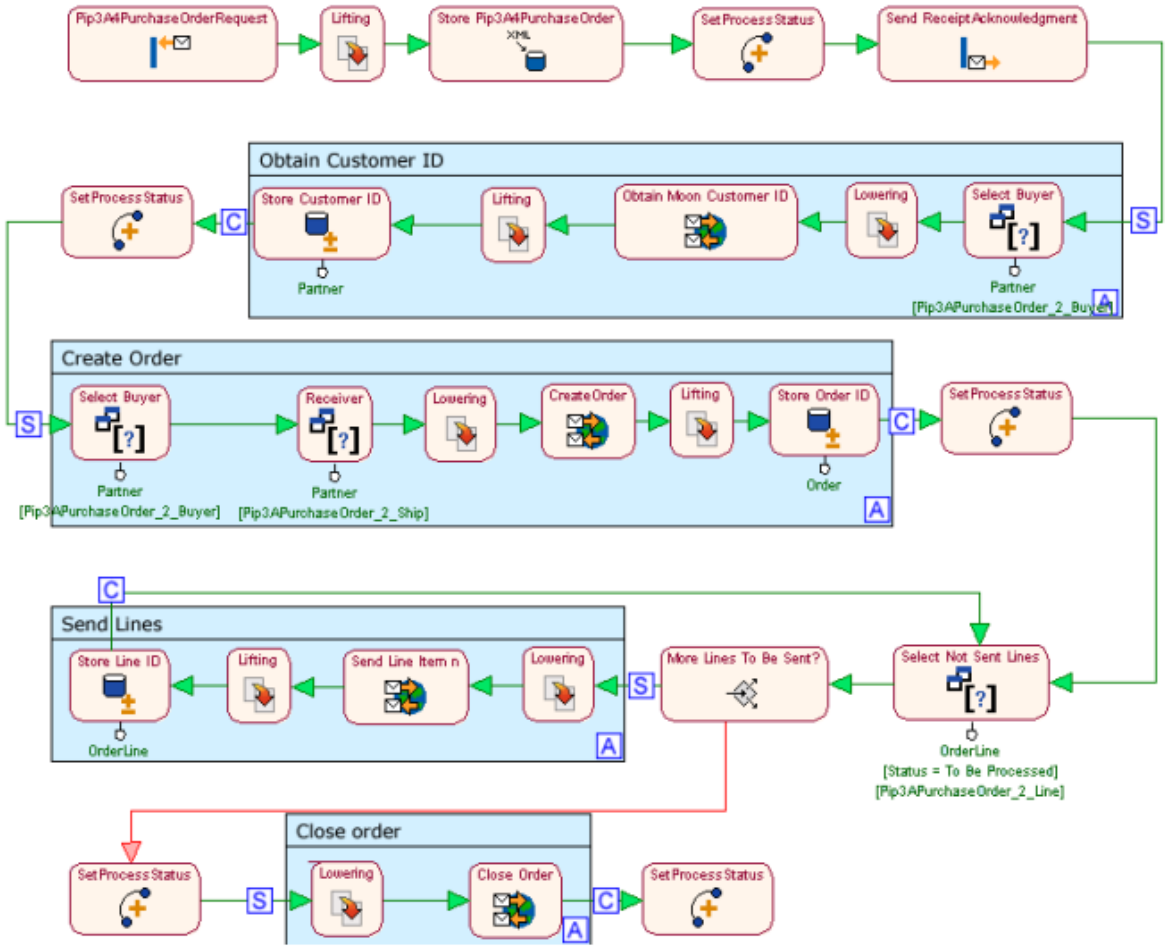


Рисунок 2.9 - Модель WebML веб-сервісу wwMediator.

Глибший рівень деталізації вимагає використання гіпертекстових моделей WebML та аналізу семантики, вбудованої в навігацію та композицію сторінок. Така уточнена мета деталізується з точки зору завдань, що виконуються користувачем, та даних, що маніпулюються, що підвищує значущість цілей WSMO, які можна згенерувати. У цьому випадку ми опускаємо автоматично згенерований код через обмеження простору.

*Дизайн wwMediators за допомогою WebML*

Однією з головних переваг цього підходу є простота проектування та впровадження складних wwMediators. Якщо смуга руху ідентифікована як wwMediator на рівні BPMN, основну інформацію про проектування медіаційних служб можна витягти з високорівневого опису взаємодій BPMN (зокрема, інформацію про можливу хореографію служби та базову специфікацію інтерфейсу та параметрів). Каркасна модель медіатора генерується автоматично,

і дизайнер може вдосконалити її на рівні концептуального проектування. Потім опис медіатора WSMO можна отримати з діаграм WebML.

На рис. 2.9 представлено детальну специфікацію wwMediator у WebML. Цю специфікацію можна використовувати для створення робочого веб-сервісу, що забезпечує посередництво між веб-сервісами Blue та Moon [22]. Специфікація WebML включає деякі операції *опускання* та *підняття,* що відповідають WSMO ooMediator, та забезпечує посередництво між моделлю даних веб-сервісу джерела та цільової моделі. У WebML це посередництво полягає у створенні таблиць стилів XSLT за допомогою візуального інструменту.

## 2.2 Мови опису та композиції веб-сервісів

Ми каталогізуємо шість підвимірів, а саме: тип, опис, формат даних, протокол взаємодії , стиль взаємодії та вибір, для характеристики компонентів. Тип вказує на те, який тип функціональності надається, опис – як вона рекламується, формат даних вказує на мову, яка використовується для представлення обмінюваних даних, протокол взаємодії визначає, як сервіси взаємодіють, стиль взаємодії виражає, як ініціюється зв'язок, а вибір вказує, коли сервіси вибираються для композиції.

*Тип.* Ми визначаємо три типи компонентів залежно від того, чи виступає компонент як джерело даних, чи надає доступ до логіки програми, чи має графічний інтерфейс.

—*Компоненти даних* складаються з сервісів, які містять веб-джерело у заданому форматі (наприклад, RDF+XML, RSS) з визначеною структурою даних та (можливо) зв'язками між елементами даних. Ці компоненти можна комбінувати для формування нових сервісів. Інструменти mashup зазвичай надають користувачеві можливість створювати сервіси даних (наприклад, Yahoo! Pipes підтримує візуальну агрегацію різних джерел даних).

—*Компоненти логіки застосунку* забезпечують бізнес-функціональність інших застосунків, таку як перевірка наявності товарів на складі, обробка

онлайн-платежів або запит на відправку товарів. Так звані пакети управління бізнес-процесами (BPM) забезпечують необхідну функціональність для інтеграції компонентів логіки застосунку для досягнення заданих бізнес-цілей [23]. Прикладами таких пакетів є BPM від Pega , Боніта BPM , та Oracle BPM .

—Компоненти *інтерфейсу користувача* включають контент, отриманий з веб-сторінок та віджетів інтерфейсу користувача, таких як віджети W3C, портлети Java або пропрієтарні формати [24]. Типовими віджетами інтерфейсу користувача є віджети входу, віджети карти та віджети пошуку. Композиція віджетів інтерфейсу користувача зазвичай відбувається всередині контейнерів або механізмів віджетів, які забезпечують створення та рендеринг віджетів і підтримують основні інфраструктурні послуги (наприклад, керування користувачами, постійне сховище, переадресацію URL-адрес). Liferay - це портал для інтеграції портлетів Java; Apache Rave це рушій для інтеграції W3C та OpenSocial віджети, тоді як забезпечують інтеграцію віджетів інтерфейсу користувача [25].

*Протокол взаємодії.* SOAP та REST є архетипними протоколами у веб - сервісах, і ми також включаємо OSGi як новий протокол, який пропонує альтернативу поширеному підходу SOA.

—*SOAP* (раніше Simple Object Access Protocol) - це простий протокол зв'язку на основі XML, який дозволяє обмін інформацією через HTTP та RPC. Сервіс SOAP базується на операціях, тобто він надає доступ до дій (операцій ), що виконуються веб-сервісом. Щодо формату повідомлень, SOAP визначає стандартний формат повідомлень для зв'язку, описуючи, як інформація повинна бути упакована в XML-документ. Мови, що підтримують композицію веб-сервісів на основі SOAP, включають BPEL та WS-CDL [26].

—*REST* (Representational State Transfer) – це архітектурний шаблон, який надає доступ до даних та функціональності через ресурси, доступ до яких здійснюється через виділені URL-адреси через HTTP. Сервіси REST мають шаблон запит-відповідь, де методи HTTP *Post , Get , Put* та *Delete* для заданого ресурсу зіставляються з відповідними операціями CRUD *Create, Read, Update* та

*Delete.* Відповіді сервісів містять представлення запитуваного ресурсу, представлене у форматах CSV, JSON, XML або подібних. Композиція сервісів RESTful була реалізована в таких мовах, як JOpera, яка має мову візуальної композиції, а також в інструментах mashup, таких як SABRE, що дозволяє інтеграцію джерел даних RESTful. Спробою забезпечити композицію сервісів RESTful у BPEL є розширення BPEL «BPEL for REST» [27].

—*OSGi* (Open Services Gateway Initiative) – це специфікація, яка визначає загальну та відкриту архітектуру для розробки, розгортання та управління сервісами всередині віртуальної машини Java (JVM). Вона містить спрощену модель сервісів, яка дозволяє публікувати, зв'язувати та асоціювати сервіси через реєстр сервісів. Сервіс– це звичайний об'єкт Java, семантично визначений його інтерфейсом сервісу, який зазвичай є інтерфейсом Java. Таким чином, формат даних – це об'єкти Java, і приклади мов, платформ або інструментів для створення таких сервісів все ще нечисленні; прикладами є SOA-платформи на основі Service Component Architecture (SCA), такі як Apache Tuscany і Паремус.

*Опис.* Ми групуємо різні способи опису компонентів у чотири групи різні лінії: SOAP (WSDL, SSDL), REST (WADL, RAML, Swagger), Semantic WS

(OWL-S, WSDL-S, WSMO) та дескриптори розгортання.

—*WSDL* (Web Service Description Language) - це специфікація на основі XML для опису веб-сервісів. Вона описує операції, що складають сервіс, повідомлення, якими обмінюється кожна операція, частини, що формують кожне повідомлення, та прив’язки протоколу.

— *WADL/RAML/Swagger:* WADL (Web Application Description Language) — це опис HTTP-додатків (зазвичай REST) на основі XML. WADL описує сервіс як набір ресурсів та їх зв'язків. RAM L (RESTful API Modeling Language) - це мова на основі YAML для опису RESTful API. Swagge r - це специфікація для опису RESTful веб-сервісів, що містить «інтерактивну» документацію, яка дозволяє створювати, використовувати та візуалізувати REST API.

— *OWL-S/WSDL-S/WSMO:* OWL-S (Онтологічна Веб-Мова для Сервісів, раніше DAML-S) – це онтологія, яка надає стандартний словник для

семантичного опису сервісів. Вона включає передумови та (умовні) ефекти в описі веб-сервісів, а також збагачені семантичні представлення вхідних та вихідних даних веб-сервісів [28]. Подібними технологіями є WSDL-S (Семантика Веб-Сервісів) та WSMO (Онтологія Моделювання Веб-Сервісів). Усі ці мови спрямовані на покращення виявлення, сумісності та композиції так званих Семантичних Веб-Сервісів.

— *Дескриптори розгортання:* Для компонентів інтерфейсу користувача, які потребують локального встановлення, дескриптори виконують подвійну функцію: вони описують інтерфейс компонентів і водночас служать конфігураторами розгортання. Віджети W3C містять відповідний файл config.xml, портлети Java - файл portlet.xml. Портлети, доступні віддалено через WRSP, базуються на загальних веб-сервісах і, отже, описуються за допомогою WSDL. Аналогічно, пакети сервісів OSGi (JAR-файли, що упаковують класи та ресурси Java) містять XML-файл з описом розгортання, тоді як сервіси OSGi з віддаленим доступом також надають відповідний опис WSDL.

*Формат даних.* Ми розділили різні формати обміну повідомленнями, що використовуються в композиції веб-сервісів, на три категорії:

—*JSON* (JavaScript Object Notation) - це легкий формат обміну даними [29]. Він є текстовим та зрозумілим для людини. Він був розроблений для представлення структурованих даних мовою сценаріїв JavaScript, але сьогодні JSON не залежить від мови, і всі основні мови програмування надають парсери JSON. JSON простіший і менш багатослівний, ніж *XML* (елемент XML має ім'я, а вміст укладено між парами відповідних тегів). Однак XML має багатшу семантику, наприклад, XML підтримує вузли різних типів і різні типи даних. Більшість середовищ композиції веб-сервісів, що підтримують RESTful-сервіси, підтримують JSON. Наприклад, Drupal, веб-фреймворк, який дозволяє інтеграцію API, надає бібліотеку для підтримки JSON як формату корисного навантаження.

— *RSS/Atom* - це формати синдикації на основі XML для обміну веб-стрічками. RSS та Atom використовуються для публікації веб-контенту, який

регулярно оновлюється (наприклад, дописи в блогах та онлайн-новини ). Велика кількість інструментів mashup, визнаних однією з найважливіших технологій Web 2.0, підтримують створення RSS/Atom-стрічок [30]. Прикладами є Damia або Yahoo! Pipes.

—*Об'єкти Java* є екземплярами класів Java. OSGi, наприклад, використовує об'єкти Java як формат обміну даними в композиції [31]. Лише кілька підходів зосереджені на композиції на основі об'єктів Java. Варто згадати, фреймворк для композиції SOAP, не-SOAP (наприклад, OSGi) та не-веб-сервісів, а також рішення в стилі BPEL для композиції сервісів OSGi.

*Стиль взаємодії.* Механізм, який підтримує компонент для зв'язку зі своїми клієнтами, визначає стиль взаємодії компонента.

— *Pull (Пускання):* Цей стиль використовується, коли клієнт явно викликає веб-сервіс за шаблоном запит-відповідь. Таким чином, зв'язок ініціюється клієнтом, і компонент не може зв'язатися з клієнтом, якщо немає конкретного запиту. Стратегія pull підтримується більшістю мов композиції, включаючи BPEL, яка має спеціальні дії виклику та отримання, а також інструменти mashup, такі як Yahoo! Pipes. - *Push (Пускання):* Цей стиль дозволяє веб-сервісу зв'язуватися з клієнтом навіть без явних запитів, за умови, що клієнт зареєструвався/підписався на веб-сервіс. Реєстрація необхідна для інформування компонента про кінцеву точку зв'язку клієнта та про події, що цікавлять (наприклад, зміна, видалення). Цей шаблон проектування відомий як *публікація-підписка* . Стратегія push рідко зустрічається в мовах композиції веб-сервісів з API на основі REST, оскільки протокол REST вимагає, щоб клієнт ініціював зв'язок [32]. BPEL підтримує push-взаємодії з дією *отримання*, яка може очікувати (прослуховувати) вхідні повідомлення, та *обробники подій* , які можуть реагувати на загальні події, ініційовані партнерами. - *Бізнес-протоколи:* Окрім шаблонів «push» або «pull», бізнес-протокол визначає обмеження порядку для послідовностей викликів [33]. Тобто, за допомогою протоколу постачальник послуг встановлює правила розмови між сервісом та його клієнтами. Наприклад, зазвичай необхідно додавати товари до кошика для покупок, *перш ніж*

замовлення можна буде оформити або оплатити. Щоб сервіс міг належним чином керувати кількома паралельними розмовами, необхідно реалізувати так звані *правила кореляції повідомлень*, які дозволяють сервісу співвідносити вхідні повідомлення з відповідними екземплярами розмови, на які вони посилаються. Наприклад, у BPEL ці правила називаються *наборами кореляції* та являють собою набір властивостей, які однозначно ідентифікувати розмову. Веб-сервіси, що реалізують бізнес-протокол, відомі як такі, *що зберігають стан* ; в іншому випадку вони відомі як такі, *що не зберігають стан.*

*Вибір.* Це процес пошуку та ідентифікації конкретних веб- сервісів для використання в композиції, враховуючи вимоги до композиції. Вибір компонентів може відбуватися на трьох різних етапах життєвого циклу композиції сервісу: під час проектування, під час розгортання та під час виконання. Якщо сервіси прив'язані до процесу під час проектування, такий підхід називається *статичною композицією* ; якщо вони прив'язані під час розгортання або виконання, такий підхід називається *динамічною композицією.*

— *Час проектування* - це фаза, під час якої розробник створює композитний сервіс. Розробник вибирає сервіси один раз і назавжди, і, якщо композиція не змінюється, вибрані веб-сервіси постійно прив'язуються до композиції сервісу. Представником вибору компонентів під час проектування є SECE, платформа для контекстно-залежної композиції сервісів на основі визначених користувачем правил; у SECE користувач створює композицію, вибираючи дії з попередньо визначеного набору. Дії представляють собою попередньо визначені взаємодії з конкретними веб-сервісами.

— *Час розгортання* - це фаза, протягом якої композиція встановлюється в середовищі виконання для виконання. Коли платформа композиції сервісів дозволяє вибирати атомарні сервіси під час розгортання, зазвичай композиція, розроблена під час проектування, зберігається як шаблон, який можна переналаштовувати щоразу, коли композиція розгортається. Наприклад, це відбувається в середовищі створення сервісів (SCE), яке надає шаблони композиції сервісів, що визначаються як абстрактні описи композицій

багаторазового використання, що містять заповнювачі для сервісів. Під час розгортання генератор коду прив'язує заповнювачі сервісів до конкретних сервісів.

— *Час виконання* - це фаза, протягом якої виконується композиція сервісу. Цей вид вибору, у більшості платформ або інструментів, що реалізують цей тип вибору, базується на алгоритмах, які прив'язують абстрактні сервіси до конкретних на основі атрибутів якості обслуговування (QoS), таких як ціна, тривалість виконання, безпека, доступність та надійність. Наприклад пропонують глобальний підхід до планування вибору сервісів на основі обмежень якості та уподобань. Прототип METEOR-S підтримує всі три типи відбору [34].

### 2.3 Розроблення цільового веб-сервісу

Типи систем та доменів, які обслуговують мови композиції сервісів:

—*Мешапи* - це веб-застосунки, які об’єднують існуючі веб-ресурси, включаючи дані, презентації та функціональність застосунків. Дані та презентації зазвичай надаються у формі стандартних форматів обміну даними, таких як XML та JSON, форматів синдикації, таких як RSS або Atom-канали, або як HTML, ShockWave Flash (SWF) чи інші графічні елементи, такі як віджети. Функціональність застосунків зазвичай забезпечується через відкриті API, натхненні REST, розроблені такими мовами, як Ruby або JavaScript. Найпопулярнішим інструментом для створення мешапів сьогодні є Yahoo! Pipes, який об’єднує джерела даних. IBM InfoSphere MashupHub зосереджується на управлінні корпоративним контентом (ECM) та інтеграції різнорідних джерел даних (наприклад, веб-сервісів, баз даних, локальних файлів). JackBe

Presto - це платформа для створення мэшапів для корпоративних мэшапів з графічним редактором, заснованим на парадигмі підключення. Мова розмітки корпоративних мэшапів (EMML) - це предметно-орієнтований діалект XML для мэшапів.

— *Бізнес-процеси:* Бізнес-процес – це набір дій, функціональних ролей та зв'язків, що описують функцію бізнесу, яка досягає бізнес-мети [35]. Системи управління бізнес-процесами (BPMS) підтримують проектування, виконання та управління бізнес-процесами та являють собою потужний інструмент для інтеграції «бізнес для бізнесу» (B2B). *Фактично* стандартною мовою моделювання процесів є BPMN, версія 2.0 якої також має співвідношення 1:1 до BPEL. Ще однією відповідною мовою бізнес-процесів високого рівня є Yet Another Workflow Language (YAWL). Основоположні зусилля щодо використання веб-сервісів для інтеграції B2B включають eFlow та DySCo [36].

— *Наукові робочі процеси:* Системи управління науковими робочими процесами (SWfMS) спираються на предметно-орієнтовані мови, що дозволяють вченим ефективно визначати, повторно використовувати, виконувати та контролювати експерименти та аналіз даних за допомогою актів композиції та оркестрації. SWfMS здебільшого орієнтовані на дані, що також означає, що більшість SWfMS обирають підхід до композиції, орієнтований на потік даних, з неявним потоком керування. Triana, Taverna та Kepler є репрезентативними SWfMS на основі композиції.

Нотація композиційної мови складається із системи позначок, знаків, піктограм або символів, що представляють сервіси, оператори потоку даних та потоку керування (словник), правил їх поєднання (граматика) та їхнього значення (семантика). Ми визначили три класи нотацій: текстові, візуальні та гібридні (поєднання текстових та візуальних). Текстові та візуальні нотації відрізняються кількістю вимірів, які вони використовують; текстові мови використовують один вимір для вираження складеного сервісу (наприклад, послідовність символів), а візуальні мови використовують більше одного (наприклад, включаючи просторове розміщення та графічні елементи) [37].

*Текстові.* Ми розділили текстові позначення на три категорії.

— *На основі XML:* XML - це мова розмітки, розроблена для представлення структурованих даних у машинному та людиночитаному форматі. Вона широко використовується для обміну даними через Інтернет, і на

ній базується кілька широко відомих стандартизованих мов (наприклад, XSLT та XQuery). XML широко використовується для визначення мов композиції веб-сервісів. Наприклад, DAML-S, WSFL, BPEL та WSCI - усі вони базуються на XML.

— Нотація *на основі коду* передбачає використання комп'ютерних мов, які не базуються на XML, вони можуть бути власними або базуватися на існуючих текстових комп'ютерних мовах. Наприклад, ql.io, інструмент для мэшапу даних, розроблений eBay, дозволяє інтеграцію HTTP API за допомогою предметно-орієнтованої мови на основі SQL та JSON. - *Контрольовані природні мови:* це мови, словниковий запас та граматика яких є підмножинами природної мови, і тому їх легше вивчати та використовувати, ніж мови програмування. «Контроль» полягає у виборі підтримуваного словникового запасу та граматики. Наприклад, CoScripter дозволяє користувачеві описувати веб-процеси за допомогою таких інструкцій, як «перейти до», Aghaee та Pautasso пропонують EnglishMash для розробки мэшапу з попереднім переглядом у реальному часі, а пропонують мову на основі шаблонів.

*Візуальне.* Візуальні мови програмування (VPL) пропонують абстракції, які приховують технологічні деталі за допомогою візуальних символів та графічних позначень [38]. Мета полягає в ефективному представленні інформації та спрощенні розуміння. Наш аналіз визначив чотири типи візуальних представлень, три з яких представляють окремі позначення, а інший включає будь-яку їх комбінацію.

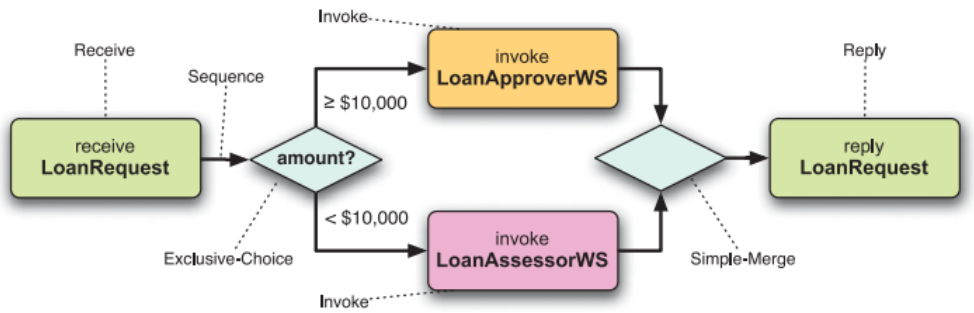


Рисунок 2.10 - Представлення BPEL-процесу з анотаціями потоку керування.

— Підходи *на основі електронних таблиць* зазвичай орієнтовані на кінцевих користувачів-програмістів. Прикладами табличних нотацій у композиції сервісів є Husky [39]; останній пропонує мову, де комірки інкапсулюють основні елементи програмування (наприклад, виклик сервісу), а потік керування виводиться з розташування комірок, де дві події є послідовними, якщо вони розташовані у двох сусідніх комірках (зліва направо).

— Нотації *на основі діаграм* складаються із символів та з'єднувачів, де символи зазвичай є геометричними фігурами, що представляють артефакт композиції, а з'єднувачі - це дроти або стрілки, що представляють потік керування або потік даних, або зв'язок між артефактами. Наприклад, JOpera використовує графи потоків як для специфікації потоку даних, так і для потоку керування (на рисунку 2.10 показано діаграму потоку керування). Інструменти mashup, які застосовують діаграми потоків, - це JackBe Presto та Yahoo! Pipes, але діаграмні представлення діаграм потоків також можна знайти в наукових робочих процесах. Наприклад, Kepler використовує піктограми для представлення дискретних обчислювальних компонентів та стрілки для представлення потоків даних. Інші типи діаграм, що використовуються для композиції сервісів, - це діаграми станів, мережі Петрі та діаграми активності UML.

— *Інші метафори* включають візуальні позначення, які використовують інші представлення реальних об'єктів або ситуацій, легко впізнавані користувачами. Використані метафори включають, наприклад, розкадровки та пазли [40].

*Гібрид.* Гібридні нотації є комбінацією попередніх нотацій. Наприклад, Vegemite – це інструмент машапу, який розширює CoScripter середовищем, подібним до електронної таблиці, під назвою *VegeTable.* Результатом є текстова нотація на основі скриптів та візуальна нотація на основі електронної таблиці.

Парадигма мови – це підхід до програмування, заснований на узгодженому наборі принципів і практик, які визначають придатність мови для вирішення певних типів задач. Запропоновані досі інструменти композиції

сервісів охоплюють різноманітні парадигми, які ми класифікуємо на шість класів.

*На основі скриптів.* Мови скриптів (наприклад, Perl) інтерпретуються (не компілюються) і зазвичай служать чітко визначеним, предметно-орієнтованим цілям. Мови скриптів часто мають низьку складність і, отже, потенційно підходять також для звичайних користувачів (яким, однак, все ще потрібно вивчити мову програмування ). Ця парадигма застосовується до композиції сервісів у різних галузях, наприклад, у сферах мэшапів та робочих процесів , де мови скриптів, такі як Swift, дозволяють специфікувати наукові робочі процеси.

*Засновано на потоці.* Ця парадигма визначає програми як мережі процесів, з'єднаних в орієнтований ациклічний граф. Цей підхід зазвичай використовується в композиції сервісів, оскільки процеси в таких мовах є «чорними скриньками», наприклад, веб-сервіси, які надають функціональність віддаленим клієнтам, не розкриваючи своїх внутрішніх компонентів. Поширеними конструкціями у випадку композиції сервісів є конектори потоку керування та потоку даних. Yahoo! Pipes є прикладом підходу, заснованого на потоку даних, чия мова графічного моделювання має конектори, що визначають передачу даних між процесами. BPEL також базується на потоці, проте з акцентом на потоці керування.

*Функціональні.* Функціональні мови базуються на конструкції математичних функцій. Програми визначаються як функції, оцінка яких представляє вихід програми. Функції не мають побічних ефектів і стану, оскільки результат функції залежить виключно від її вхідних даних, тому ця парадигма вважається чисто орієнтованою на дані. Ця парадигма широко використовується в наукових робочих процесах, де важлива паралелізація обчислень, що забезпечується веб-сервісами, змодельованими як функції без обмежень на впорядкування. Представлення сервісів як функцій дійсне для сервісів, які не мають стану та побічних ефектів. Функціональна мова, наприклад, використовується в інструменті mashup MashMaker. Однак варто зазначити, що функціональне програмування в композиції сервісів зазвичай не повністю

відповідає парадигмі чистого функціонального програмування, оскільки чисте функціональне програмування має деякі практичні обмеження (наприклад, воно не дозволяє операції вводу/виводу).

*Засновані на правилах.* Системи програмування на основі правил складаються з фактів, правил та стратегій керування. Факти та правила – це знання системи. Факти – це інформація (тобто твердження та зв'язки) , а правила – це вирази умов-дій, що перетворюють факти. Стратегії керування вирішують конфлікти, якщо спрацьовують суперечливі правила . Системи на основі правил є дуже модульними: їх можна розбити на частини, вирішити окремо та інтегрувати згодом. Наприклад, використовують бізнес-правила, специфічні для фаз життєвого циклу композиції сервісів (наприклад, правила даних, правила обмежень, правила винятків), для керування процесом композиції сервісів; SWORD, інструментарій розробника для композиції веб-сервісів, представляє сервіси як правила, які перетворюють входи (умови) на виходи (дії) та здатний визначати, чи можна реалізувати бажаний складений сервіс, визначений за допомогою фактів та правил, за допомогою набору заданих сервісів чи ні.

*На основі ECA.* Системи, керовані подіями, використовувалися для підтримки взаємодій у кількох класах слабопов'язаних та динамічних програм. Більш конкретно, правила дій за умовами подій (ECA) використовувалися для кодування логіки композитних сервісів як функції подій під час виконання, а не фактів у базі знань. Правила ECA особливо привабливі для підтримки налаштування композитних сервісів, оскільки правила можна легко додавати, змінювати або видаляти, щоб відображати нові вимоги. Хоча композиції на основі ECA здаються схожими на підходи, засновані на правилах, вони нелегко піддаються систематичному міркуванню, такому як перевірка властивостей, оскільки їм бракує необхідної бази знань.

*Запитання-орієнтовані.* Мови запитів розроблені для обробки даних (тобто отримання, вставки, зміни, видалення) на високому рівні абстракції. Квінтесенцією мови, що базується на запитах, є структурована мова запитів (SQL). Мови запитів класифікуються залежно від артефакту, який вони

запитують; наприклад, SQL та OQL – це мови запитів *до баз даних* , XSLT та XQuery – це мови запитів *XML , а SPARQL – це мова запитів на основі графів*. Існує кілька прикладів мов запитів, що застосовуються до композиції сервісів: XQSE, заснована на XQuery, інтегрує сервіси даних на платформі AquaLogic; XL інтегрує веб-сервіси в XML-документи; та ql.i о.  - це мова програмування на основі SQL для отримання даних із сервісів.

### 2.4 Висновки по розділу

У другому розділі розглянуто методи та інструменти розробки веб-сервісів. Проаналізовано підходи до проєктування застосунків на основі веб-сервісів, а також засоби опису й композиції сервісів для забезпечення їхньої взаємодії. Визначено архітектурні рішення та технологічний стек, необхідний для реалізації цільового веб-сервісу. Результати розділу дали змогу сформувати структуру майбутньої системи та обґрунтувати вибір інструментів розробки, що забезпечують масштабованість, гнучкість і надійність програмного рішення.

# РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ ВЕБ-СЕРВІСУ

## 3.1 Модуль моделювання та проєктування веб-сервісу

Модельно-керована інженерія (MDE) є основною технологією, що використовується в представленому двокомпонентному механізмі для впровадження автоматизації. Більш конкретно, наш механізм MDE відповідає нашій новій 2D MDE-архітектурі, яка проілюстрована на рисунку 3.1 (її повне теоретичне визначення виходить за рамки цієї роботи).

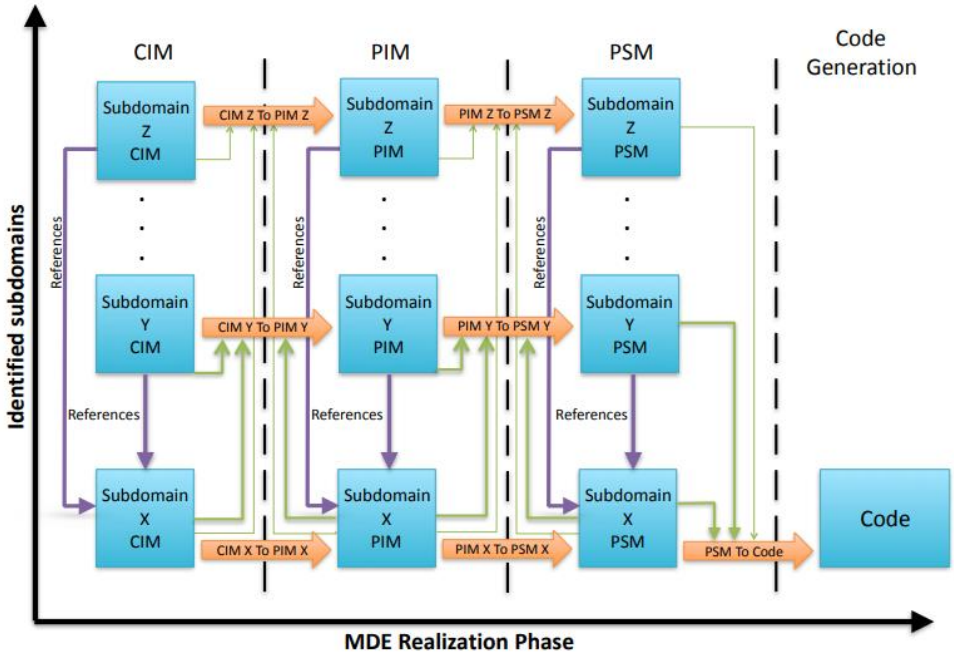


Рисунок 3.1 - Абстрактна архітектура 2D MDE-рушія.

Наш двигун повністю втілює переваги компартменталізації простору задач, що дозволяє розробникам та користувачам MDE працювати з меншими моделями, метамоделями та перетвореннями, тим самим знижуючи сприйняту складність під час побудови MDE-двигунів, а також їх використання. Потреба в міждоменній експертизі також зменшується, тоді як ефективність метамоделювання підвищується, оскільки початкова складна область -

розділяється на простіші конгруентні. Продуктивність додатково підвищується шляхом впровадження паралелізму як на етапі розробки MDE-двигуна, так і на етапі його використання, оскільки незалежні піддомени дають незалежні метамоделі та відповідні ланцюги перетворень.

Підсумовуючи, архітектура 2D MDE розвивається за двома осями. Горизонтальна, вісь фаз реалізації, стосується фаз модельно-керованого проектування, які будуть включені до механізму 2D MDE. З'єднувальними елементами вздовж цієї осі є перетворення моделі до моделі або моделі до тексту. Тип бажаної методології MDE, яка буде використана, визначає кількість фаз реалізації. У цьому випадку використовується MDA, як це було запропоновано групою відкритого управління (OMG), і наразі є найпопулярнішим варіантом MDE.

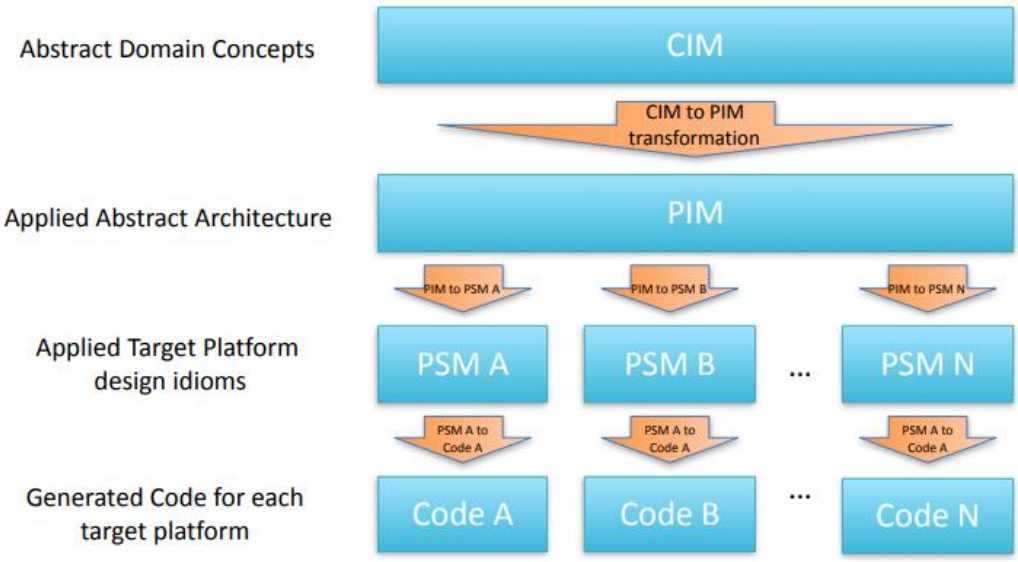


Рисунок 3.2 - Фази модельно-орієнтованої архітектури

Отже, Вісь Фази Реалізації (або скорочено Вісь Реалізації) включає чотири фази MDA (проілюстровані на рисунку 3.2), а саме:

— *Обчислювально-незалежна модель (CIM):* На першому етапі розробник моделює систему, використовуючи абстрактну предметно-орієнтовану мову, яка приховує деталі проектування та реалізації. Результатом цього процесу є CIM-модель задуманої системи, яка містить усі предметні

концепції, що мають бути включені до неї, а також їхні концептуальні взаємозв'язки.

— *Платформно-незалежна модель (PIM):* Після того, як розробник завершує створення CIM-моделі системи, відбувається автоматизований ланцюг перетворень. Спочатку механізм MDA виконує перетворення "Модель-до-Моделі" (M2M) для створення відповідної моделі PIM. Це перетворення CIM-до-PIM уточнює всі елементи моделі CIM, застосовуючи передбачений дизайн та архітектуру системи. Однак це робиться абстрактно, без орієнтації на якусь конкретну платформу реалізації на цьому етапі. Тобто, застосовується архітектурний стиль, такий як *"Клієнт-Сервер",* без будь-яких конкретних ідіом дизайну мови програмування/платформи. Зазвичай між моделями CIM та PIM існує взаємний зв'язок; кожна модель CIM перетворюється на одну модель PIM певної архітектури.

— *Платформно-специфічна модель (PSM):* Після впровадження моделі PIM відбувається друге перетворення M2M – PIM-до-PSM. Цього разу архітектура системи додатково вдосконалюється з урахуванням цільової платформи виконання передбачуваної системи, наприклад, програми Java з базовою базою даних SQL. Наприклад, деякі елементи PIM додатково переглядаються з використанням деяких ідіом цільової мови програмування, доступних бібліотек тощо. На цьому етапі зазвичай існує зв'язок «один-до-багатьох» між одним PIM та кількома можливими PSM цільової платформи.

— *Генерація коду:* Останнім кроком є створення коду запланованої системи. Отже, відбувається перетворення моделі в текст (M2T). Його вхідними даними є модель PSM, а вихідними - згенерований код, де може знадобитися доповнити додатковий код для додаткової/специфічної для конкретного випадку функціональності. Рівень повноти значною мірою залежить від того, наскільки ефективно заплановану систему можна змоделювати за допомогою метамоделі CIM (мови програмування, специфічної для предметної області).

З іншого боку, вертикальна вісь або вісь піддоменів містить метамоделі кожного піддомену, які отримані після застосування компартменталізації

доменів до складного домену. Наприклад, на рисунку 3.1 показано три такі метамоделі X, Y та Z, які разом утворюють початковий складний домен. Уздовж осі піддоменів сполучні елементи є посиланнями від однієї метамоделі до інших. Концепції, що належать до метамоделей Y та Z, посилаються на концепції метамоделі X, тоді як посилань на них або між ними немає. Тобто, метамодель X не залежить від Y та Z, оскільки її визначення не потребує їхніх концепцій, Y та Z залежать від X, тоді як вони незалежні один від одного. Звичайно, може бути багато простіших або складніших сценаріїв. Наприклад, деякі піддомени можуть залежати від кількох інших, посилаючись на кілька інших метамоделей. Концептуальна ясність кожної метамоделі та сукупність зв'язків між метамоделями залежать від успішної компартменталізації доменів.

Домен, у якому наш механізм MDE забезпечує автоматизацію, поділено на чотири піддомени. Перший – це піддомен REST, який забезпечує структурні концепції для продуктивних систем. Другий – це базова автентифікація, яка змінює поведінку базових компонентів REST, на які посилаються, щоб вбудувати можливості автентифікації чи ні. Третій піддомен стосується пошуку за загальними ключовими словами в базі даних, тоді як четвертий автоматизує взаємозв'язок передбачуваних систем з існуючими веб-сервісами в Інтернеті. У наступних підрозділах ці вищезгадані метамоделі представлені більш детально.

*Архітектурний стиль REST*

Оскільки цей двоскладовий механізм спрямований на створення RESTful сервісів, він вбудовує в створені сервіси переважно архітектурний стиль REST. У цьому випадку як модель архітектурного стилю REST використовується модель зрілості Річардсона (RMM). Згідно з цією моделлю, веб-сервіс знаходиться на 3-му рівні RMM (іншими словами, він називається RESTful), тоді і тільки тоді, коли не порушуються наступні правила проектування (проілюстровано на рисунку 3.3).

Правило 1: Структурним блоком веб-сервісу має бути *ресурс* з унікальним URI. Отже, в цьому ресурсоорієнтованому підході кожен сервіс складається з кількох ресурсів, кожен з яких моделює невелику частину

функціональності сервісу.

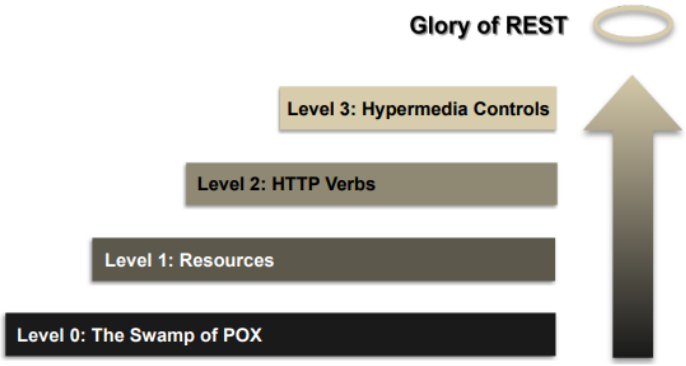


Рисунок 3.3 - Модель зрілості RESTful

— Правило 2: Веб-API кожного ресурсу враховує семантику кожного HTTP-дієслова. Тобто, дієслово *POST* використовується для створення нових ресурсів, *PUT* для оновлення існуючих, *GET* для отримання існуючих та *DELETE* для видалення існуючих ресурсів.

— Правило 3: Ресурси мають бути семантично переплетені один з одним за допомогою *гіпермедійних посилань*. Тобто, щоразу, коли сервер надсилає відповідь своєму клієнту, він вбудовує в цю відповідь деякі URI з можливими подальшими діями для цього клієнта.

Останнє правило RMM, так зване HATEOAS, є особливістю стилю REST, яка найбільше відрізняє його від інших веб-архітектур і значно допомагає уникнути зв'язку між сервісами та їхніми клієнтами. А саме, сервісу не потрібно публікувати попередньо визначений інтерфейс, а радше відповідає на запити клієнтів, які також містять гіпермедійні посилання, посилаючись на ресурси, до яких клієнт має доступ. Отже, дотримуючись моделі RMM REST, сервер може змінювати свій інтерфейс, наприклад, додаючи або видаляючи гіпермедійні посилання до/зі своїх відповідей, проте клієнт все ще зможе знаходити ці доступні посилання, що відповідають його протоколу зв'язку, та покращувати стан програми. Компромісом цього зменшення зв'язку є те, що зв'язок між сервером і клієнтом іноді може бути занадто інтерактивним, проте такі побічні ефекти можна передбачити за допомогою оптимізації на стороні сервера.

### 3.2 Моделювання та реалізація RESTful веб-сервісів

Метамоделі CIM, що складають предметно-орієнтовану мову представленого 2D MDE-рушія, – це чотири метамоделі конкретного призначення. Як уже обговорювалося, наш MDE-рушій моделює RESTful-сервіси, які можуть вбудовувати важливі не-CRUD- функціональні можливості, такі як базова автентифікація, сумісність з існуючими сторонніми сервісами, а також функції пошуку за ключовими словами в базі даних. Ця не-CRUD-функціональність метамоделюєтья у відповідних трьох метамоделях, що посилаються на основну REST-модель, яка забезпечує структуру створених сервісів.

Наш механізм включає цю важливу функціональність, відмінну від CRUD, щоб підвищити продуктивність розробника кількома способами. По-перше, обрана функціональність, відмінна від CRUD, є широко поширеною та дуже поширеною в більшості веб-сервісів, тому автоматизація її впровадження зменшує зусилля розробника. По-друге, це зменшує потребу в міждоменній експертизі, оскільки розробники, які не знають, як реалізувати механізми пошуку або не знайомі з механізмами автентифікації, можуть вбудовувати таку функціональність у свої сервіси з мінімальними знаннями предметної області. Нарешті, оскільки ця важлива частина функціональності автоматизована, вона завжди вбудовує однакові якості коду, а її правильність та валідність не залежать від щоденних звичок програмування розробника. У наступних підрозділах представлено ці метамоделі та пояснено їхні елементи та їхню семантику.

*Мета-модель REST CIM*

Первинна метамодель повинна вбудовувати необхідні концепції для моделювання ресурсів, їхнього спільного веб-API та їх веб-взаємодії. На рисунку 3.4 показано основні елементи частини метамоделі REST CIM, які пояснюються нижче, змодельовані за допомогою метамоделі Ecore.

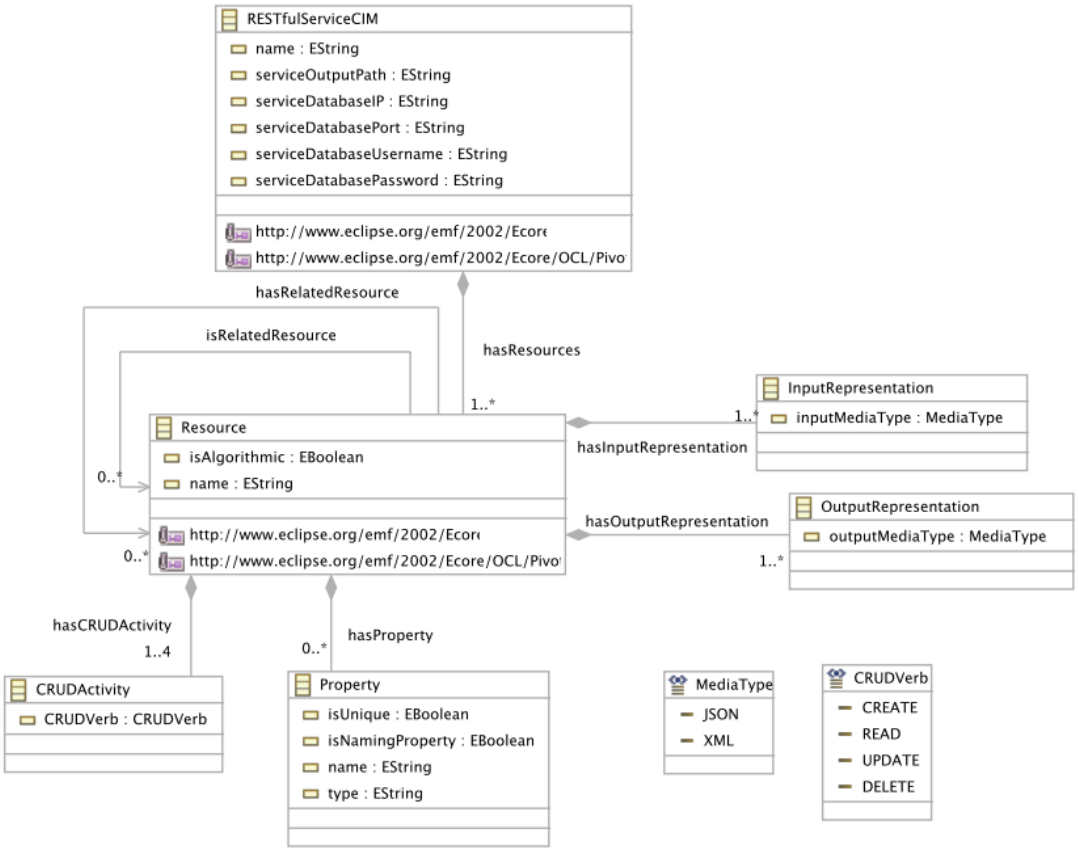


Рисунок 3.4 - Базова метамодель CIM для MDE в S-CASE.

Елемент *RESTfulServiceCIM* є кореневим елементом основної метамоделі. Він фактично вбудовує всю CIM. Він містить назву передбачуваної системи, папку вихідного коду та необхідну інформацію для входу до схеми бази даних, яка буде створена та використана як локальне сховище даних. Тобто, дійсну комбінацію імені користувача/пароля сервера бази даних, який працює на наданій комбінації порту:IP.

Елемент *Resource* моделює ресурс RMM. Очевидно, що це найважливіша концепція для ресурсоорієнтованого моделювання, яка пов'язана з концепціями, що моделюють дані ресурсу, представлення та можливі дії. Він вбудовує назву ресурсу та логічне значення, щоб вказати, чи буде ресурс моделювати дані та їх примітивні операції *CREATE, READ, UPDATE* або *DELETE* , чи вбудовуватиме в RESTful-обгортку якийсь алгоритм, що не є CRUD (наприклад, обчислення найкоротшого шляху, оплата кредитною карткою тощо). Можна помітити відсутність атрибута URI для кожного ресурсу. Це не є необхідним, оскільки

механізм MDE автоматично обчислює URI сервісу, тоді як розробнику просто потрібно вирішити лише, де буде розгорнуто сервіс. Кожен ресурс також може мати деякі елементи *Representation*, які моделюють різні прийнятні медіаформати, доступні у переліку *MediaType*, наприклад, *JSON* або *XML*.

Таблиця 3.1

Ілюстративні гіпермедійні посилання RESTful-сервісу, включені у відповідь на GET-запит списку покупок

| **HTTP-метод** | **Гіпермедійне посилання (Hypermedia Link)** | **Значення властивості найменування** |
|---|---|---|
| GET | [http://www.example.com/list/85/item101](http://www.example.com/list/85/item101) | Чорна футболка (Black T-Shirt) |
| DELETE | [http://www.example.com/list/85/item101](http://www.example.com/list/85/item101) | Чорна футболка (Black T-Shirt) |
| GET | [http://www.example.com/list/85/item102](http://www.example.com/list/85/item102) | Червона футболка (Red T-Shirt) |
| PUT | [http://www.example.com/list/85/item102](http://www.example.com/list/85/item102) | Червона футболка (Red T-Shirt) |

Елемент *Property* представляє атрибут ресурсу. Кожен такий атрибут повинен мати *назву* та *тип* рядка типу, а також логічне значення, яке вказує, чи є властивість однозначною (кратність 1) чи багатозначною. Крім того, кожен ресурс повинен мати рівно одну *властивість найменування.* Це значення властивості найменування включається до списків ресурсів, отриманих клієнтами, щоб мати змогу розрізняти різні ресурси та вибирати ті, які вони хочуть повністю отримати або на які вони хочуть діяти. У таблиці 3.1 показано деякі гіпермедійні посилання списку покупок. Без атрибута властивості найменування у відповідь було б включено лише HTTP-дієслово та посилання. Однак у цьому випадку розробник, який розробив цей ресурс, вибрав його назву як властивість найменування, таким чином значення «Чорна/Червона футболка» також включаються до відповіді сервера та допомагають клієнту вирішити, з яким елементом діяти.

Елемент *CRUDActivity* моделює абстрактну форму кожного дієслова *CRUD* і може приймати будь-яке значення, змодельоване за допомогою

перелічення *CRUDVerb* . Ці елементи формують спільний веб-API будь-якого ресурсу, суворо дотримуючись відповідної семантики стилю REST. Таблиця 3.2 надає інтуїтивно зрозуміле зіставлення між дієсловами CRUD та HTTP.

Таблиця 3.2

Зіставлення дієслів CRUD з HTTP

| Операція CRUD (CRUD Verb) | HTTP-метод (HTTP Verb) |
|---|---|
| CREATE (Створення) | POST |
| READ (Читання) | GET |
| UPDATE (Оновлення) | PUT |
| DELETE (Видалення) | DELETE |

Гіпермедійні посилання, як це диктується 3-м правилом RMM, моделюються за допомогою двох типів асоціацій, які може мати елемент ресурсу. Це *асоціації hasRelatedResource* та *isRelatedResource* . Перша моделює здатність одного ресурсу мати нуль або більше пов'язаних ресурсів, тоді як друга моделює здатність ресурсу бути пов'язаним з нуль або більше іншими ресурсами. На основі цих зв'язків, коли клієнт отримує представлення одного ресурсу, він також отримує список гіпермедійних посилань пов'язаних ресурсів цього ресурсу, що дозволяє пересилати стан програми за допомогою RESTful. У таблиці 3.1 ілюструються такі гіпермедійні посилання, кожне з яких містить необхідний HTTP-дієслово, унікальний URI ресурсу, до якого потрібно дістатися, та його властивість іменування, як уже обговорювалося. У таблиці 3.3 підсумовано основні концепції цієї метамоделі CIM, які моделюють основні концепції дизайну REST.

Таблиця 3.3

Моделювання основних концепцій REST-проектування

| **Концепція REST (REST Concept)** | **Забезпечується елементом(ами) CIM (Satisfied by CIM element(s))** |
|---|---|
| Ресурсно-орієнтований дизайн (Resource Oriented Design) | Ресурс та представлення (Resource and Representations) |
| Загальний веб-API (Common Web API) | CRUDActivity |
| Гіпермедіа (Hypermedia) | hasRelatedResource та isRelatedResource |

Окрім структурних обмежень, вбудованих у представлену метамодель , наш механізм MDE також вбудовує поведінкові обмеження, розроблені з використанням логіки предикатів першого порядку (FOL), щоб мати змогу міркувати про коректність сформованості моделей екземплярів та накладати перевірки валідації на екземпляри під час виконання. Ці поведінкові обмеження були розроблені з використанням еквівалентних формулам FOL , виразів OCL у метамоделях Ecore.

У таблиці 3.4 представлено кілька формул FOL, що використовуються в нашому механізмі MDE, предикати яких визначено в таблиці 3.6. Наприклад, перша з них визначає унікальність кожного типу дії CRUD, яку може мати ресурс. У термінології FOL вона однозначно стверджує, що: *«Для будь-якого x, для будь-якого y, x є ресурсом, а y є дією створення, а y є дією створення ресурсу x, тоді і тільки тоді, коли для будь-якого yi, якщо yi є дією створення, а yi є дією створення ресурсу x, означає, що yi дорівнює y».* Таким чином, нижній РЕГІСТР нашого механізму здатний активно виявляти невідповідності у введених розробником даних та направляти його/її до їх виправлення.

Таблиця 3.4

Ілюстративні формули логіки предикатів першого порядку

| № | Логіка предикатів першого порядку (для редактора формул) |
|---|---|
| 1 | $(\forall x)(\forall y)\big(Rx \ \backslash amp\ Cy \ \backslash amp\ Fyx \ \equiv (\forall y_1)(Cy_1 \backslash amp\ Fy_{1x} \subset y_1 = \ y)\big)$ |
| 2 | $(\forall x)(Rx \ \backslash amp\ (\exists y)(Py \ \backslash amp\ Hyx) \equiv \backslash neg\ Ax)$ |
| 3 | $(\forall x)\Big(Rx \ \backslash amp\ \neg Ax \equiv (\exists y)\big(Py \ \backslash amp\ Hyx \ \backslash amp\ Iy \ \backslash amp\ (\forall y_1)(Py_1 \backslash amp\ Hy_{1x} \backslash amp\ Iy_1 \subset y_1 = \ y)\big)\Big)$ |
| 4 | $(\forall x)(\forall y)(\forall z)\big(Mx \ \backslash amp\ Ry \ \backslash amp\ Kyx \ \backslash amp\ Nz \ \backslash amp\ Lzy \equiv (\forall y_1)(Ry_1 \backslash amp\ Ky_{1x} \backslash amp\ Lzy_1 \subset y_1 = \ y)\big)$ |

З іншого боку, Таблиця 3.5 визначає відповідні вирази OCL, які моделюють цю формулу FOL. Наприклад, обмеження OCL 1 перевіряє, чи кожен ресурс має унікальні типи дій, обмеження 2 перевіряє, чи всі неалгоритмічні - ресурси мають принаймні одну властивість, обмеження 3 гарантує, що кожен

неалгоритмічний ресурс має принаймні одну властивість найменування, і, нарешті, обмеження 4 перевіряє унікальність імен ресурсів, які може містити CIM.

Таблиця 3.5

Ілюстративні обмеження OCL

| **№ (Nr.)** | **OCL-обмеження елемента CIM (OCL constraint of CIM element)** | **Визначення OCL-обмеження (OCL constraint definition)** |
|---|---|---|
| **1** | Resource | self.hasCRUDActivity->isUnique(CRUDVerb) |
| **2** | Resource | (self.isAlgorithmic = false) implies self.hasProperty->notEmpty() |
| **3** | Resource | self.hasProperty->notEmpty() implies self.hasProperty->one(isNamingProperty = true) |
| **4** | RESTfulServiceCIM | self.hasResources->isUnique(name) |

*Базова метамодель автентифікації*

Частина метамоделі базової автентифікації CIM моделює необхідні концепції безпеки, щоб мати змогу автоматично вбудовувати механізм автентифікації в передбачувану систему. Зокрема, ці концепції дозволяють розробнику впроваджувати схему автентифікації типу пароль/ім'я користувача. Оскільки RESTful-сервіси не мають стану (наприклад, немає cookie сеансу), - комбінація пароль/ім'я користувача передається за допомогою HTTP-заголовка авторизації запитів клієнта. Рисунок 3.5 ілюструє метамодель автентифікації. Сірі елементи – це згадані елементи метамоделі REST CIM, які вже представлені на рисунку 3.4, тому вони спрощені на цій діаграмі.

Основна концепція автентифікації - це модель *автентифікації (AuthenticationModel* ). Повний REST- сервіс зі схемою автентифікації повинен мати рівно одну модель автентифікації (AuthenticationModel), яка є спеціалізацією концепції ресурсу (Resource).

Таблиця 3.6

Пояснення символу предиката

| Символ предиката (Predicate Symbol) | Значення (Meaning) |
|---|---|
| **R** | є ресурсом (is Resource) |
| **C** | є операцією створення (is Create Activity) |
| **F** | є операцією створення для (is Create Activity of) |
| **P** | є властивістю (is Property) |
| **H** | є властивістю для (is Property of) |
| **A** | є алгоритмічним ресурсом (is Algorithmic Resource) |
| **I** | є властивістю найменування (is Naming Property) |
| **M** | є RESTful CIM (is RESTful CIM) |
| **K** | є ресурсом для (is Resource of) |
| **N** | є назвою ресурсу (is Resource Name) |

Модель автентифікації визначає, що один з ресурсів сервісу та дві його властивості будуть використовуватися як токени автентифікації. Таким чином, оскільки використовується схема базової автентифікації (Basic Authentication), кожна модель автентифікації (AuthenticationModel) має рівно два *токени автентифікації (AuthenticationTokens),* які є спеціалізацією вищезгаданої концепції властивостей (Property). Крім того, токени автентифікації (AuthenticationTokens) додатково спеціалізуються на тому, щоб бути паролем *(Parol word)* або *іменем користувача (UsernameName).* Таблиця 3.7 ілюструє таку модель автентифікації (AuthenticationModel).

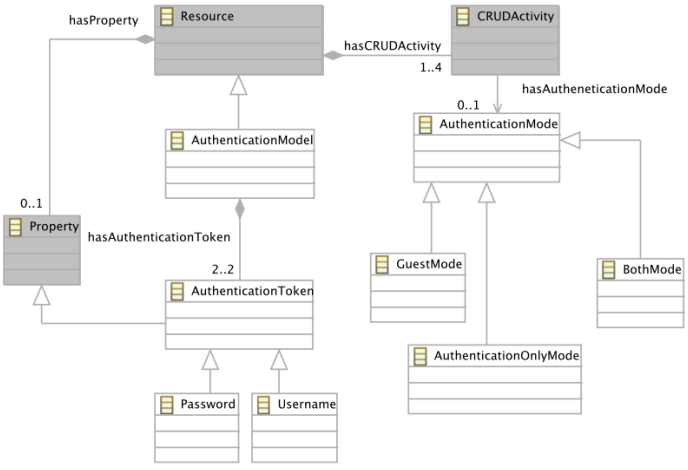


Рисунок 3.5 - Метамодель базової автентифікації CIM.

Основна концепція автентифікації - це модель *автентифікації (AuthenticationModel* ). Повний REST- сервіс зі схемою автентифікації повинен мати рівно одну модель автентифікації (AuthenticationModel), яка є спеціалізацією концепції ресурсу (Resource). Модель автентифікації визначає, що один з ресурсів сервісу та дві його властивості будуть використовуватися як токени автентифікації. Таким чином, оскільки використовується схема базової автентифікації (Basic Authentication), кожна модель автентифікації (AuthenticationModel) має рівно два *токени автентифікації (AuthenticationTokens),* які є спеціалізацією вищезгаданої концепції властивостей (Property). Крім того, токени автентифікації (AuthenticationTokens) додатково спеціалізуються на тому, щоб бути паролем *(Parol word)* або *іменем користувача (UsernameName).* Таблиця 3.7 ілюструє таку модель автентифікації (AuthenticationModel).

Таблиця 3.7

Ілюстративний ресурс, що використовується як модель автентифікації

| **Назва ресурсу (Resource Name)** | **Назва властивості (Property Name)** | **Спеціалізація базової автентифікації (Basic Authentication Specialization)** |
|---|---|---|
| UserAccount | username | Username AuthenticationToken |
| UserAccount | pass | Password AuthenticationToken |

Слід зазначити, що AuthenticationModel може також мати інші властивості , окрім AuthenticationTokens, такі як властивість електронної пошти тощо, як у прикладі Таблиці 3.7.

Оскільки веб-API кожного Ресурсу, як уже обговорювалося, складається з його CRUDActivities, решта концепцій стосуються точного налаштування схеми базової автентифікації, яка використовуватиметься для кожної CRUDActivity. Якщо базова автентифікація застосовується до RESTful сервісу,

то кожна CRUDActivity повинна мати рівно один *AuthenticationMode.* Цей режим точно визначає, чи повинен клієнт бути автентифікованим чи ні, щоб використовувати функціональність базової CRUDActivity. У нашій метамоделі CIM визначено 3 режими. GuestMode *не* виконує жодних перевірок автентифікації та дозволяє будь-якому клієнту робити запити. *AuthenticationOnlyMode* виконує перевірки автентифікації та дозволяє робити запити лише успішно автентифікованим клієнтам. Нарешті, існує змішаний режим ( *BothMode* ), який дозволяє гостьовим клієнтам робити запити, але також виконує перевірки автентифікації, якщо клієнт має встановлений HTTP-заголовок авторизації. У таблиці 3.8 ілюструються деякі CRUDActivities ресурсу Product та їх режим автентифікації.

Таблиця 3.8

Приклад режимів автентифікації для кількох CRUDActivities

| **Назва ресурсу (Resource Name)** | **Операція CRUD (CRUDActivity)** | **Дієслово CRUD (CRUDVerb)** | **Режим автентифікації (Authentication Mode)** |
|---|---|---|---|
| Product | createProduct | CREATE | BothMode |
| Product | updateProduct | UPDATE | AuthenticationOnlyMode |
| Product | readProduct | READ | BothMode |
| Product | deleteProduct | DELETE | AuthenticationOnlyMode |

Як і в основній метамоделі REST, обмеження OCL визначаються для перевірки коректності сформованості екземпляра моделі CIM щодо її концепцій базової автентифікації. Такі обмеження гарантують, що існує лише одна модель автентифікації (AuthenticationModel), яка має рівно два токени автентифікації (AuthenticationTokens), один з яких є ім'ям користувача, а інший - паролем (Password). Інші обмеження перевіряють, чи існує рівно один режим автентифікації (AuthenticationMode) для кожної активності CRUDActivity передбаченої системи тощо.

*Метамодель пошуку в базі даних*

Частина метамоделі пошуку в базі даних CIM моделює необхідні концепції для автоматизації пошуку за широко використовуваними ключовими

словами. Вона містить концепції, які дозволяють розробнику моделювати деякі ресурси, щоб мати змогу шукати властивості інших ресурсів за ключовим словом. Отже, згенерований сервіс автоматично вбудовуватиме необхідну функціональність для обробки запитів клієнта на пошук та надсилання зворотних посилань на будь-які збіги артефактів. Рисунок 3.6 ілюструє ці метамоделі пошуку. Знову ж таки, згадані концепції метамоделі REST CIM, які вже були представлені в попередньому підрозділі, виділені сірим кольором та спрощені.

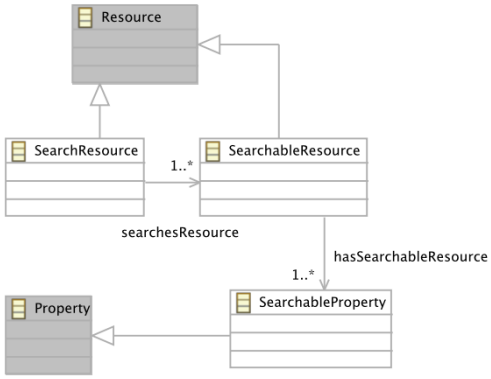


Рисунок 3.6 - Пошук у базі даних у метамоделі CIM.

Найважливішою концепцією цього рівня функціональності, що не є CRUD, є концепція *SearchResource* . Ця концепція є спеціалізацією Resource та моделює алгоритмічні ресурси, що містять алгоритм, що не є CRUD, здатний обробляти вхідні запити пошуку за ключовими словами, робити відповідні запити до бази даних та об'єднувати список посилань для зіставлених артефактів. *SearchableResource* також є спеціалізацією концепції Resource, яка моделює ресурс, властивості якого (деякі або всі) будуть доступні для пошуку деяким SearchResource. Ці властивості моделюються за допомогою концепції *SearchableProperty*, яка є спеціалізацією концепції Property. Визначені обмеження OCL підтверджують, що кожен SearchResource шукає принаймні один SearchableResource, який, у свою чергу, має принаймні одну SearchableProperty.

У таблиці 3.9 показано два об'єкти SearchResources та властивості SearchableProperties об'єктів SearchableResources, які вони шукають. Слід зазначити, що SearchResource може шукати властивості різних SearchableResources, і що SearchableProperty може бути доступною для пошуку за кількома SearchResources.

Таблиця 3.9

Приклад моделювання пошуку в базі даних

| **Назва ресурсу пошуку (SearchResource Name)** | **Назва цільового ресурсу (SearchResource Name / Target Resource)** | **Назва властивості пошуку (SearchProperty Name)** |
|---|---|---|
| ProductSearch | Product | description |
| ProductSearch | Product | name |
| ProductSearch | Tag | tagName |
| TagSearch | Tag | tagName |

*Метамодель стороннього обгортання*

Решта концепцій метамоделі CIM стосуються сумісності з існуючими RESTful сервісами сторонніх розробників. Іншими словами, ця частина метамодель CIM дозволяє розробнику моделювати деякі з його/її передбачуваних системних ресурсів як RESTful клієнтів, кожен з яких надсилатиме запити до одного конкретного стороннього RESTful сервісу. Крім того, можна моделювати вхідні/вихідні дані кожного сервісу та вирішувати, чи слід зберігати його відповіді в локальній базі даних. Рисунок 3.7 ілюструє ці концепції та їх зв'язки. Концепції, на які посилається метамодель REST CIM, виділені сірим кольором та спрощені.

Основною концепцією цієї частини є *RESTClientResource*. Це спеціалізація концепції Resource з відкритим веб-API. Кожен такий RESTClientResource має *TargetRESTService*, який містить URL-адресу стороннього сервісу та очікуваний CRUD-дієслово, яке має використовуватися. Якщо існують параметри запиту, вони моделюються за допомогою концепції *QueryParam* . Концепції *InputDataModel* та *OutputDataModel* моделюють вхідні та вихідні дані цільового RESTful сервісу.

Обидві вони є спеціалізаціями концепції Resource і як такі, вони мають представлення медіаформату та деякі властивості, що моделюють дані вхідної або вихідної моделі. Нарешті, існує три спеціалізації OutputDataModel. Перша, *NonPersistentOutput* , моделює вихідні дані, які не будуть зберігатися в локальній базі даних, а будуть лише надіслані назад клієнту. AutoPersistentOutput використовує метадані OutputDataModel для автоматичного створення повністю постійного ресурсу з веб-API, тоді як третя спеціалізація, *ExistentCRUDPersistentOutput* , використовує існуючий ресурс системи як OutputDataModel.

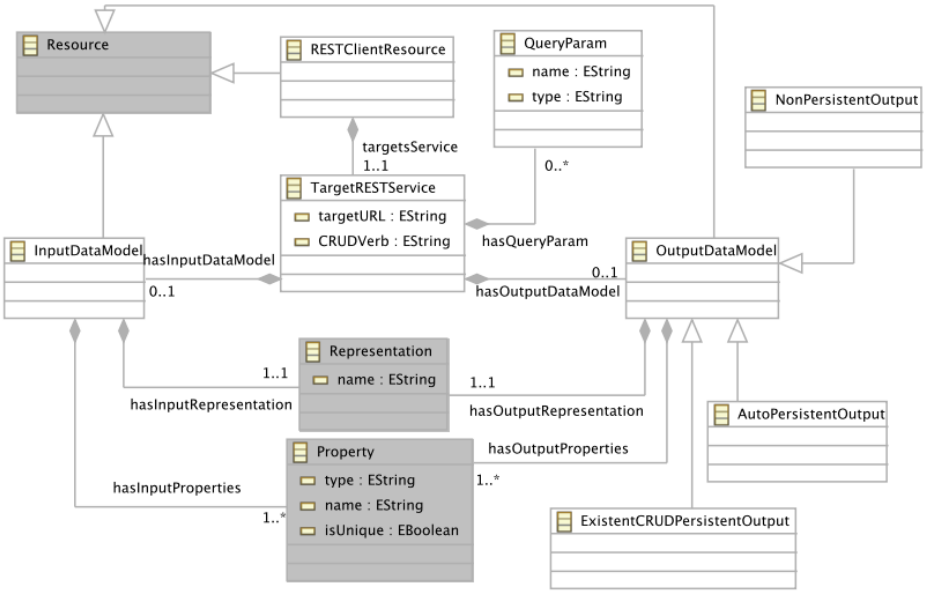


Рисунок 3.7 - Мета-модель CIM-обгортки сторонніх сервісів.

У таблиці 3.10 наведено приклад створеної моделі. Насамкінець, автоматизація, досягнута нашим механізмом, залежить від бажаної функціональності, яку має вбудовувати передбачений сервіс. Очікується, що всі сервіси, що моделюють структуру даних, яка має бути надана через веб-API, включаючи можливості автентифікації, пошук за ключовими словами та/або взаємодію з існуючими сторонніми сервісами, будуть повністю автоматизованими.

У випадках, коли потрібна загальна функціональність веб-сервісу поряд з

деякими користувацькими алгоритмічними функціями, наш механізм створює більшу частину передбаченого сервісу та шаблони RESTful-обгортки для будь-якого ручного коду, який розробник має додати. В крайньому випадку веб-сервісів, які не вбудовують жодних структур даних і містять лише користувацькі функції, які також недоступні як сторонні сервіси, очікується, що наш механізм створить набір шаблонів RESTful, до яких потрібно додати ручний код, щоб бути повністю функціональним. Однак у будь-якому з вищезгаданих випадків результатом є компільований та виконуваний сервіс, готовий до розгортання. Таблиця 3.10 ілюструє ці випадки.

Таблиця 3.10

Приклад моделювання взаємодії зі сторонніми розробниками

| **Концепція метамоделі CIM (CIM Meta-Model Concept)** | **Значення (Value)** |
|---|---|
| RESTClientResource | LocalWeather |
| TargetRESTService - targetURL | `http://www.example.com/LocalWeather/REST` |
| TargetRESTService - CRUDVerb | READ |
| QueryParam | Country |
| QueryParam | City |
| InputDataModel | WeatherDate |
| InputDataModel - Property | Year |
| InputDataModel - Property | Month |
| InputDataModel - Property | Day |
| InputDataModel - Representation | `application/JSON` |
| NonPersistentOutput | Forecast |
| NonPersistentOutput - Property | Temperature |
| NonPersistentOutput - Property | Humidity |

## 3.3 Демонстрація функціональності та оцінка ефективності веб-сервісу

Модулі нашої системи включають два інструменти для введення

мультимодальних вимог: Редактор вимог та Конструктор розкадровки, які також створюють екземпляри онтологій для побудови першого представлення моделі, а також механізм MDE, який застосовує перетворення моделі до моделі та моделі до тексту для остаточного створення виконуваного вихідного коду запланованого сервісу. Всі інструменти, включаючи інструкції з встановлення та використання.

У наступних підрозділах ми оцінюємо ефективність нашої методології для побудови повнофункціональних RESTful веб-сервісів з використанням мультимодальних вхідних даних від вимог до програмного забезпечення. Спочатку ми надаємо огляд послідовності дій, яких повинен дотримуватися розробник для створення веб-сервісу. Потім ми ілюструємо застосування нашої методології на практичному прикладі. Нарешті, ми надаємо деякі показники використання нашого механізму або в наших прикладах, або в дослідженнях користувачів, що охоплюють різні області застосування.

*Послідовність дій*

Спочатку розробник використовує плагіни Requirements Editor та Storyboard Creator для створення функціональних вимог та розкадровок, що описують задуманий сервіс. Ці артефакти синтаксично та семантично анотуються, а анотації згодом уточнюються користувачем. Після цього статична та динамічна онтологія системи автоматично створюються та агрегуються для створення екземпляра REST-сумісної агрегованої онтології. YAML-файл автоматично генерується з цього екземпляра та надається розробнику, щоб він/вона міг його перевірити та внести будь-які уточнення за допомогою майстра плагіна MDE.

Послідовність дій нашої системи показано на рисунку 3.8.

Механізм MDE виконує автоматизовані перетворення M2M з CIM до PIM та з PIM до PSM, і, нарешті, створює вихідний код задуманого сервісу на основі PSM. Згенерований сервіс є повністю функціональним, включаючи базу даних для його ресурсів, а також API для його кінцевих точок. Вихідний код для всіх ресурсів сервісу генерується автоматично, тоді як розробнику може знадобитися

лише доопрацювати код для алгоритмічних ресурсів.

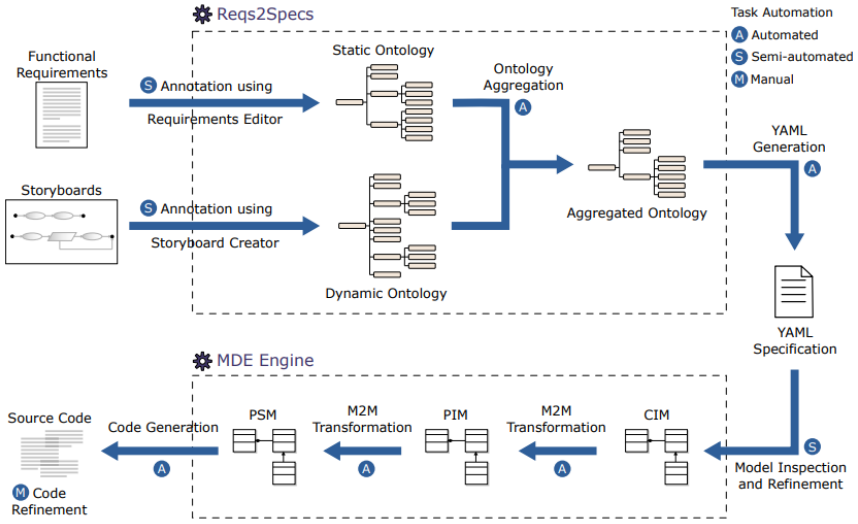


Рисунок 3.8 - Огляд послідовності дій нашої системи

*Тематичне дослідження RESTMarks*

У цьому підрозділі ми наводимо тематичне дослідження для проекту *Restmarks*. Розглянемо Restmarks як сервіс, який дозволяє користувачам зберігати та отримувати свої закладки онлайн, ділитися ними з іншими користувачами та шукати закладки за допомогою тегів. Можна уявити його як соціальний сервіс для закладок. У наступних абзацах ми ілюструємо, як можна використовувати нашу систему для легкого створення такого продукту, водночас гарантуючи, що створений сервіс є повністю функціональним та відстежуваним. Усі елементи, необхідні для відтворення нашого тематичного дослідження, включаючи вимоги, розкадровки, моделі MDE та створений код.

Перший крок процесу включає введення та анотування функціональних вимог. Ілюстративна частина вимог Restmarks, анотована та уточнена за допомогою Редактора вимог, зображена на рисунку 3.9.

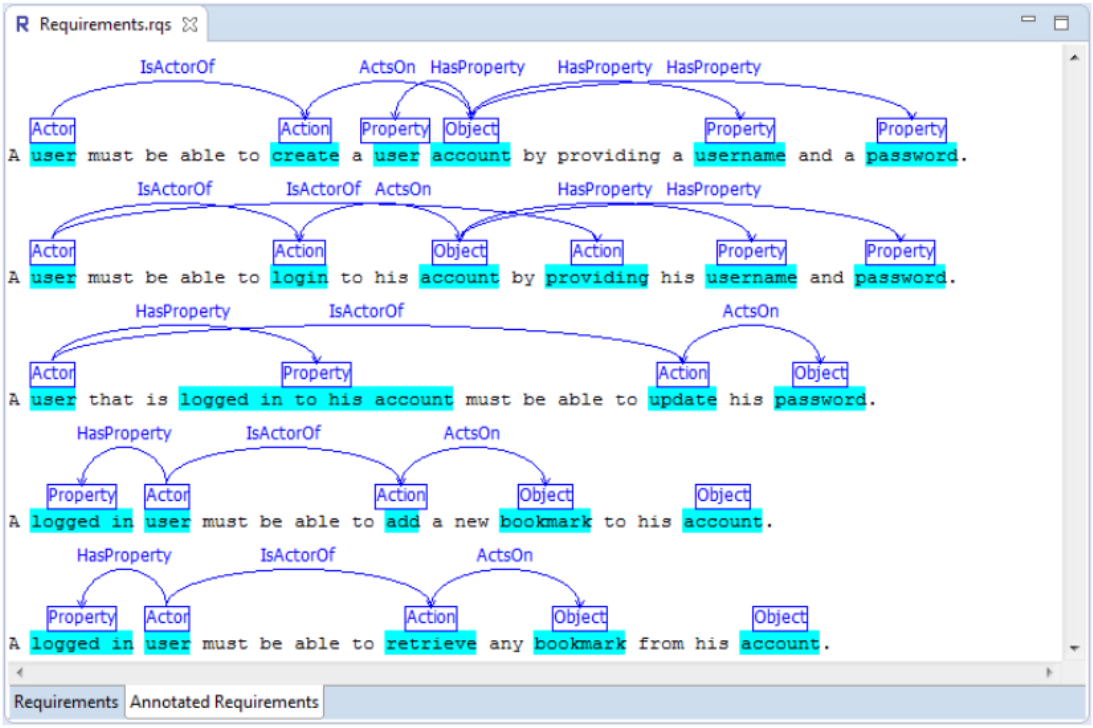


Рисунок 3.9 - Витяг з анотованих функціональних вимог до позначок проєкту

Після додавання вимог користувач також повинен ввести інформацію про динамічне представлення системи. У цьому прикладі динамічне представлення системи надано у вигляді розкадровок. Припустимо, що Restmarks має такі динамічні сценарії:

1. Додати закладку: Користувач додає закладку до своєї колекції та, за бажанням, додає тег до щойно доданої закладки.
2. Створити обліковий запис: Користувач створює новий обліковий запис.
3. Видалити закладку: Користувач видаляє одну зі своїх закладок.
4. Вхід до облікового запису: Користувач входить до свого облікового запису.
5. Пошук закладок за тегом у всій системі: Користувач шукає закладки, вказуючи назву тегу. Пошук охоплює всі загальнодоступні закладки.
6. Пошук закладок за тегом для всього користувача: Користувач шукає закладки, вказуючи назву тегу. Пошук здійснюється як у публічних, так і в приватних закладках користувача.

7. Показати закладку: Система показує користувачеві певну закладку.

8. Оновити закладку: Користувач оновлює інформацію в одній зі своїх закладок.

Розкадровки вищезазначених сценаріїв створені за допомогою Storyboard Creator. Приклад діаграми розкадровки з використанням цього інструменту показано на рисунку 3.10. Розкадровка на цьому рисунку стосується першого сценарію додавання нової закладки. Решту вимог та розкадровок проекту пропущено через обмеження простору.

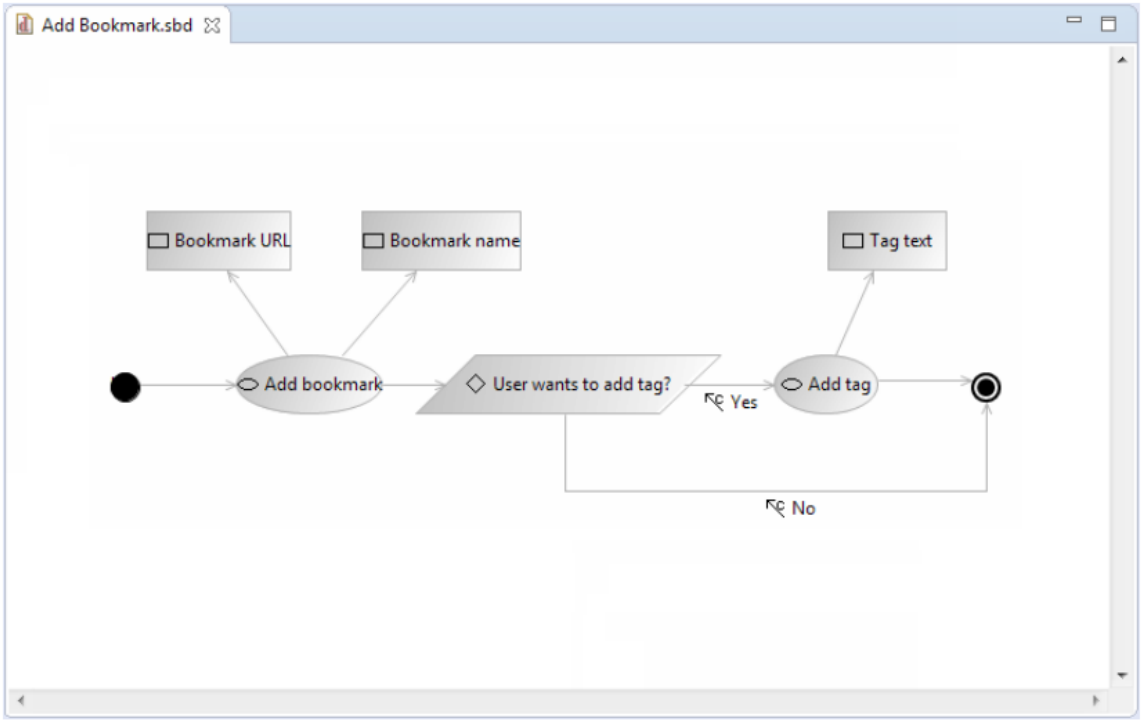


Рисунок 3.10 - Діаграма розкадровки «Додати закладку» для позначок залишку проекту

Після створення анотованих вимог та розкадровок наступним кроком є створення статичної та динамічної онтологій. Дві онтології об'єднуються для створення екземплярів агрегованої онтології.

Ілюстративне створення екземплярів OWL-класів Resource , Property та Activity агрегованої онтології наведено в Таблиці 3.10.

Таблиця 3.11

Створені екземпляри класів Resource, Property та Activity для позначок залишку проекту

| Клас OWL (OWL Class) | Екземпляри OWL (OWL Instances) |
|---|---|
| Resource | bookmark, tag, account, password |
| Property | username, password, private, public, user |
| Activity | search_bookmark, add_tag, add_bookmark, delete_bookmark, update_tag, update_password, login_account, retrieve_bookmark, update_bookmark, get_bookmark, delete_tag, mark_bookmark, create_account |

-

Зрештою, YAML-представлення, яке описує проєкт, експортується з агрегованої онтології. Для Restmarks YAML-файл показано на рисунку 3.11, налаштований за допомогою майстра CIM для створення більш повного CIM.

Після створення YAML-представлення запланованої системи, MDE вживає заходів, надаючи розробнику можливість удосконалити проаналізовані програмні артефакти та/або визначити нову функціональність. Модуль механізму MDE надає набір зрозумілих екранів, кожен з яких змінює концепції CIM (основна частина REST, базова автентифікація тощо). На рисунку 3.10 зображено основний екран метамоделі REST CIM та деякі вдосконалення, які розробник виконав для створення коректної моделі системи, наприклад, додавання нових ресурсів (bookmarkShare), визначення типів даних, форматів медіа-представлення вводу/виводу та зв'язків між ресурсами.

```
- !!cim.Resource
  Name: account
  IsAlgorithmic: false
  CRUDActivities: [Create, Read, Update, Delete]
  Properties: [username, password]
  RelatedResources: [bookmark]
- !!cim.Resource
  Name: tagSearch
  IsAlgorithmic: true
  CRUDActivities: []
  Properties: []
  RelatedResources: []
- !!cim.Resource
  Name: tag
  IsAlgorithmic: false
  CRUDActivities: [Create, Read, Update, Delete]
  Properties: [name, description]
  RelatedResources: [tagSearch]
- !!cim.Resource
  Name: bookmark
  IsAlgorithmic: false
  CRUDActivities: [Create, Read, Update, Delete]
  Properties: [url, scope]
  RelatedResources: [tag]
```

Рисунок 3.11 - Приклад YAML-файлу для позначок залишку проекту

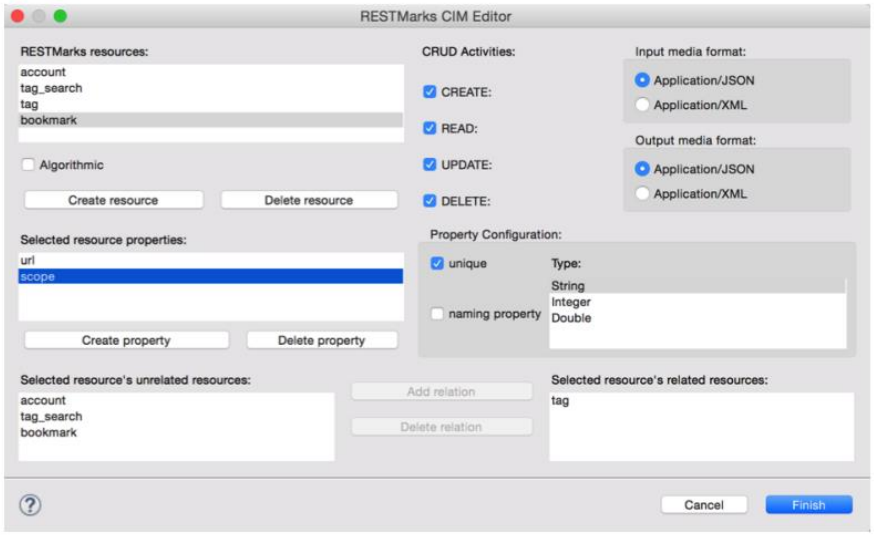


Рисунок 3.12 - Користувацький інтерфейс метамоделі REST CIM.

На рисунках 3.13 та 3.14 представлено інтерфейс користувача автентифікації, за допомогою якого розробник може вибрати модель автентифікації та токени своєї системи, а потім повністю визначити режим автентифікації для CRUDActivities сервісу.

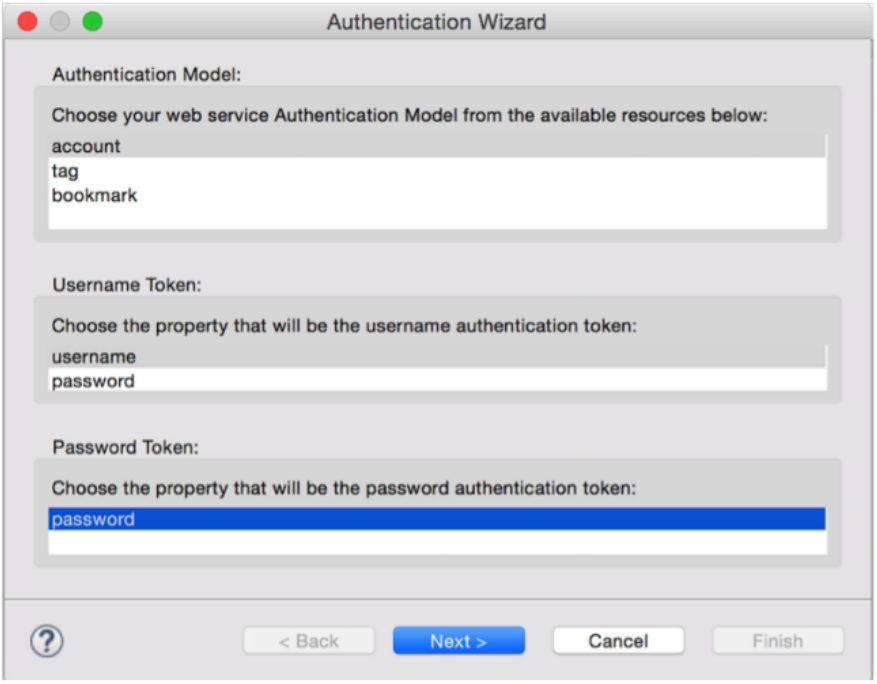


Рисунок 3.13 - Вибір моделі автентифікації за допомогою базового інтерфейсу автентифікації.


Рисунок 3.14 - Визначення бажаної автентифікації кожної CRUDActivity

На рисунку 3.15 зображено інтерфейс користувача для надання можливостей пошуку. У цьому випадку спеціалізував ресурс TagSearch як ресурс пошуку, який здійснює пошук за властивістю *description ресурсу Tag*.

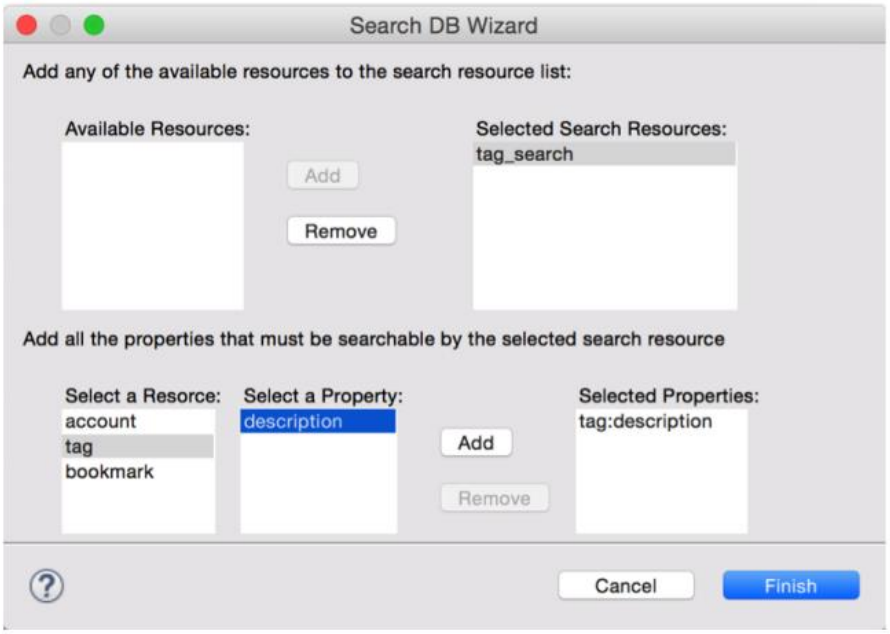


Рисунок 3.15 - Визначення пошукового ресурсу TagSearch.

Нарешті, на рисунку 3.16 зображено інтерфейс користувача для визначення взаємодії зі сторонніми сервісами. У цьому випадку розробник спеціалізував ресурс bookmarkShare для можливості взаємодії з Facebook та обміну закладками.

Після того, як розробник пройде всі вищезгадані екрани, модуль MDE (нижній регістр) створює код сервісів. Для цього ілюстративного прикладу було створено 38 файлів Java, що складають 3843 рядки коду, плюс файл maven pom.xml для побудови сервісу.

Окрім Restmarks, який слугував прикладом дослідження, представленим у попередньому підрозділі, ми додатково оцінюємо нашу методологію та, зокрема, оцінюємо скорочення зусиль за допомогою нашого механізму ще в чотирьох проектах. Одним з них є проект RESTReviews, який ми створили як зразок інтернет-магазину, що підтримує купівлю товарів, написання відгуків тощо. Для трьох інших проектів ми провели дослідження на основі методології, що наведена нижче.

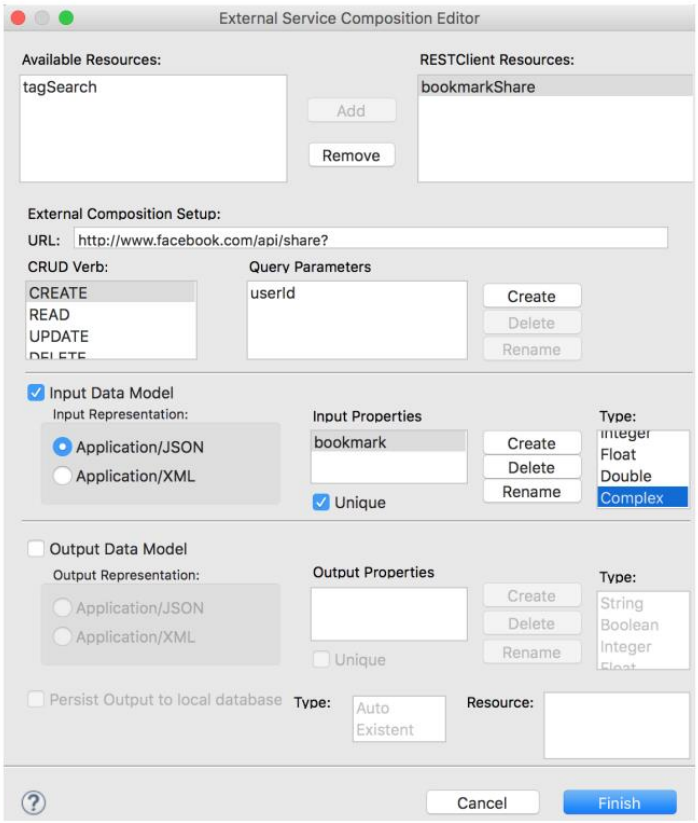


Рисунок 3.16 - Визначення ресурсу bookmarkShare.

*Мета дослідження* Метою нашого дослідження є оцінка того, чи - призводить наш механізм до підвищення продуктивності розробників у командах розробників у корпоративному середовищі.

*Дизайн дослідження* Ми надали наш механізм трьом окремим командам професійних розробників, кожна з яких розробляла та створювала власний застосунок для домену у своєму корпоративному середовищі в контексті їхньої роботи в проекті S-CASE. Ці три дослідження користувачів охоплюють різні сфери застосування: одне з них – це застосунок Інтернету речей (IoT), одне – міні-соціальна мережа, а останнє – застосунок Інтернет як послуга (IaaS). Більше інформації про застосунки можна знайти в Інтернеті.

Спочатку ми навчили кожну команду розробників за допомогою вебінарів (2 години), спеціалізованих тематичних посібників (2 години) та інтерактивних коучингових сесій (2 години ), загалом 6 годин навчання на команду. Потім кожна з команд розробила свій додаток, використовуючи наш механізм (Фаза А), а потім без нього (Фаза Б), використовуючи власну методологію. Нашою метою було виміряти зниження зусиль. Усі команди використовували метод метрики цільових питань, щоб визначити методологію оцінки нашого механізму, а потім повідомити про виміряні покращення процесу розробки програмного забезпечення.

У першому рядку зазначено тип проведеної оцінки, а саме: оцінку як тематичних досліджень авторами або оцінку в дослідженнях користувачів. Другий рядок таблиці містить інформацію про кількість вимог до програмного забезпечення, які розробники мали додати до нашого механізму у вигляді функціональних вимог та/або розкадровок. Третій рядок представляє автоматично створені рядки коду (LoC) мовою Java. Четвертий рядок представляє коефіцієнт *MU*, який означає час, необхідний для створення бажаного сервісу за допомогою нашого механізму, тоді як п'ятий рядок представляє коефіцієнт *MD*, який означає зусилля ручної розробки, необхідні для забезпечення тієї ж функціональності, яка була автоматично згенерована за допомогою нашого механізму, наприклад, створення інтерфейсу сервісу,

налаштування його бази даних, вбудовування можливостей пошуку та побудова взаємодії із зовнішніми сервісами, тестування та налагодження відповідно. Останні два рядки представляють скорочення зусиль. У передостанньому рядку значення скорочення зусиль для наданих тематичних проектів розраховується з урахуванням також кривої навчання нашого механізму. В останньому рядку також наведено зменшення зусиль без кривої навчання, щоб проілюструвати підвищення продуктивності після того, як розробник навчиться використовувати наш механізм.

*Результати дослідження* У таблиці 3.11 підсумовано результати оцінювання двох вибіркових проектів та трьох досліджень користувачів.

Таблиця 3.12

Зменшення зусиль за допомогою представленого механізму

| **Метрика (Metric)** | **RESTMarks** | **RESTReviews** | **IoT app** | **Social Network app** | **IaaS app** |
|---|---|---|---|---|---|
| **Тип оцінювання** *(Assessment Type)* | Автори | Автори | Дослідження користувачів | Дослідження користувачів | Дослідження користувачів |
| **Кількість вимог / розкадровок** *(Number of Requirements/Storyboards)* | 21 | 10 | 13 | 14 | 12 |
| **Згенеровано рядків коду Java** *(Java LoC generated)* | 3843 | 2888 | 7430 | 8115 | 6466 |
| **Коефіцієнт MU, год** *(MU Coefficient (hours))* | Н/Д | Н/Д | 2 | 2.4 | 1.4 |
| **Коефіцієнт MD, год** *(MD Coefficient (hours))* | Н/Д | Н/Д | 10.5 | 10.8 | 9.3 |
| **Зниження трудовитрат, %** *(Effort Reduction (%))* | Н/Д | Н/Д | 23.81% | 22.2% | 20.4% |
| **Зниження трудовитрат без кривої навчання, %** *(Effort Reduction without Learning Curve (%))* | Н/Д | Н/Д | 80.9% | 77.8% | 85% |

*Загрози валідності* Загроза валідності вищезгаданих результатів дослідження полягає в тому, що три команди розробників самостійно повідомили про результати покращення продуктивності. Однак ми вважаємо

загрозу самостійного звітування незначною, оскільки команди розробили та застосували надійну методологію для вимірювання підвищення продуктивності, засновану на методі метрики цільових питань. Інша можлива загроза полягає в тому, що команди використовували свою бажану методологію для ручного раунду побудови системи (Фаза Б). Однак, оскільки кожна команда розробників використовувала технології та інструменти, з якими вони добре знайомі, вони досягли піку своєї продуктивності саме на цьому етапі, тим самим мінімізуючи цю загрозу. Нарешті, порядок експерименту з розробки, спочатку з використанням нашого механізму (Фаза А), а потім без нього (Фаза Б), також не сприяє нашій методології; навпаки, це може сприяти ручному раунду, оскільки командам розробників довелося повторно вирішувати вже знайому їм проблему розробки.

Насамкінець, порівняно з іншими популярними фреймворками, такими як RAML, наш механізм підвищує продуктивність розробників двома способами. По-перше, наш механізм підтримує розробника на типовомуетапі «Вимоги до проектування» в програмній інженерії, оскільки він отримує вхідні дані у вигляді вимог до програмного забезпечення та автоматизує створення моделі проектування. Натомість , в інших популярних фреймворках розробник повинен вручну створювати початкову модель (наприклад, модель RAML), ретельно вивчаючи вимоги до передбачуваної системи , що вимагає значних зусиль. По-друге, результатом таких автоматизаторів Web-API зазвичай є каркас ресурсів, а не повністю розгортаний код, який також може вбудовувати загальну функціональність веб-сервісів, як у випадку з нашим механізмом.

Останнім часом проблема автоматизації процесу розробки програмного забезпечення привернула увагу кількох дослідників. Хоча MDE довів свою ефективність для створення прототипів REST-сумісних веб-сервісів, сучасні інструменти не повністю охоплюють можливості парадигми RESTful. У цій роботі ми розробили методологію, яка дозволяє розробникам моделювати свою систему, використовуючи вимоги до програмного забезпечення з мультимодальних форматів, а також механізм MDE, який застосовує

перетворення моделі до моделі для створення готового до розгортання RESTful веб-сервісу.

Наша система отримує вхідні дані у вигляді функціональних вимог та - потоків дій у вигляді розкадровок, охоплюючи таким чином як структурні, так і поведінкові аспекти запланованої системи. Використання NLP та семантики - полегшує вилучення специфікацій з цих вимог, тоді як розроблені онтології створюють відстежувану модель, яка є високосумісною з парадигмою RESTful. Створена модель (YAML-файл) містить усі необхідні елементи сервісу, включаючи ресурси, дії та властивості CRUD, а також підтримку гіпермедійних посилань.

Крім того, запропонований MDE нижній регістр, на відміну від більшості інших підходів, виходить за рамки REST-моделювання, також вбудовуючи загальні функції, необхідні в багатьох веб-сервісах. Він підтримує функціональність базової автентифікації, популярний пошук за ключовими словами в базах даних та сумісність з існуючими RESTful-сервісами сторонніх розробників. Більше того, він моделює необхідний, компільований та виконуваний код-заповнювач, щоб допомогти розробнику в його ручному програмуванні тих частин його передбачуваної системи, які неможливо автоматизувати.

Підсумовуючи внесок цієї роботи, ми створили набір інструментів для розробників, які дозволяють витягувати специфікації з функціональних вимог та розкадровок і перетворювати ці специфікації у вихідний код запланованої системи. Після оцінки наших інструментів та методології на прикладі проекту Restmarks ми можемо зробити висновок, що наша система забезпечує інтуїтивно зрозуміле відстежуване рішення для напівавтоматичного створення прототипу веб-сервісу з вимог до програмного забезпечення. Подальша оцінка нашої методології в дослідженнях користувачів щодо скорочення часу розробки показує, що наша система може значно полегшити розробку RESTful веб-сервісів.

Подальша робота над нашою методологією може бути спрямована в

кількох напрямках. По-перше, що стосується модуля Reqs2Specs, то в наших найближчих планах - постійне вдосконалення інструментів шляхом отримання відгуків від користувачів. Щодо нижнього регістру MDE, подальший розвиток включає дослідження ручної обробки коду розробником після першого запуску рушія MDE, автоматизацію нефункціональних аспектів, а також автоматизоване створення відповідного веб-клієнта. Нарешті, майбутні дослідження включають подальшу оцінку нашої методології в промислових умовах та оцінку її ефективності як для якості створеного сервісу, так і для скорочення часу розробки.

### 3.4 Висновок по розділу

У третьому розділі виконано моделювання, проєктування та реалізацію веб-сервісу відповідно до визначених вимог. Особливу увагу приділено розробці RESTful веб-сервісу, який забезпечує ефективний обмін даними між компонентами системи. Проведено демонстрацію функціональних можливостей створеного сервісу та оцінено його ефективність за показниками продуктивності, надійності та зручності використання. Отримані результати підтвердили доцільність застосування сучасних підходів інженерії програмного забезпечення для створення якісних і масштабованих веб-сервісів.

# ВИСНОВКИ

У ході виконання бакалаврської роботи було досліджено принципи інженерії програмного забезпечення для веб-сервісів та реалізовано програмне рішення, побудоване на основі сучасних підходів до проєктування й розробки сервісно-орієнтованих систем. Робота охопила аналіз теоретичних основ веб-сервісів, дослідження методів композиції програмних компонентів, проєктування архітектури та реалізацію RESTful веб-сервісу.

На першому етапі було проведено аналіз предметної області, розглянуто основні принципи інженерії програмного забезпечення, особливості композиції програмних компонентів та таксономію веб-сервісів. Це дозволило визначити сучасні підходи до побудови розподілених систем і сформувати основу для подальшого проєктування програмного рішення.

У процесі виконання роботи було досліджено методи та інструменти розробки веб-сервісів, розглянуто особливості проєктування сервісно-орієнтованих застосунків, а також мови опису та композиції сервісів. На основі проведеного аналізу було розроблено структуру цільового програмного рішення та визначено механізми взаємодії між його компонентами.

Практична частина роботи включала моделювання та проєктування веб-сервісу, реалізацію RESTful архітектури та перевірку працездатності створеного механізму. Було продемонстровано основні функціональні можливості системи, оцінено її ефективність та підтверджено відповідність поставленим вимогам.

Результати дослідження показали, що використання принципів інженерії програмного забезпечення та сервісно-орієнтованого підходу дозволяє створювати масштабовані, надійні та гнучкі веб-сервіси, які забезпечують ефективну взаємодію між програмними системами. Запропоноване рішення може бути використане як основа для розробки більш складних інформаційних систем і веб-платформ.

Перспективами подальшого розвитку роботи є впровадження мікросервісної архітектури, розширення функціональних можливостей сервісу,

інтеграція з хмарними платформами та застосування сучасних засобів автоматизованого тестування і розгортання програмного забезпечення.

Отже, поставлену мету роботи досягнуто, а визначені завдання виконано в повному обсязі. Розроблене програмне рішення підтвердило ефективність використання сучасних методів інженерії програмного забезпечення для створення веб-сервісів.

## СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА


1. Erl, T. (2016). Service-Oriented Architecture: Analysis and Design for Services and Microservices. Prentice Hall.
2. Erl, T. (2017). SOA Principles of Service Design. Prentice Hall.
3. Newman, S. (2021). Building Microservices: Designing Fine-Grained Systems (2nd ed.). O'Reilly Media.
4. Richardson, C. (2022). Microservices Patterns (2nd ed.). Manning Publications.
5. Sommerville, I. (2020). Software Engineering (10th ed.). Pearson Education.
6. Pressman, R. S., & Maxim, B. R. (2020). Software Engineering: A Practitioner's Approach (9th ed.). McGraw-Hill.
7. Wazlawick, R. S. (2019). Object-Oriented Analysis and Design for Information Systems. Elsevier.
8. Larman, C. (2021). Applying UML and Patterns (4th ed.). Pearson Education.
9. Hohpe, G., & Woolf, B. (2021). Enterprise Integration Patterns. Addison-Wesley.
10. Alur, D., Crupi, J., & Malks, D. (2019). Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall.
11. Fielding, R. T. (2000). Architectural Styles and the Design of Network-Based Software Architectures. University of California.
12. ISO/IEC 12207:2017. Systems and Software Engineering — Software Life Cycle Processes. International Organization for Standardization.
13. ISO/IEC 25010:2023. Systems and Software Quality Requirements and Evaluation (SQuaRE). International Organization for Standardization.
14. ISO/IEC 29148:2018. Systems and Software Engineering — Life Cycle Processes — Requirements Engineering. International Organization for Standardization.

15. ISO/IEC 27001:2022. Information Security Management Systems — Requirements. International Organization for Standardization.

16. World Wide Web Consortium (W3C). (2025). Web Services Architecture. https://www.w3.org/TR/ws-arch/

17. World Wide Web Consortium (W3C). (2025). SOAP Version 1.2 Specification. https://www.w3.org/TR/soap12-part1/

18. World Wide Web Consortium (W3C). (2025). Web Services Description Language (WSDL). https://www.w3.org/TR/wsdl20/

19. OASIS. (2025). Web Services Business Process Execution Language (WS-BPEL). https://www.oasis-open.org

20. OpenAPI Initiative. (2025). OpenAPI Specification. https://spec.openapis.org

21. Swagger Documentation. (2025). https://swagger.io/docs/

22. Red Hat. (2025). Introduction to Service-Oriented Architecture. https://www.redhat.com/en/topics/integration/what-is-soa

23. IBM Cloud Learn Hub. (2025). REST APIs Explained. https://www.ibm.com/topics/rest-apis

24. IBM Cloud Learn Hub. (2025). Service-Oriented Architecture (SOA). https://www.ibm.com/topics/soa

25. Oracle Documentation. (2025). RESTful Web Services Developer's Guide. https://docs.oracle.com

26. Oracle Documentation. (2025). Java API for RESTful Web Services (JAX-RS). https://docs.oracle.com

27. Apache Software Foundation. (2025). Apache CXF Documentation. https://cxf.apache.org/docs/

28. Apache Software Foundation. (2025). Apache Camel Documentation. https://camel.apache.org/manual/

29. Kubernetes Documentation. (2025). Services, Networking and Service Discovery. https://kubernetes.io/docs/

30. Docker Documentation. (2025). Docker Engine Overview. https://docs.docker.com

31. Nginx Documentation. (2025). Reverse Proxy and Load Balancing. https://nginx.org/en/docs/

32. Jenkins Documentation. (2025). Continuous Integration and Continuous Delivery. https://www.jenkins.io/doc/

33. SonarSource. (2025). SonarQube Documentation. https://docs.sonarsource.com

34. OWASP Foundation. (2025). OWASP API Security Top 10. https://owasp.org/API-Security/

35. Postman Learning Center. (2025). API Testing and Development. https://learning.postman.com

36. IEEE Computer Society. (2024). Service-Oriented Computing and Applications. https://www.computer.org

37. Papazoglou, M. P. (2018). Web Services: Principles and Technology. Pearson Education.

38. Josuttis, N. M. (2017). SOA in Practice: The Art of Distributed System Design. O'Reilly Media.

39. Pautasso, C., Zimmermann, O., & Leymann, F. (2018). RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. Journal of Web Engineering, 17(1), 1–30.

40. Lewis, J., & Fowler, M. (2014). Microservices: A Definition of This New Architectural Term. Martin Fowler. https://martinfowler.com/articles/microservices.html

**БІБЛІОГРАФІЧНА ДОВІДКА**

**Тема бакалаврської роботи:** "Інженерія програмного забезпечення для ВЕБ-сервісів"

Обсяг пояснювальної записки: 77 аркушів

Дата закінчення бакалаврської роботи 10 червня 2026р.

Підпис студента _____