

***БАКАЛАВРСКА
РОБОТА***

БР.ІІ – 00.00.00.000 ІІЗ

Група ІІ-22-4

Кантор Арсен

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Кантор Арсен Олегович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

БАКАЛАВРСЬКА РОБОТА

Архітектурні підходи до побудови модульних програмних систем

(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 А.О. Кантор
(підпис, ініціали та прізвище здобувача)

Науковий керівник Корнута Володимир Андрійович, к.т.н., доцент
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-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

Кантор Арсену Олеговичу

(прізвище, ім'я, по-батькові)

1. Тема проекту (роботи) "Архітектурні підходи до побудови модульних програмних систем"

керівник проекту (роботи) Корнута Володимир Андрійович, к.т.н., доцент

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 Багаторівнева архітектура (рис. 1.1, ст. 10)

2 Модульна когезія (рис. 1.7, ст. 21)

3 Зв'язки між бетонними елементами, їх типами та графами версій. (рис. 2.7, ст. 43)

4 Знімок екрана ArchEdit (рис. 3.4, ст. 52)

5 Знімок екрана управління. (рис. 3.7, ст. 58)

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 обґрунтування 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	02.06.2026	виконано

Студент _____ Кантор А.О.
(підпис)

Керівник роботи _____ Корнута В.А.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 82 сторінок, 26 рисунків, 4 таблиці, список використаних джерел із 40 найменувань.

Метою роботи є дослідження архітектурних підходів до побудови модульних програмних систем та аналіз методів забезпечення масштабованості, гнучкості й підтримованості програмного забезпечення.

Об'єкт дослідження: модульні програмні системи.

Предмет дослідження: архітектурні підходи, методи та інструменти побудови модульних програмних систем, зокрема принципи модульного проєктування, оцінювання якості архітектури та технології підтримки рішення.

Результати дослідження: проведено аналіз сучасних архітектурних підходів до побудови модульних систем, досліджено принципи модульної архітектури та сучасні інструменти підтримки модульних програмних систем.

У першому розділі розглянуто теоретичні основи модульної архітектури програмного забезпечення, переваги та недоліки модульного проєктування, а також метрики оцінювання якості модульних систем.

У другому розділі досліджено методи та інструменти проєктування модульних програмних систем, архітектурні підходи до побудови модульної архітектури та моделювання еволюції програмних систем і лінійок продуктів.

У третьому розділі розглянуто реалізацію модульних програмних систем, інструментальні засоби підтримки модульної архітектури, процеси впровадження та аналіз ефективності використання модульного підходу.

Висновок: використання модульної архітектури дозволяє підвищити гнучкість, масштабованість і підтримованість програмного забезпечення, спростити процеси модернізації систем.

КЛЮЧОВІ СЛОВА: МОДУЛЬНА АРХІТЕКТУРА, ПРОГРАМНІ СИСТЕМИ, SOFTWARE ARCHITECTURE, MODULAR DESIGN, MICROSERVICES, COMPONENT-BASED DEVELOPMENT, SOFTWARE QUALITY, SCALABILITY, MAINTAINABILITY, SOFTWARE ENGINEERING.

ANNOTATION

The bachelor's thesis contains of 82 pages, 26 figures, 4 tables, and a reference list containing 40 sources.

The aim of the work is to study architectural approaches to building modular software systems and to analyze methods for ensuring software scalability, flexibility, and maintainability.

Research object: modular software systems.

Research subject: architectural approaches, methods, and tools for building modular software systems, including principles of modular design, architecture quality assessment, and technologies supporting software modularity.

Research results: modern architectural approaches to building modular systems were analyzed, the principles of modular architecture were studied and modern tools for supporting modular software systems were reviewed.

The first section describes the theoretical foundations of modular software architecture, the advantages and disadvantages of modular design, and metrics for evaluating the quality of modular systems.

The second section analyzes methods and tools for designing modular software systems, architectural approaches to building modular architectures, and modeling the evolution of software systems and product lines.

The third section covers the implementation of modular software systems, tools for supporting modular architecture, deployment processes, and the analysis of the effectiveness of the modular approach.

Conclusion: the use of modular architecture improves software flexibility, scalability, and maintainability, simplifies system modernization processes, and ensures efficient reuse of software components.

KEY WORDS: MODULAR ARCHITECTURE, SOFTWARE SYSTEMS, SOFTWARE ARCHITECTURE, MODULAR DESIGN, MICROSERVICES, COMPONENT-BASED DEVELOPMENT, SOFTWARE QUALITY, SCALABILITY, MAINTAINABILITY, SOFTWARE ENGINEERING.

ЗМІСТ

ВСТУП 7

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ МОДУЛЬНИХ ПРОГРАМНИХ СИСТЕМ 9

1.1 Основні принципи модульної архітектури програмного забезпечення 9

1.2 Переваги та недоліки модульного проєктування програмних систем 16

1.3 Метрики оцінювання якості та ефективності модульних систем 20

1.4 Висновки по розділу 25

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ ПРОЄКТУВАННЯ МОДУЛЬНИХ ПРОГРАМНИХ СИСТЕМ 26

2.1 Архітектурні підходи та передумови створення модульних систем 26

2.2 Методи та підходи до побудови модульної архітектури 30

2.3 Моделювання еволюції архітектури та лінійок програмних продуктів 41

2.4 Висновки по розділу 45

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ МОДУЛЬНИХ ПРОГРАМНИХ СИСТЕМ 46

3.1 Інструментальні засоби підтримки модульної архітектури 46

3.2 Реалізація та впровадження модульної програмної системи 60

3.3 Аналіз ефективності та результатів використання модульного підходу 67

3.4 Висновки по розділу 80

ВИСНОВКИ 81

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА 83

					БР.ІП – 00.00.00.000 ПЗ			
<i>Змн.</i>	<i>Арк.</i>	<i>№ докум.</i>	<i>Підпис</i>	<i>Дата</i>				
<i>Розроб.</i>		Кантор А.О.			Архітектурні підходи до побудови модульних програмних систем Пояснювальна записка	<i>Літ.</i>	<i>Арк.</i>	<i>Акрушів</i>
<i>Перевір.</i>		Корнута В.А.					7	
<i>Реценз.</i>		Романишин Т.Л.				ІФНТУНГ ІІ-22-4		
<i>Н. Контр.</i>		Піх М.М.						
<i>Затверд.</i>		Бандура В. В.						

ВСТУП

У сучасних умовах стрімкого розвитку інформаційних технологій програмне забезпечення стає дедалі складнішим, а вимоги до його масштабованості, гнучкості та підтримуваності постійно зростають. Сучасні програмні системи повинні забезпечувати високу продуктивність, можливість швидкого оновлення функціональності, адаптацію до змін бізнес-вимог та інтеграцію з великою кількістю зовнішніх сервісів. У зв'язку з цим особливого значення набувають архітектурні підходи до побудови модульних програмних систем, які дозволяють розділяти складні програмні рішення на незалежні компоненти та забезпечують ефективне управління їх взаємодією.

Модульний підхід є одним із ключових принципів сучасної програмної інженерії, оскільки він забезпечує спрощення процесів розробки, тестування, супроводу та масштабування програмного забезпечення. Використання модульної архітектури дозволяє зменшити складність системи, забезпечити повторне використання програмних компонентів та підвищити якість програмного продукту.

Актуальність теми зумовлена необхідністю створення гнучких та масштабованих програмних систем, здатних швидко адаптуватися до змін функціональних вимог і технологічного середовища. Традиційні монолітні архітектури часто супроводжуються складністю підтримки, низькою гнучкістю та високими витратами на модернізацію. У той же час модульна архітектура забезпечує можливість поступового розвитку системи, незалежного оновлення окремих компонентів та спрощення інтеграції нових функціональних можливостей.

Метою роботи є дослідження архітектурних підходів до побудови модульних програмних систем, аналіз сучасних методів модульного проектування та оцінювання ефективності використання модульної архітектури у процесі розробки програмного забезпечення.

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						7
Змн.	Арк.	№ докум.	Підпис	Дата		

Завданнями дослідження є аналіз предметної області та сучасних підходів до побудови модульних програмних систем, дослідження принципів модульної архітектури, визначення переваг і недоліків модульного проектування, аналіз метрик оцінювання якості модульних систем, дослідження архітектурних підходів та інструментів підтримки модульності, моделювання еволюції архітектури програмних систем, реалізація модульної програмної системи та оцінка ефективності використання модульного підходу.

Об'єктом дослідження є процеси проектування та реалізації модульних програмних систем у сучасному середовищі розробки програмного забезпечення.

Предметом дослідження є архітектурні підходи, методи, моделі та інструменти побудови модульних програмних систем, зокрема принципи декомпозиції програмного забезпечення, компонентно-орієнтованого проектування та забезпечення взаємодії між модулями.

Методи дослідження включають аналіз наукових джерел для вивчення теоретичних основ модульної архітектури, порівняльний аналіз сучасних архітектурних підходів, моделювання структури програмної системи, експериментальний метод для реалізації та тестування модульних компонентів, а також аналіз ефективності використання модульного підходу в процесі розробки програмного забезпечення.

У роботі передбачається дослідження сучасних методів побудови модульних програмних систем, аналіз інструментів підтримки модульної архітектури та реалізація архітектурного підходу до створення масштабованої програмної системи. Особлива увага приділятиметься забезпеченню гнучкості, підтримуваності та повторного використання програмних компонентів. Очікується, що використання модульної архітектури дозволить підвищити якість програмного забезпечення, спростити процес модернізації систем та скоротити витрати на їх підтримку і розвиток.

Бакалаврська робота містить 82 сторінок, 26 рисунків, 4 таблиці, 3 розділи, список використаних джерел із 40 найменувань.

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						8
Змн.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ПОБУДОВИ МОДУЛЬНИХ ПРОГРАМНИХ СИСТЕМ

1.1 Основні принципи модульної архітектури програмного забезпечення

Сьогодні ми живемо у світі, що характеризується високим рівнем комп'ютеризації. Комп'ютери та пов'язані з ними технології контролюють більшість аспектів сучасного життя. Одним з основних компонентів цих комп'ютерів є програмне забезпечення, яке стосується інструкцій, що вказують комп'ютеру, що робити. Програмне забезпечення включає весь набір програм, процедур та підпрограм, пов'язаних з роботою комп'ютерної системи. Цей термін використовується для відрізнєння цих інструкцій від фізичних компонентів комп'ютерної системи – апаратного забезпечення [1].

З моменту появи цифрових комп'ютерів у 1950-х роках, написання програмного забезпечення еволюціонувало від використання машинної мови через низькорівневі мови асемблера до сучасних мов високого рівня. Хоча мови високого рівня дозволяють писати складні комп'ютерні програми, вони також ускладнюють їх проектування. Зі збільшенням розміру та складності програмних систем проблема проектування вийшла за рамки алгоритмів та структур даних обчислень: проектування та визначення загальної структури системи виникло як новий вид проблеми.

Структурні питання включають загальну організацію та структуру управління; протоколи для зв'язку, синхронізації та доступу до даних; призначення функціональності елементам дизайну; фізичний розподіл; склад елементів дизайну; масштабування та продуктивність; а також вибір серед альтернатив дизайну [2] - це і є архітектура. Таким чином, архітектуру програмного забезпечення можна розуміти як високорівневу структуру програмної системи. Як таку, архітектуру програмного забезпечення можна розглядати як таку, що складається з трьох основних компонентів: структури

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						9
Змн.	Арк.	№ докум.	Підпис	Дата		

системи, процесу створення такої структури та документації структури.

Деякі поширені стилі для представлення архітектури програмного забезпечення - це "канальний та фільтрувальний", "багатошаровий", "репозиторіальний", "сервісно-орієнтована архітектура" (SOA), "розподілена" та "модульна". Для представлення складної взаємодії компонентів необхідно прийняти певний стиль для процесу. У цій роботі ми зосереджуємося на модульній архітектурі проектування програмного забезпечення. Щоб зрозуміти модульний дизайн, ми спочатку розглянемо, як багатошаровий та SOA - два найпопулярніші стилі - визначають структури архітектури програмного забезпечення.

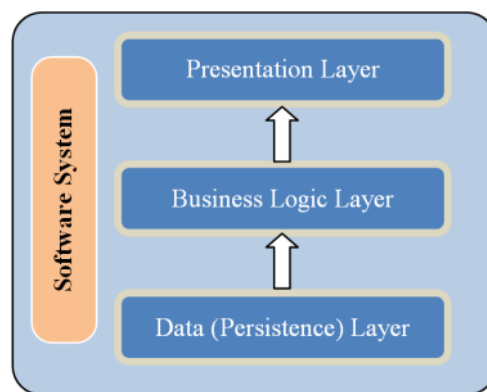


Рисунок 1.1 - Багаторівнева архітектура

Багатошарова архітектура

Багатошарова архітектура - популярний стиль архітектури програмного забезпечення, що фокусується на групуванні пов'язаних функцій у програмному застосунку на окремі шари, розташовані один на одному. Кожен шар забезпечує функціональність, згруповану за спільною відповідальністю або роллю, з явними та слабо пов'язаними взаємодіями між шарами [3]. Таким чином, цей стиль допомагає підтримувати чіткий розподіл відповідальностей, що, у свою чергу, підтримує гнучкість та зручність обслуговування програмної системи. Аналогічно, Ренгайя зазначає, що багатошарова архітектура розподіляє ролі та відповідальність навколо ширшої технічної функції та зображує перевернуту

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						10
Змн.	Арк.	№ докум.	Підпис	Дата		

піраміду, де попередній шар має доступ до більш сфокусованих шарів нижчого рівня [4]. Таким чином, багатошаровий стиль підкреслює фізичне та часто логічне розташування програмного застосунку. (рис. 1.1) Ця багаторівнева структура простежується, наприклад, в архітектурі програмного забезпечення прискорювачів IBM, як показано на рис. 1.2.

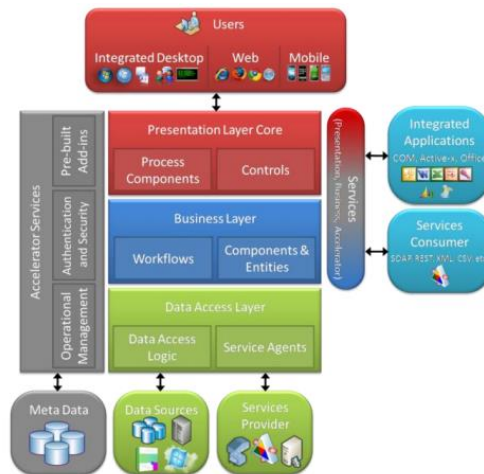


Рисунок 1.2 - Багаторівнева архітектура програмного забезпечення IBM Accelerator

Сервісно-орієнтована архітектура

У сучасному складному та технологічно пов'язаному світі ключовим компонентом проектування програмного артефакту є його взаємодія з іншими артефактами. Міжартефактна взаємодія дозволяє організації швидко перебудовувати та адаптувати свої бізнес-процеси у відповідь як на внутрішні, так і на зовнішні зміни [5]. Стиль сервісно-орієнтованої архітектури (SOA) стосується того, як різні бізнес-функції системи взаємодіють одна з одною. Ці бізнес-функції визначаються як набір послуг.

По суті, SOA передбачає наявність набору сервісів, які можуть виконувати певні бізнес-функції, а ваші клієнти можуть використовувати ці сервіси для виконання своєї роботи [6]. Зазначають, що SOA кодифікує, як ми можемо публікувати, використовувати та ідентифікувати сервіси в різних технічних та функціональних межах. Таким чином, архітектурний стиль SOA

значною мірою зосереджений на аспектах взаємодії програмного додатка.

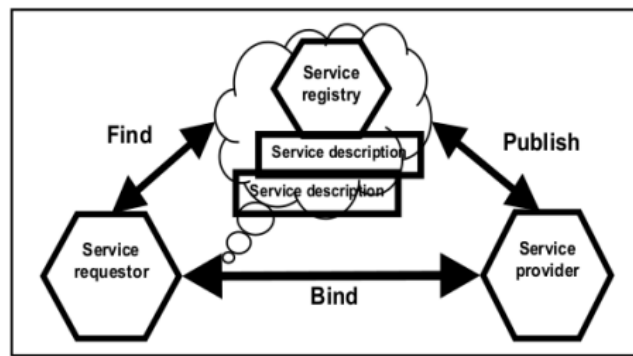


Рисунок 1.3 - Сервісно-орієнтована архітектура

Хоча багаторівнева архітектура та SOA є складними стилями для програмних систем, вони стикаються з певними недоліками, особливо у світі програмного забезпечення, що постійно розвивається. Багаторівнева система в основному розглядається як три рівні – рівень доступу до даних, бізнес-рівень, рівень представлення – які працюють разом, щоб з'ясувати взаємозв'язок між різними елементами програмної системи. Однак у багатьох сучасних програмних проектах рівні самі по собі стали дуже великими, оскільки вони містять кілька компонентів, які залежать один від одного. Іноді ця матриця залежностей настільки складна, що вона природно вимагає розділення рівнів на більш детальні підрівні [7]. Це ставить питання про те, скільки рівнів може мати програмне забезпечення та як керувати складними взаємозв'язками між різними компонентами. Стиль SOA має аналогічні притаманні недоліки щодо того, за яку функціональність буде відповідати сервіс.

Для роботи багаторівневого або SOA-стилю певні компоненти повинні знати та розуміти інші компоненти на цьому рівні або сервісі. Крім того, вони матимуть пов'язані залежності, які можуть порушити роботу всієї програмної програми у разі збою одного компонента. З появою проектів вільного програмного забезпечення з відкритим вихідним кодом (FOSS), що розробляються кількома розробниками з усього світу, як ми можемо змусити всіх цих розробників працювати на одному рівні або сервісі [8]. Багаторівневий та

SOA-стиль структурують монолітну програмну систему, де «єдина» програма пропонує всі варіанти використання та сервіси. Ці архітектурні підходи не тільки негнучкі, але й обмежують навички. Новіші архітектори, дизайнери та розробники програмного забезпечення з різними способами роботи не можуть реалізувати свої навички, не декомпозуючи всю систему. Такий підхід обмежує ступінь, до якої програмна система може адаптуватися та використовувати нові технології та інструменти.

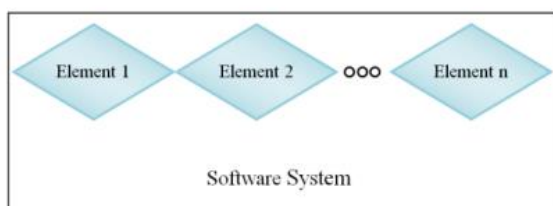


Рисунок 1.4 - Елементи SOA

Модульна архітектура

Модульний стиль архітектури допомагає нам розглядати систему не просто як ієрархію шарів або з точки зору наданих послуг, а як рівень глибини, як композицію менших «модулів» [4]. Кірк визначає модуль як «розгортану, керовану, власно повторно використовувану, компоновану, бездержавну одиницю програмного забезпечення, яка забезпечує стислий інтерфейс для споживачів» [9].

Таким чином, модульна архітектура розкладає програмне забезпечення на менші програми (модулі) зі стандартизованими інтерфейсами, що дозволяє здійснювати зв'язок між модулями та основною системою.

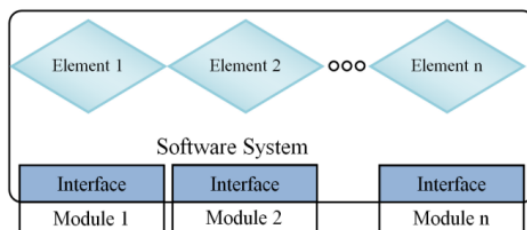


Рисунок 1.5 - Модульна архітектура

Зв'язок між модулями може бути реалізований як потік вводу/виводу, буфери вводу/виводу, конвеєрні або інші типи з'єднань. Кожен модуль системи повинен мати одну конкретну відповідальність, що допомагає користувачеві чітко зрозуміти систему. Це також має допомогти інтегрувати модуль з іншими компонентами [10].

Модуль має чіткий бізнес-контекст, обмежений фізичним рівнем, що його оточує, працює в наданому контексті та виражає свою область дії через публічний інтерфейс [4]. Отже, модуль допомагає нам розуміти, розширювати та керувати системою як під час проектування, так і під час виконання, тобто модульність під час проектування та модульність під час виконання.

Наприклад, модульна системна архітектура для безпілотних літальних апаратів (БПЛА) та безпілотних наземних транспортних засобів (НПАТ), запропонована Джакумідісом, структурує окремі модулі для різних функцій БПЛА/НПАТ, а також модулі планування маршруту [11]. Модульність цієї архітектури робить досить складну систему можливою як для розробки, так і для розгортання. Крім того, вона дозволяє реалізовувати функціональність БПЛА без потреби в НПАТ і навпаки.

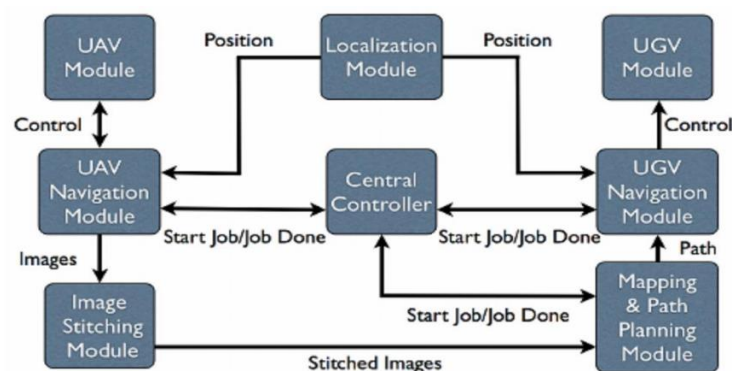


Рисунок 1.6 - Архітектура модульної системи

Концепції модульної архітектури

Модульна архітектура в науковій практиці сприймається як виробнича парадигма для проектування та розробки складних артефактів [12]. Тому вона є критичним елементом у визначенні проектування та розробки складних

програмних проектів, оскільки забезпечує всебічне визначення програмного проекту. Таким чином, для досягнення модульності архітектору необхідно враховувати певні концепції:

А. Взаємодія модуля з додатком

Кожен модуль (артефакт) повинен обмінюватися даними та ресурсами з основною системою та іншими модулями. Таким чином, розробник повинен забезпечити інтерфейси/точки входу для цього зв'язку. Інтерфейси також визначатимуть контроль над тим, як/що/хто/коли відбувається обмін [13].

В. Реєстрація модуля

У модульній конструкції необхідно створити механізм, за допомогою якого система буде знати про існування певного модуля. Існує два підходи до цього:

а) Відкриття

У цьому підході головна програма сканує наявність модуля та, після виявлення, реєструє його. Потім головна програма веде реєстр усіх виявлених модулів та їх статусів. Багато сучасних фреймворків для розробки програмного забезпечення широко застосовують концепцію виявлення. Утиліта PHP composer, яка використовується в Symfony, Laravel та кількох інших PHP-фреймворках, автоматично виявляє модулі та реєструє їх у ядрі фреймворку.

б) Конфігурація

Розробник створює спеціальні параметри конфігурації, щоб забезпечити реєстрацію модуля. Цей параметр конфігурації розгортає поведінку модуля за замовчуванням, дозволяючи йому вивчати інші модулі та існуючі інтерфейси/точки входу.

с) Структура модуля (розділення)

Модуль повинен мати структуру для взаємодії з додатком та іншими модулями. Ця структура повинна визначати оптимальну та практичну логіку призначення [15].

д) Події

Як і в подієво-керованій архітектурі, модулю може знадобитися

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						15
Змн.	Арк.	№ докум.	Підпис	Дата		

структура для використання подій. Він повинен бути здатним не лише «*слухати*» та «*реагувати*» на події, але й «*викликати*» свої події, які запускатимуть реакції в інших частинах програмної системи [14].

е) Конфігурація

Оскільки модуль є невеликою настроюваною підсистемою, існує потреба забезпечити конфігурацію модуля відповідно до потреб користувача.

1.2 Переваги та недоліки модульного проектування програмних систем

Модульна архітектурна конструкція була вдосконалена, щоб виправити недоліки популярних багаторівневих та SOA-архітектур. Завдяки впровадженню гранулярності та розділення послуг, ці стилі проектування підтримали проектування та розробку в іншому випадку дуже складних програмних систем, можна виокремити переваги і недоліки таблиця 1.1

Таблиця 1.1

Підсумок переваг і недоліків модульної системи

№ (S/N)	Переваги	Недоліки
1	Можливість кастомізації (налаштування під користувача)	Архітектурна невідповідність
2	Менша взаємозалежність	Фізична мінливість
3	Сторонні розширення	Неточність у модульному аналізі
4	Незалежна розробка	Нечіткі міжмодульні межі
5	Невелике та надійне ядро додатка	Складність у виявленні нестабільностей
6	Можливість повторного використання	—
7	Придатність до рефакторингу	—
8	Масштабованість	—

Крім того, модульна системна архітектура є перевагою, оскільки вона забезпечує:

а. Налаштування

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						16
Змн.	Арк.	№ докум.	Підпис	Дата		

Загальний стандарт визначає системи, функціональність та послуги яких можуть змінюватися завдяки модульній конструкції. Увімкнення або вимкнення деяких модулів може повністю змінити те, як система працює та надаються послуги. Університет Кібабії у західній Кенії посилив потребу в модульних налаштуваннях своєї системи планування ресурсів підприємства (ERP), де різні модулі ERP налаштовуються відповідно до потреб користувача, не впливаючи на інші модулі системи [15]. Такої гнучкості неможливо досягти при використанні багаторівневого або SOA-підходу.

в. Менша взаємозалежність

Кожен модуль у системі є більш незалежним від самої основної програмної системи. Доки інтерфейси сумісні, як модулі, так і основна програмна система можуть розвиватися незалежно.

с. Розширення третіх сторін

Ренгайя зазначає, що модулі не є частиною основної системи та взаємодіють з нею та один з одним лише через чітко визначені точки входу. Таким чином, модулі можуть розроблятися третіми сторонами [16]. Система ERP в Університеті Кібабії охоплює модулі, розроблені сторонами, відмінними від основного постачальника. Наприклад, модуль управління навчанням розроблено спільнотою Moodle з відкритим вихідним кодом [17].

д. Незалежний розвиток

Оскільки основна система та модулі мають незалежний модульний дизайн, їх можуть розробляти зовнішні розробники. Ця особливість принесла користь багатьом проектам безкоштовного програмного забезпечення з відкритим вихідним кодом (FOSS) [18]. Крім того, основні системи кожного модуля можуть випускатися з незалежними циклами випуску та потенційно розроблятися за допомогою різних технологій та інструментів. Модульна система медичних записів OpenMRS має різні модулі, розроблені з використанням різних технологій. Хоча основна система розроблена на Java та використовує фреймворк Spring, репозиторій модулів містить модулі, розроблені з використанням фреймворків AngularJS, ReactJS та Vue, серед інших

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						17
Змн.	Арк.	№ докум.	Підпис	Дата		

фреймворків, що чітко вказує на те, як модульність дозволяє здійснювати незалежну розробку [18]. Аналогічно, Нардуццо та Россі у своєму дослідженні проектування складних програмних артефактів пояснюють досягнення різних проектів безкоштовного/відкритого програмного забезпечення (FOSS) (серед них: операційна система GNU, ядро Linux, ядро HURD) модульним підходом, прийнятим цими проектами FOSS [8].

Е. Менше ядро програми

Розмір основної програмної системи значно зменшується, оскільки багато функцій можна реалізувати за допомогою модулів. Це призводить до кращого розуміння системи та покращення її обслуговування.

Ф. Можливість повторного використання

Добре продуманий модуль є повністю повторно використовуваним. Ви можете повторно використовувати старе рішення, коли у вас знову виникне така сама потреба.

Г. Рефакторизація

Чим менше взаємозалежностей у проєкті, тим легше вносити великі зміни в кілька модулів [19].

Н. Масштабованість

Модульна конструкція дозволяє масштабувати програмні застосунки, оскільки практично неможливо створювати великі застосунки без належної модуляризації. Брінкман та Деламор зазначають, що складність монолітної системи, побудованої без модулів, знизить продуктивність [19].

Хоча модульна архітектура забезпечує зручну архітектуру для програмних проєктів і ставить розробників програмного забезпечення в центр еволюції програмного забезпечення, підкреслюючи тим самим, що гнучкість бізнесу може бути забезпечена завдяки критичному проєкту, вона також пов'язана з деякими проблемами:

А. Архітектурна невідповідність

Одна з проблем пов'язана з системами, які інтегрують ортогональну функціональність в єдиний модульний артефакт, що вводить штучне поєднання

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						18
Змн.	Арк.	№ докум.	Підпис	Дата		

функціональних можливостей, зумовлене певними вимогами реалізації. Хоча таке поєднання може мати певну локально оптимальну продуктивність, це часто може відбуватися за рахунок глобальної оптимальності програмної системи.

в. Фізична мінливість

Фізична мінливість стосується того, як працюють різні змінні модулі в одній програмній системі. Щоб програмний застосунок підтримував усі ці змінні модулі, йому потрібно буде забезпечити загальний інтерфейс/точку входу. Під час проектування узагальненого інтерфейсу часто трапляється, що ні об'єднання всіх можливих можливостей, ні перетин таких можливостей не є задовільним. Таким чином, загальний інтерфейс продукту підтримує можливості, які часто знаходяться посередині [20].

с. Неточність в аналізі модульності

Вибираючи певні архітектурні абстракції, стилі та механізми для декомпозиції системи, архітектори можуть залишити деякі функціональні можливості/сервіси немодуляризованими. Таким чином, ці функціональні можливості не будуть вичерпно передбачені окремими модульними одиницями в описі архітектури, часто залишаючи функціональні сліди в деяких модулях. Такий архітектурний опис може призвести до деяких хибнопозитивних результатів у процесі оцінки архітектури [21]. В архітектурному описі залишається повністю модульована функція, що може призвести до хибнонегативних результатів у процесі аналізу.

д. Розмиті міжмодульні межі

Залежність між системними вимогами є ключовою інформацією для архітекторів програмного забезпечення для проектування чітко визначених модулів. Однак, оскільки модулі обирають різні шляхи для змін у дизайні, існуючі метрики зв'язку можуть неточно визначати архітектурні міжмодульні - залежності. Загальним результатом є розмиті міжмодульні межі та тісний зв'язок інтерфейсів [21]. Аналогічно, еволюція програмної системи може продовжувати збільшувати складність дизайну, фактично пропускаючи скінченні деталі модульних характеристик. Це явище, якщо його не контролювати, також

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						19
Змн.	Арк.	№ докум.	Підпис	Дата		

приведе до нечіткого опису міжмодульних меж.

Е. Неточність у визначенні нестабільностей:

Вихід модульної системи базується на безперебійному та плавному функціонуванні кожного модуля та зв'язку між модулями. Там, де в системі є нестабільність, залишається проблема у визначенні джерела нестабільності в складній модульній системі. Проблема полягає в тому, що звичайні метрики не можуть точно ідентифікувати нестабільний елемент/модуль.

1.3 Метрики оцінювання якості та ефективності модульних систем

Модуль – це сукупність компонентів, які мають спільну характеристику та збираються відповідно до цієї спільної характеристики для досягнення визначеної мети. У кожному модулі компоненти міцно пов'язані між собою та відносно слабо пов'язані з компонентами в інших модулях. Важливо вимірювати міцність цих зв'язків між компонентами, що визначає модульність програмної системи. Архітектори програмного забезпечення використовують метрики модульності програмного забезпечення для моніторингу проектів, виявлення невідповідностей та виявлення ризиків, таких як низька модульність, у програмних проектах з ранніх стадій розробки проекту [22]. Деякі з метрик для вимірювання модульності включають:

А. Згуртованість

Згуртованість стосується взаємозв'язку між внутрішніми елементами та того, наскільки цілісні ці зв'язки [23].

Теорія щодо проектування структурованих систем зосереджується на різних формах зв'язності для модульної архітектури:

1) Функціональна зв'язність, що стосується випадків, коли всі елементи модуля сприяють виконанню одного чітко визначеного завдання.

2) Послідовна зв'язність - це коли елементи модуля групуються, оскільки вихід одного елемента є входом для іншого елемента (наприклад, функція, яка зчитує дані з файлу та обробляє їх).

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						20
Змн.	Арк.	№ докум.	Підпис	Дата		

- 3) Логічна зв'язність у випадках, коли елементи групуються логічно.
- 4) Часова зв'язність, коли компоненти пов'язані між собою в часовому просторі, наприклад, модуль ініціалізації.
- 5) Комунікаційна зв'язність, якщо всі дії в модулі діють на одні й ті ж вхідні або вихідні дані.
- 6) Процедурна зв'язність у випадках, коли дії модуля послідовно виконуються разом для виконання певного завдання.
- 7) Випадкова зв'язність, коли елементи в модулі пов'язані між собою незапланованим та випадковим чином. Цей зв'язок вважається безглуздом, оскільки він може призвести до подальшої декомпозиції модуля [24].

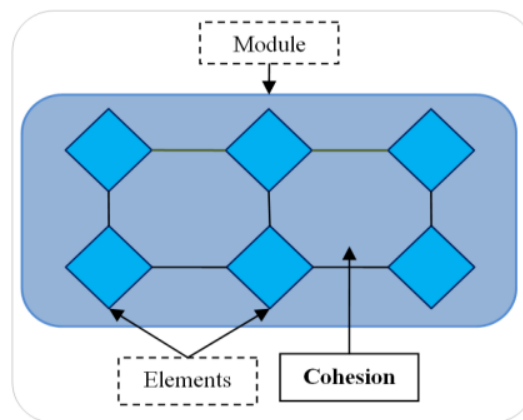


Рисунок 1.7 - Модульна когезія

в. Зв'язок (ступінь взаємозалежності)

Зв'язок у програмній інженерії стосується ступеня взаємозалежності між програмними модулями, вимірюючи, наскільки тісно пов'язані два модулі [25]. Документ зі специфікацією вимог до програмного забезпечення визначає різні аспекти міжмодульної залежності та незалежності.

Ступінь взаємозалежності між модулями можна загалом класифікувати як аферентний та еферентний зв'язок. Аферентний зв'язок – це метрика, яка вимірює загальну кількість елементів поза модулем, що залежить від елементів усередині модуля. На противагу цьому, еферентний зв'язок вимірює загальну кількість елементів усередині модуля, що залежать від елементів поза модулем

[22]. Крім того, визначає різні форми зв'язку в модульних конструкціях: 1) Зв'язок контенту, який стосується взаємозалежності, коли один модуль може безпосередньо отримувати доступ або змінювати, або звертатися до внутрішнього вмісту іншого модуля. Це найвища форма взаємозалежності. 2) Спільний зв'язок, коли кілька модулів мають доступ для читання та зміни спільної глобальної області. 3) Зв'язок керування у випадках, коли один модуль контролює функціонування іншого модуля. 4) Зв'язок даних відбувається, коли один або декілька модулів мають певний параметризований обмін даними. 5) Зв'язок штампа, коли модулі мають спільні структури даних, але працюють над різними розділами спільної структури даних [24].

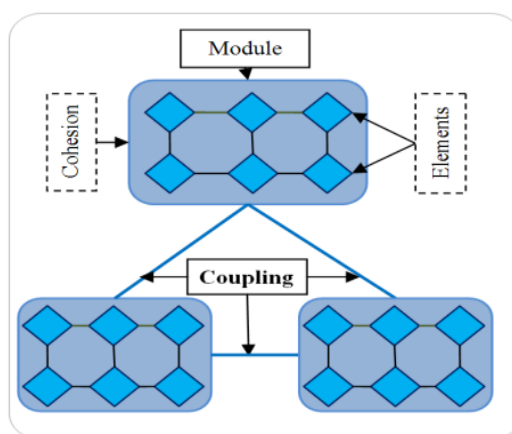


Рисунок 1.8 - Модульне з'єднання

с. Нестабільність

Метрика нестабільності вимірює нестабільність компонентів, де стабільність вимірюється шляхом обчислення зусиль, необхідних для зміни компонента без впливу на інші компоненти в програмному застосунку. Сантос та ін. підтримують цю позицію, аналізуючи міру нестабільності Мартіна. Вони вважають, що якщо об'єкт має високе значення нестабільності, то існує високий ризик небажаних змін, які можуть вплинути на поведінку аналізованого об'єкта через зміни в інших системних об'єктах [25].

Якість програмного артефакту та його довговічність значною мірою визначаються його архітектурою [26]. Тому вкрай важливо, щоб архітектура

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						22
Змн.	Арк.	№ докум.	Підпис	Дата		

програмного забезпечення розвивалася з часом, щоб задовольняти потреби користувачів програмного забезпечення, що постійно змінюються. Еоін Вудс прагматично розглядає п'ять епох еволюції програмних систем та супутні п'ять етапів архітектур програмного забезпечення [27]. Огляд висвітлює шлях, який модульність пройшла протягом архітектурних епох до сучасної архітектури. З кожною еволюцією модульність змінювалася від початкових монолітних модулів до сучасних мікросервісних модулів мірою йшла паралельно з модульністю індустрії програмного забезпечення, причому методи та потреби архітекторів змінювалися у відповідь на мінливі виклики, з якими стикається галузь.

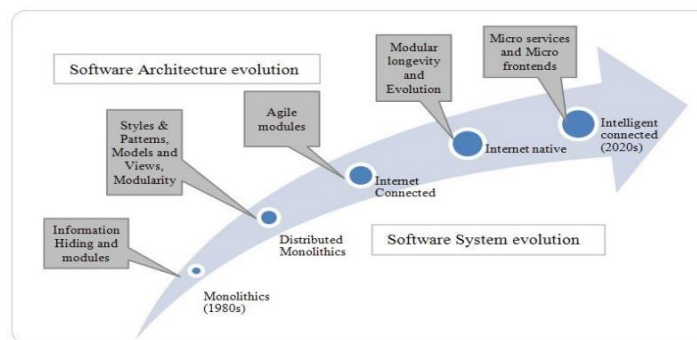


Рисунок 1.9 - Щоденник подорожі по модульності

Сучасні програмні системи є більш мережецентричними, а інтелектуальна модульність трансформувалася, щоб забезпечити інтелектуальні інтерфейси та точки входу, які більше не потребують прив'язки до фізичних обчислювальних станцій. Девід Гарлан підтримує цю позицію та визначає мережецентричну природу програмних артефактів як рушійну силу для сучасних та майбутніх програмних архітектур [28]. Розвиваючись для підтримки повторного використання коду за допомогою сучасних фреймворків, концепція модульності вбудовується в нові архітектури, водночас дозволяючи розробляти нові програмні парадигми [27].

Концепція модульності вбудована в нові хмарні технології, такі як SaaS, PaaS та IaaS. Оскільки бізнес стає більш диференційованим, його потреби в

програмному забезпеченні стають більш індивідуальними, що вимагає налаштування програмного забезпечення для створення унікального досвіду. Постачальники SaaS використовують модульні SaaS-системи, де клієнти можуть отримувати різний досвід, вмикаючи або вимикаючи пакети послуг у модулях.

Подорожній нотаток Еойна показує, що сучасні та майбутні інтелектуальні, інтернет-орієнтовані системи продовжуватимуть бути динамічними та складатися з дрібнозернистих мережевих модулів (мікросервісів) [29]. Модулі часто будуються на платформах SaaS/PaaS, що дозволяє клієнтам вибирати, які модулі відповідають їхнім потребам, економічним можливостям та технічній життєздатності для задоволення унікальних вимог. Модульність – це архітектурний дизайн, який готовий залишатися домінуючим стилем проектування програмних систем.

Модульність займає ключове місце в проектуванні хороших архітектур програмних систем. Кілька програмних проектів застосували модульний дизайн, виходячи за рамки багаторівневих та SOA-підходів. Архітектура вирішує проблему монолітної складності та гранулярних багаторівневих систем, які важко проектувати, розробляти та впроваджувати. Дослідження довели та задокументували, що модульний дизайн забезпечує можливість рефакторингу, повторного використання, налаштування, співпрацю програмного забезпечення та масштабованість. Крім того, встановлено, що модульність призводить до мінімалістичних основних застосунків, що дозволяє глибоко зрозуміти та спростити обслуговування основної системи.

Однак, модульне проектування також пов'язане з деякими труднощами: архітектурна невідповідність, фізична мінливість та розмиті межі між модулями наражають розробників на перешкоди, які потребують вирішення. Однак, слід зазначити, що значний розвиток показників вимірювання модульності дозволить передбачати ці проблеми та, таким чином, вирішувати їх на початкових етапах розробки програмних проектів. Крім того, сучасні тенденції в проектуванні та розробці програмних систем показують, що модульність залишається домінуючим стилем, де існуючі та нові стилі включають концепцію модульності

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						24
Змн.	Арк.	№ докум.	Підпис	Дата		

для усунення властивих обмежень немодульних систем.

1.4 Висновки по розділу

У першому розділі розглянуто теоретичні основи побудови модульних програмних систем. Проаналізовано ключові принципи модульної архітектури, що забезпечують розподіл системи на незалежні компоненти з чітко визначеними інтерфейсами. Досліджено переваги та обмеження модульного підходу, а також метрики оцінювання якості таких систем. Отримані результати формують базове розуміння архітектурних засад модульного проєктування програмного забезпечення.

					БР.ІІ – 00.00.00.000 ІЗ	Арк.
						25
Змн.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 2. МЕТОДИ ТА ІНСТРУМЕНТИ ПРОЄКТУВАННЯ МОДУЛЬНИХ ПРОГРАМНИХ СИСТЕМ

2.1 Архітектурні підходи та передумови створення модульних систем

Розуміння нашого підходу вимагає досвіду в дослідженні архітектури програмного забезпечення, великого обсягу попередніх робіт з розробки мов опису архітектури програмного забезпечення (ADL) та деяких відомостей про сам XML.

З моменту розвитку абстракції та модуляризації в розробці програмного забезпечення, абстрактні моделі (формальні чи інші) структури системи відіграли ключову роль у розробці великих програмних систем. Зі зростанням складності систем також зростала підтримка все більш абстрактних концепцій у мовах програмування та середовищах [30]. Об'єктно-орієнтована розробка, наприклад, забезпечує корисну абстракцію вище рівня операторів програми, інкапсулюючи пов'язану функціональність та стан в одну конструкцію: об'єкт. Дослідження архітектури програмного забезпечення призвели до рівня абстракції, що перевищує рівень простих об'єктів, модулів або рядків коду. Описи архітектури програмних систем зазвичай складаються щонайменше з трьох ключових сутностей: компонентів, з'єднувачів та конфігурацій.

Компоненти є локусами обчислень в архітектурі та можуть бути - пов'язаними зі станами. У цьому сенсі вони нагадують об'єкти, але компоненти та об'єкти відрізняються кількома способами. Компоненти, як правило, більші за об'єкти, часто інкапсулюючи цілу готову бібліотеку або інструмент. Компоненти рідко створюються та знищуються протягом життя програмної системи, об'єкти створюються та знищуються постійно.

Конектори – це місця комунікації в архітектурі. Конектори можуть бути неявними, як виклик процедури, або явними, що нагадують компонент. Прості конектори можуть бути реалізовані як базова черга повідомлень або точка входу API, складні конектори можуть інкапсулювати проміжне програмне

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						26
Змн.	Арк.	№ докум.	Підпис	Дата		

забезпечення або перетинати межі процесів. Це може розмити межу між компонентами та конекторами; гарним емпіричним правилом є те, що конектори можуть змінювати форму або синтаксис даних, які вони передають, але загалом не впливають на семантику рівня програми; тобто вони не змінюють значення даних.

Конфігурації описують розташування компонентів та з'єднувачів, а також властивості цього конкретного розташування [31]. Найчастіше це виражається через набір явних зв'язків між компонентами та з'єднувачами, які визначають архітектурну топологію. Крім того, це може включати обмеження або шаблони розташування компонентів та з'єднувачів або їхньої поведінки.

Мови опису архітектури

Окрім компонентів, з'єднувачів та конфігурацій, опис програмної системи на архітектурному рівні може містити майже будь-яку інформацію. Види інформації, присутні в описі архітектури, залежать від області, в якій розробляється програмне забезпечення, та від того, як використовується опис архітектури.

Поширення спеціалізованих мов опису архітектури протягом останнього десятиліття підтверджує це. Розширено специфікації компонентів та роз'ємів інформацією про поведінку мовою комунікації послідовних процесів (CSP), щоб можна було аналізувати архітектури. Описано компоненти з частково - впорядкованими наборами подій, які називаються POSET, щоб можна було моделювати їхню поведінку. Описано конфігурації систем, розподілених між кількома процесами та машинами.

Незважаючи на те, що ці нотації мають багато спільних концептуальних елементів (компоненти, конектори, інтерфейси, посилання тощо), вони не мають спільної навіть синтаксичної основи. Це робить інструменти кожної нотації несумісними з іншими. Інші ADL, аналогічно несумісні. Проблему посилює те, що всім цим мовам бракує підтримки розширюваності, тому додавання функцій до будь-якої з них вимагатиме значних змін у всіх допоміжних інструментах.

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						27
Змн.	Арк.	№ докум.	Підпис	Дата		

Найбільш помітним винятком з цього правила в архітектурній спільноті, яка була розроблена як мова архітектурного обміну. Ядро Асте складається з основних типів конструкцій, які зустрічаються практично в кожній архітектурній нотації: компоненти, з'єднувачі, інтерфейси (які називаються портами та ролями), посилання (які називаються з'єднаннями) тощо. Кожна з цих основних сутностей декорована набором довільних властивостей пари ім'я-значення. Ця стратегія дозволяє певну розширюваність, але її корисність обмежена кількома обмеженнями.

```
<name lang="en-us">
  <first>John</first>
  <last>Doe</last>
</name>
```

Рисунок 2.1 - Приклад розміченого тексту в XML.

(1) Неможливо розширити основний набір елементів (компоненти, з'єднувачі, порти, ролі) для додавання нових сутностей першого класу. Це забороняє архітектурні конструкції, які природним чином не представлені як декорації на одному з основних елементів, таких як дерева версій компонентів.

(2) Важко кодувати та керувати складними структурами даних як наборами властивостей пар "ім'я-значення".

(3) Асте не підтримується метамовою, яка дозволяє опис допустимих властивостей та їх форматування.

Таким чином, здатність Асте служити мовою обміну універсальним призначенням або розширюваною мовою ADL зменшується.

XML як основа для розширюваної мови моделювання

Побудова розширюваної мови моделювання для програмних систем вимагає способу визначення допустимих конструкцій у мові та способу розширення або зміни конструкцій у мові для додавання або модифікації функцій. Існує багато металов та середовищ, в яких можна створити розширювану модульну мову, серед них XML, Lisp, GME. З технічної та

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						28
Змн.	Арк.	№ докум.	Підпис	Дата		

теоретичної точки зору, ми могли б використовувати будь-яке з цих середовищ для розробки нашої інфраструктури (хоча не всі вони еквівалентні; технічні відмінності між цими підходами є). Однак з практичної точки зору XML має набагато більшу підтримку з боку комітетів зі стандартизації, постачальників інструментів та практиків, ніж будь-яка інша альтернатива. XML є платформно-нейтральним, не прив'язаним до будь-якої конкретної апаратної чи мережевої архітектури. Багато допоміжних стандартів підтримують XML, такі як XLink, який дозволяє зв'язувати всередині та між XML-документами, та XPath, який дозволяє індексувати певні елементи та атрибути в документі. Доступна безліч готових інструментів для побудови, візуалізації, зберігання та маніпулювання XML-документами. Це не означає, що XML є ідеальним навіть для нашого застосування. Конкретні недоліки використання XML детально. Однак ми вважаємо, що підтримка XML модульної розширюваності та зовнішні переваги значно переважають його технічні недоліки [32].

XML-документи – це текстові документи, в яких частина тексту розмічена за допомогою спеціально відформатованих тегів, що визначають початок і кінець сегмента розміченого тексту. Дані, розділені початковим і кінцевим тегами, називаються елементами; початкові теги можуть містити додаткові анотації, які називаються *атрибутами*. Дані, розмічені елементами та атрибутами, показано на рисунку 2.1.

Можна ввести граматику елементів та атрибутів над XML-документами, використовуючи одну з кількох доступних метамов. Доступно кілька метамов для визначення XML-граматик, дві з яких були стандартизовані Консорціумом Всесвітньої павутини (W3C): DTD та XML-схеми. DTD пропонують обмежені механізми для підтримки модульної розширюваності, що підтверджується стандартом *Модуляризація XHTML*. Однак XML-схеми набагато більш здатні в цьому відношенні, пропонуючи простий спосіб визначення конструкцій в одній схемі та їх розширення в іншій за допомогою механізму, дуже схожого на об'єктно-орієнтоване підтипування. Підтип може додавати нові елементи та атрибути до моделі вмісту свого супертипу. Однак, на відміну від об'єктно-

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						29
Змн.	Арк.	№ докум.	Підпис	Дата		

орієнтованих типів, підтипи XML-схем також можуть бути більш обмежувальними, ніж їхні супертипи, у певних аспектах. Зокрема, вони можуть обмежувати кардинальність елементів у своєму супертипі. Наприклад, якщо елемент у супертипі має потужність 0-n, потужність елемента в обмеженому підтипі може бути 0-0, що повністю виключає елемент.

2.2 Методи та підходи до побудови модульної архітектури

Існуючі мови опису архітектури є або занадто специфічними (наприклад, одноцільові ADL, такі як Rapide, Darwin або Koala), або занадто загальними (наприклад, Асме). Крім того, вартість адаптації нотації та її інструментів для підтримки нової функції - навіть коли ця функція лише незначно розширює можливості цільової ADL - є непомірно високою. Щоб виправити це, ми розробили інфраструктуру, що підтримує модульну розширюваність. Нашими цілями під час створення цієї інфраструктури були:

- Він повинен накладати якомога менше обмежень на те, що можна виразити на рівні архітектури;

- Це повинно дозволяти додавання нових функцій моделювання та зміну існуючих функцій з часом;

- Це повинно дозволяти експериментувати з новими функціями та їх комбінаціями;

- Він повинен забезпечувати бібліотеку корисних модулів загального призначення, що застосовуються до широкого кола областей;

- Це повинно дозволяти повторне використання визначених функцій моделювання в інших проектах; та

- Він повинен забезпечити розробників інструментів підтримкою для створення та маніпулювання архітектурними моделями, навіть коли базова нотація може і буде змінюватися.

Основні елементи нашої інфраструктури показано на рисунку 2.2. В основі інфраструктури лежить набір мовних модулів, реалізованих у нашій

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						30
Змн.	Арк.	№ докум.	Підпис	Дата		

реалізації у вигляді XML-схем. Ці схеми визначають конструкції моделювання та розширюють конструкції моделювання з інших схем. Деякі з цих схем є відносно універсальними, що містять функції моделювання, застосовні до багатьох областей. У нашій інфраструктурі ці універсальні схеми разом називаються xADL 2.0. Інші схеми можуть бути більш специфічними для певної області. Розкладання функцій на окремі схеми є нетривіальним і (ми вважаємо) є ключовим фактором підтримки гнучкості цієї інфраструктури. Над цим мовним рівнем знаходиться рівень інструментів, орієнтованих на синтаксис. Ці інструменти використовують синтаксис, визначений у схемах, для автоматичного надають послуги, які є корисними загалом незалежно від вмісту будь-якої конкретної схеми, - до них належать парсери, серіалізатори, прив'язки даних та синтаксично-орієнтовані редактори. Основна перевага цих інструментів полягає в тому, що їх не потрібно змінювати, коли до мовного рівня додаються нові схеми. Нарешті, інфраструктура містить рівень семантичних інструментів.

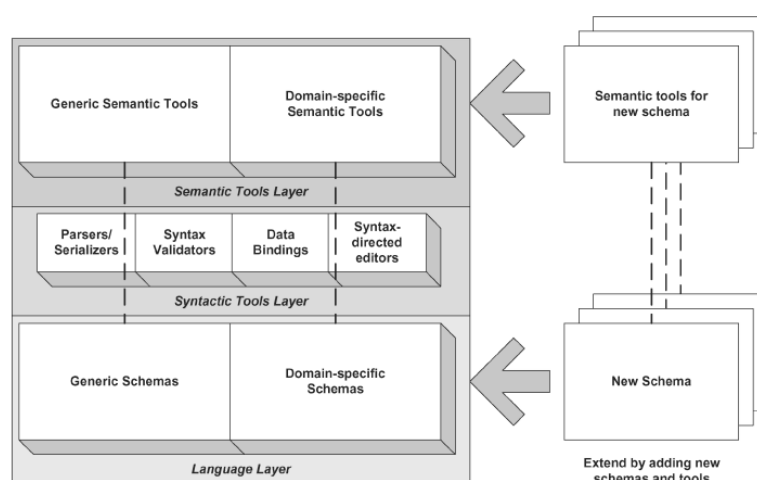


Рисунок 2.2 - Огляд підходу, що зображує взаємозв'язок між мовними модулями.

Реалізованими в XML-схемах, синтаксичними інструментами, що працюють з будь-якими схемами, та семантичними інструментами, специфічними для схеми. Ці інструменти створені для підтримки функцій певних схем у мовному рівні та зазвичай використовують синтаксично-

орієнтовані інструменти для читання та запису описів архітектури. Ці інструменти можуть включати візуальні редактори, інструменти аналізу, генератори коду тощо. Оскільки ці інструменти пов'язані з певними схемами та функціями схем, їх може знадобитися змінити, коли схеми у мовному рівні змінюються.

Ця інфраструктура сприяє простому процесу розробки на основі архітектури програмного забезпечення [33]. Спочатку архітектор шукає ADL (у цьому випадку композицію схем), яка вже розроблена та відповідає потребам поточного проекту. Якщо такої ADL не існує, то вибирається найближча ADL (та, що має найбільшу кількість функцій/модулів, застосовних до поточного проекту). Непотрібні модулі можна видалити.

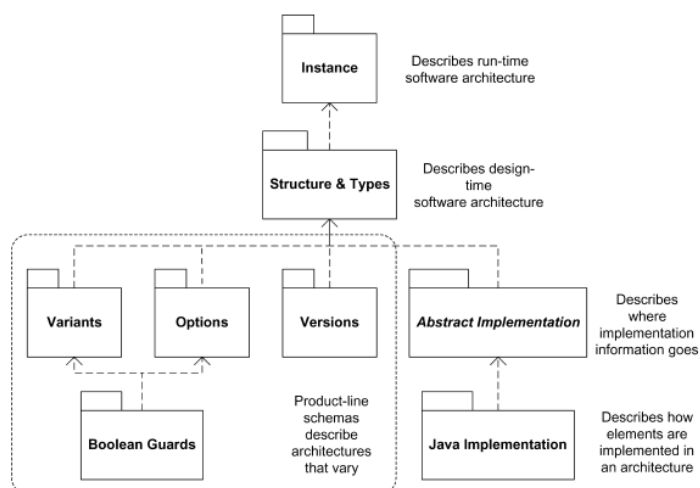


Рисунок 2.3 - XML-схеми xADL 2.0 та їхні зв'язки.

Додаткові необхідні функції можуть надходити з одного з трьох джерел: шляхом повторного використання модулів з інших ADL, шляхом розширення існуючих модулів або шляхом розробки нових модулів, якщо відповідної основи не існує. Очевидно, що повторне використання або розширення існуючих модулів є кращим за розробку нових через можливість повторного використання супутніх інженерних знань і, що ще важливіше, семантичних інструментів, які були розроблені разом з оригінальними модулями.

Мови опису архітектури повинні щонайменше надавати методи для опису компонентів, з'єднувачів, інтерфейсів та посилань [34]. Окрім цього ядра, кожна область матиме унікальні потреби моделювання. Однак, існують деякі функції ADL, які загалом застосовні та бажані в багатьох різних областях. Ми вважаємо, що добре розроблені реалізації цих функцій, які є синтетично- та семантично сумісними одна з одною та основними конструкціями, є ключовою частиною успішної інфраструктури розробки ADL.

Відповідно, ми використали XML-схеми для визначення набору універсальних конструкцій, корисних для моделювання архітектур програмного забезпечення в багатьох областях. Ці конструкції можна використовувати як є, або їх можна розширити для підтримки додаткових функцій моделювання. Замість того, щоб визначати всі ці універсальні конструкції в одній великій монолітній XML-схемі, ми вирішили згрупувати їх у схеми на основі їхнього цільового призначення. Результатом є набір багаторазових «модулів», які разом складають універсальну ADL, відому як xADL 2.0. Поточний набір схем xADL 2.0 та їхні залежності показано на рисунку 2.3.

Розуміючи, що ці схеми будуть використовуватися в багатьох областях, ми намагалися зробити їх якомога більш загальними. Наприклад, xADL 2.0 визначає основні конструкції, такі як компоненти та конектори, але не диктує, як вони повинні поводитися або як їх можна пов'язувати між собою. Ці характеристики різняться.

з ADL до ADL, щоб їх можна було вказати в схемах розширення. Однак, залежно від ситуації та вимог моделі, ці загальні функції можуть бути достатніми самі по собі, як показує наш досвід роботи з ними.

Основні, загальні функції xADL 2.0:

- (1) розділення моделей часу виконання та часу проектування програмної системи;
- (2) зіставлення реалізацій, що відображають специфікацію ADL архітектури на виконуваний код; та
- (3) здатність моделювати аспекти архітектурної еволюції та архітектури

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						33
Змн.	Арк.	№ докум.	Підпис	Дата		

лінійки продуктів.

Таблиця 2.1

Схеми та функції xADL 2.0

Призначення (Purpose)	Схема (Schema)	Особливості / Функції (Features)
Моделювання під час проєктування та виконання	Екземпляри (Instances)	Компоненти часу виконання, конектори, інтерфейси та екземпляри зв'язків; субархітектури, загальні групи.
	Структура та типи (Structure & Types)	Компоненти часу проєктування, конектори, інтерфейси, зв'язки, субархітектури, загальні групи.
Мапінг реалізації (Implementation Mappings)	Абстрактна реалізація (Abstract Implementation)	Заповнювач (placeholder) для даних про реалізацію компонентів, конекторів та інтерфейсів.
	Реалізація на Java (Java Implementation)	Конкретні дані про реалізацію для Java-компонентів, конекторів та інтерфейсів.
Управління еволюцією архітектури та архітектури лінійок продуктів	Версії (Versions)	Графи версій для типів компонентів, конекторів та інтерфейсів.
	Опції (Options)	Необов'язкові (опціональні) компоненти проєктування, конектори, інтерфейси та зв'язки.
	Варіанти (Variants)	Варіанти типів компонентів та конекторів під час проєктування.

Розподіл цих високорівневих функцій на окремі схеми показано в таблиці 2.1. У цьому розділі ми детально обговоримо кожен схему. Щоб проілюструвати, як ці схеми застосовуються для опису архітектури програмної системи, ми використаємо xADL 2.0 для опису гіпотетичної програмної системи для продукту споживчої електроніки.

Традиційні ADL розглядали архітектуру програмного забезпечення як артефакт часу проєктування, зосереджуючи свої можливості моделювання на фазі проєктування розробки. Однак дослідження еволюції та динаміки програмного забезпечення часу виконання показали, що корисно підтримувати архітектурну модель системи також під час виконання. Моделі часу

проектування та часу виконання системи будуть подібними, але не ідентичними. Під час проектування модель системи може містити основні метадані про елементи (автор, розмір, текстові описи), специфікацію передбачуваної (не спостережуваної) поведінки та обмеження на розташування елементів. На противагу цьому, аспекти часу виконання системи, які можуть бути фіксовані в ADL, включають фізичний розподіл програмної системи між машинами, ідентифікатор процесу, в якому виконується кожен компонент, або поточний стан певного з'єднувача (наприклад, працює, обробка повідомлень, заблоковано).

Дві схеми підтримують розділення часу проектування та виконання в xADL 2.0. Вони називаються схемами Structure & Types та INSTANCES відповідно. Обидві схеми підтримують такі функції.

— *Компоненти*: Компоненти є локусами обчислень в архітектурі. У цих схемах компоненти визначені узагальнено; тобто вони мають лише унікальний ідентифікатор та короткий текстовий опис, а також набір інтерфейсів (описаних далі).

— *Конектори*: Конектори є місцями зв'язку в архітектурі. Подібно до компонентів, конектори також мають лише унікальний ідентифікатор, текстовий опис та набір інтерфейсів.

— *Інтерфейси*: Інтерфейси – це портали компонентів та конекторів до зовнішнього світу; те, що ми називаємо «інтерфейсами», також відоме як «порти» в інших ADL. Наприклад, якщо компонент реалізує певний API, він, ймовірно, міститиме інтерфейс, який вказує на те, що інші компоненти можуть здійснювати виклики до цього API на цільовому компоненті. У цих схемах інтерфейси мають унікальний ідентифікатор, текстовий опис та напрямок (індикатор того, чи надано інтерфейс, чи є він обов'язковим, чи і те, й інше). Конкретна семантика інтерфейсу не вказана в xADL 2.0.

— *Посилання*: Посилання – це зв'язки між елементами, що визначають топологію архітектури. У більшості архітектурних нотацій посилання з'єднують інтерфейси, але це обмеження не є обов'язковим.

— *Підархітектури*: Компоненти та конектори можуть бути атомарними

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						35
Змн.	Арк.	№ докум.	Підпис	Дата		

(без подальшого розбиття на архітектурному рівні) або складеними. Складені компоненти та конектори мають внутрішні архітектури, які називаються підархітектурами.

—*Загальні групи*: Групи - це просто колекції вказівників на елементи в описі архітектури. У цих схемах група не має семантики. Групи зі специфічними значеннями (наприклад, спільне авторство, спільна платформа, подібна функціональність) можуть бути вказані в схемах розширення.

Оскільки схеми «I NSTANCES» та «S STRUCTURE & TYPES» мають визначення для основних архітектурних конструкцій (компонентів, з'єднувачів тощо), вони є ядром xADL 2.0. Зберігаючи визначення цих елементів в обох схемах, їх можна розширювати окремо: схему «I NSTANCES» слід розширити для надання додаткових даних часу виконання, а схему «S STRUCTURE & TYPES» слід розширити для надання додаткових даних часу проектування. Щоб забезпечити відстеження, XLinks пов'язують елементи часу виконання з їхніми аналогами часу проектування.

Щоб побачити, як ці схеми можна використовувати для моделювання ядра архітектури програмного забезпечення, розглянемо програмну систему, яка може бути використана для керування низькоякісним телевізором. Такий пристрій може мати лише два програмні компоненти: один для взаємодії з телевізійним тюнером, а інший для керування інфрачервоним детектором, який використовується для прийому сигналів з пульта дистанційного керування. Ці два компоненти з'єднані між собою програмний роз'єм, який дозволяє інфрачервоному приймачу надсилати сигнали на телевізійний тюнер для зміни каналу. Графічне зображення цієї архітектури разом з її описом xADL 2.0 (з опущеною додатковою розміткою) показано на рисунку 2.4.

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						36
Змн.	Арк.	№ докум.	Підпис	Дата		

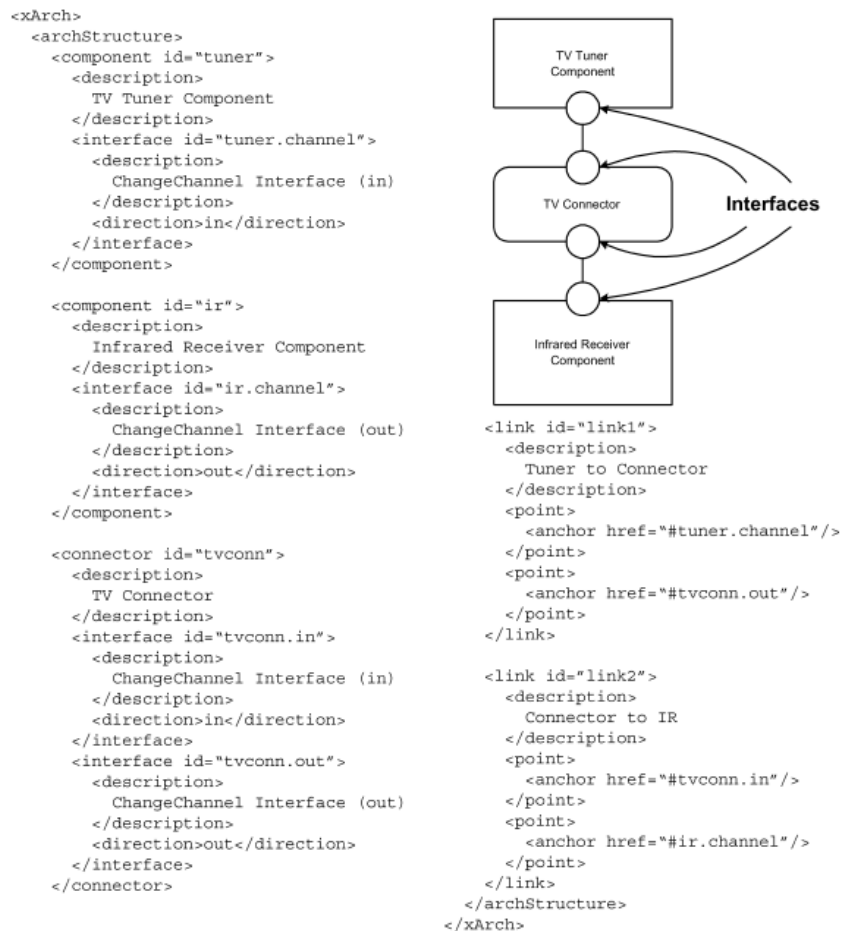


Рисунок 2.4 Діаграма прикладу архітектури телевізора з описом його структури під час проектування xADL 2.0 (скорочено, не XML-нотацією)

Типізація є важливою конструкцією в більшості ADL. xADL 2.0 включає базові структури системи типізації, яка підтримує рівність типів та композицію. У xADL 2.0 використання типів є необов'язковим. Кожен компонент, конектор або інтерфейс може необов'язково містити XML-посилання на тип; кілька елементів можуть використовувати один і той самий тип. Тип служить конструкцією, де спільні властивості елементів можна вказати один раз (у типі), а не в кожному оголошенні елемента. У схемі "Структура та типи" ці типи складаються з унікального ідентифікатора та текстового опису разом із набором підписів. Підписи - це задані інтерфейси; два компоненти або конектори одного типу повинні мати однакові типи інтерфейсів.

Щоб продемонструвати це, ми додамо типи до нашого телевізійного прикладу. Ми також додамо до нашого телевізора ще один компонент – тюнер

«картинка в картинці». Телевізори з функцією «картинка в картинці» потребують двох тюнерів для одночасного відображення обох каналів, але оскільки основний тюнер і тюнер «картинка в картинці» по суті ідентичні, вони матимуть один тип. Доповнення до нашої існуючої архітектури та її опис показано на рисунку 2.5.

Типи також використовуються в xADL 2.0 для підтримки підархітектур часу проектування (тобто компонентів або конекторів, внутрішні архітектури яких також визначені в xADL 2.0). Таким чином, два компоненти або конектори, що мають спільний тип, також мають спільну внутрішню архітектуру. Зокрема, типи можуть мати необов'язковий XLink на інший елемент ArchStructure, що описує внутрішню архітектуру, а також набір зіставлень між сигнатурами типу та інтерфейсами компонентів і конекторів у підархітектурі. Ці зіставлення служать для зв'язку зовнішньої архітектури з внутрішньою. Повний опис підархітектур у xADL 2.0 виходить за рамки цієї роботи, але кілька з них доступні на наших веб-сайтах

Деякі з наших передбачуваних семантичних обмежень для типів (наприклад, щоб тип компонента вказував на тип компонента або щоб інтерфейси відповідали своїм приписам, вираженим у сигнатурах) не можна виразити безпосередньо в XML-схемах. Однак ми створили інструменти та середовища перевірки, які допомагають користувачам дотримуватися та підтримувати ці обмеження у власних архітектурах. Зіставлення реалізацій

Другою важливою особливістю xADL 2.0 є підтримка відображення - архітектурних елементів часу проектування на виконуваний код. Кілька ADL, такі як MetaH, підтримують або вимагають відображення між специфікацією архітектури та її реалізацією [35]. Це важливо, якщо програмна система має автоматично створюватися на основі опису архітектури, і може допомогти в управлінні переходом від проектування системи до її реалізації.



Рисунок 2.5 - Діаграма розширеної версії прикладу телевізійної архітектури з додаванням компонента «картинка в картинці» та доповненнями до нашого попереднього опису xADL 2.0

Оскільки xADL 2.0 не прив'язаний до певної платформи чи мови реалізації, неможливо точно знати апріорі, які саме типи реалізацій будуть використовуватися. Очевидні можливості включають класи та архіви Java, DLL-бібліотеки Windows, спільні бібліотеки UNIX та компоненти CORBA, але складання вичерпного списку є неможливим. Щоб вирішити цю проблему, xADL 2.0 використовує дворівневий підхід.

Перший рівень специфікації є абстрактним і визначає, де в описі архітектури повинні розміщуватися дані реалізації, але не те, якими мають бути ці дані. Це вказує майбутнім розробникам, як розширити схеми xADL 2.0. додати дані для нової платформи реалізації. Схема абстрактної реалізації xADL 2.0 розширює схему структури та типів і визначає заповнювач для даних реалізації.

Цей заповнювач присутній для типів компонентів, роз'ємів та інтерфейсів. Таким чином, два елементи одного типу використовують одну реалізацію. Другий рівень специфікації є конкретним, він визначає, якими є дані реалізації для певної платформи або мови програмування. Конкретні схеми реалізації розширюють абстрактну реалізацію схема. xADL 2.0 включає схему реалізації J, яка КОНКРЕТНО визначає відображення компонентів, конекторів та типів інтерфейсів на класи Java; наразі ми також працюємо над додатковими схемами реалізації для відображення на вихідний код.

Завдяки такій конфігурації визначення конкретної реалізації для заданого елемента є простим [36]. Для елемента часу проектування, такого як компонент або конектор, користувач просто слідує за типом XLink елемента та отримує дані реалізації з типу. Елементи часу виконання, такі як екземпляри компонентів та екземпляри конекторів, потребують додаткового кроку: слідування за XLink від елемента часу виконання до елемента часу проектування, а потім слідування за XLink елемента часу проектування до його типу.

Ми можемо додати дані реалізації до типів компонентів та роз'ємів у нашому прикладі з телевізором, як показано на рисунку 2.6.

```
<xArch>
  <archStructure>
    ...
  </archStructure>

  <archTypes>
    <componentType id="tuner_type">
      <description>TV Tuner Type</description>
      ...
      <implementation>
        <mainClass>
          <javaClassName>edu.uci.isr.tv.TelevisionTuner</javaClassName>
          <url>http://www.isr.uci.edu/classes/tuner.jar</url>
          <initializationParameter>
            <name>tv_model</name>
            <value>Astro 5000</value>
          </initializationParameter>
        </mainClass>
        <auxClass>
          <javaClassName>edu.uci.isr.tv.TelevisionUtils</javaClassName>
          <url>http://www.isr.uci.edu/classes/tuner.jar</url>
        </auxClass>
      </implementation>
    </componentType>
  </archTypes>
</xArch>
```

Рисунок 2.6 - Розширений опис одного з типів телевізійних компонентів з додаванням деталей реалізації.

Тут ми показуємо, що компонент телевізійного тюнера реалізовано двома

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						40
Змн.	Арк.	№ докум.	Підпис	Дата		

класами Java, що знаходяться в одному JAR-архіві. Основний клас приймає параметр ініціалізації моделі телевізора, що корисно, якщо реалізація компонента є багатоцільовою або придатною для повторного використання в кількох контекстах (наприклад, для багатьох моделей телевізорів).

2.3 Моделювання еволюції архітектури та лінійок програмних продуктів

Багато ADL першого покоління зосереджувалися на моделюванні архітектури однієї програмної системи або продукту. Це недостатньо з двох причин. По-перше, архітектури розвиваються з часом. З розвитком продукту розвивається і його архітектура. По-друге, програмні продукти часто є частиною більшої лінійки продуктів. Лінійка продуктів - це сімейство пов'язаних програмних продуктів, які мають значні спільні частини своєї архітектури з певними точками варіації. Одна лінійка продуктів може містити продукти, локалізовані для певних регіонів або представляти різні набори функцій для маркетингових цілей (наприклад, Standard, Professional, Enterprise).

З точки зору моделювання, як еволюцію архітектури, так і лінійки продуктів можна розглядати, застосовуючи традиційні концепції від управління конфігурацією до архітектур програмного забезпечення [37]. Натхненний попередніми дослідженнями в галузі моделювання лінійок архітектурних продуктів, xADL 2.0 інтегрує ці концепції у вигляді трьох схем: схем « ВЕРСІЇ », « ОПЦІЇ » та « ВАРІАНТИ ». Версії записують інформацію про еволюцію архітектур та елементів, таких як компоненти, роз'єми та інтерфейси. Опції вказують на точки варіації в архітектурі, де структура може змінюватися шляхом включення або виключення групи елементів (компонентів, роз'ємів, посилань тощо). Варіанти вказують на точки в архітектурі, де одна з кількох альтернатив може бути замінена на елемент або групу елементів. Ми розробили ці три схеми для сумісності, щоб моделюючі могли вибирати використання лише схем « ВЕРСІЇ », « ОПЦІЇ » або « ВАРІАНТИ », або будь-якої комбінації цих трьох.

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						41
Змн.	Арк.	№ докум.	Підпис	Дата		

Версії. Схема версій додає конструкції версій до xADL 2.0. Вона визначає графи версій для типів компонентів, конекторів та інтерфейсів. У xADL 2.0 типи елементів архітектури є сутностями з версією. Рішення про версійність типів, а не таких елементів, як компоненти та конектори, було прийнято, оскільки ми вважаємо, що це найкраще відповідає семантиці системи типів. Наприклад, за допомогою версійності типів можна мати кілька екземплярів однієї й тієї ж версії компонента, конектора або інтерфейсу, просто створивши кілька екземплярів типу. Різні версії компонента, конектора або інтерфейсу можуть співіснувати в архітектурі, просто створивши екземпляри типу, які мають спільне дерево версій. Ми вважаємо, що це має сенс, оскільки з розвитком архітектур новіші версії елементів (типів) можуть мати інші характеристики, ніж старіші версії (наприклад, додаткові сигнатури).

Зв'язок між конкретними елементами (наприклад, компонентами), їх типами та графами версій зображено на рисунку 2.7. Тут тип ТВ-тюнера та тип HDTV-тюнера мають спільне походження. Тип HDTV-тюнера є пізнішою версією типу ТВ-тюнера, але обидві версії можуть бути включені в одну й ту саму архітектуру, якщо необхідно.

У цьому прикладі граф версій описує еволюцію окремого елемента . Однак, використовуючи механізм підархітектури на основі типів, визначений у схемі " СТРУКТУРА та типи " , граф версій може фіксувати еволюцію груп елементів або цілих архітектур. Це ще одна причина, чому ми вирішили версіювати типи.

Відповідно до загальної природи схем xADL 2.0, графи версій не обмежують зв'язок між різними версіями окремих елементів, наприклад, що вони повинні мати деякі спільні поведінкові характеристики або інтерфейси. Такі обмеження можуть бути вказані в схемах розширення та перевірені за допомогою зовнішніх інструментів.

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						42
Змн.	Арк.	№ докум.	Підпис	Дата		

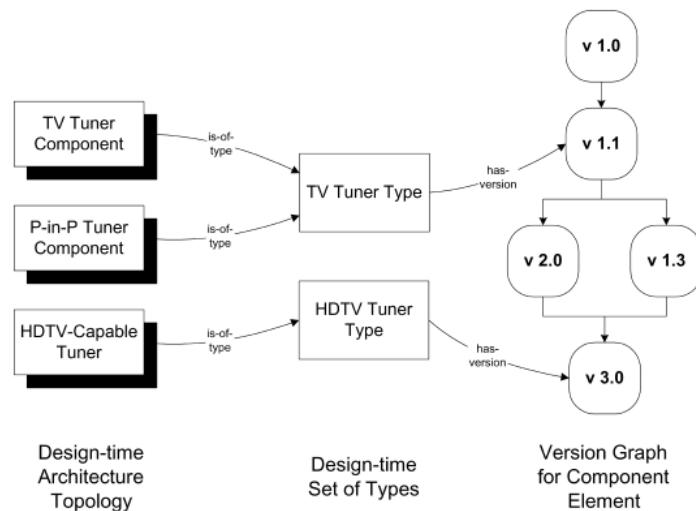


Рисунок 2.7 - Зв'язки між бетонними елементами, їх типами та графами версій.

Опції. Схема опцій дозволяє позначати компоненти часу розробки, з'єднувачі та посилання як додаткові в архітектурі. Додаткові елементи супроводжуються «умовою захисту», формат якої можна вказати в розширенні. xADL 2.0 надає схему за замовчуванням для захистів, схему логічного ЗАХИСТУ, яка дозволяє визначати захисти як логічні вирази, подібні до тих, що зустрічаються в сучасних мовах програмування. Якщо умова захисту задовольняється під час оцінювання, то додаткові елементи включаються до архітектури; інакше вони виключаються.

У нашому прикладі з телевізором ми можемо перетворити нашу архітектуру з одним продуктом на лінійку продуктів, додавши опціональність. Зробивши тюнер «картинка в картинці» опціональним разом з його посиланням на телевізійний роз'єм, ми можемо створити лінійку продуктів, яка описує два продукти: телевізор з функцією «картинка в картинці» та телевізор без неї. Зміни, які ми повинні внести до нашого існуючого опису продукту, показано на рисунку 2.8. Тут ми додаємо елемент <optional> як до тюнера «картинка в картинці», так і до його посилання на роз'єм. Охоронна умова для обох елементів <optional>: hasP_in_P == true, тому ці елементи будуть включені до архітектури лише в тому випадку, якщо цільовий продукт має функцію «картинка в картинці» (тобто змінна hasP_in_P пов'язана зі значенням true).

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						43
Змн.	Арк.	№ докум.	Підпис	Дата		

```

<xArch>
  <archStructure>
    ...
    <component id="pinctuner">
      <description>Pic in Pic Tuner
      </description>
      ...
      <optional>
        <guard>
          <booleanExp>
            <equals>
              <symbol>hasP_in_P</symbol>
              <value>true</value>
            </equals>
          </booleanExp>
        </guard>
      </optional>
    </component>
    <!--P-in-P tuner-to-connector link
    is also optional with same guard
    condition-->
  </archStructure>
</xArch>

```

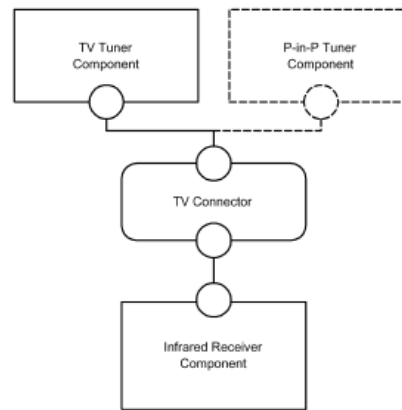


Рисунок 2.8 - Діаграма лінійки телевізійних продуктів, де елементи «картинка в картинці» є необов'язковими, з описом xADL 2.0.

Зверніть увагу, що можливо виразити архітектури з використанням захистів, які можуть призвести до неповних або неправильних архітектур (наприклад, якщо захист посилення у прикладі вище були відсутні або відрізнялися від захисту на тюнері P-in-P). Коли лінійка продуктів буде скорочена до одного продукту за допомогою нашого інструмента вибору, ці невідповідності стануть очевидними, і багато з них (наприклад, зависла ланка) будуть помічені нашими дизайнерами.

Варіанти. Схема «Варіанти» дозволяє змінювати типи певних - конструкцій часу проектування в архітектурі. Зокрема, вона визначає варіантні типи компонентів та роз'ємів. Варіантні типи містять набір можливих альтернатив. Кожна альтернатива - це тип компонента або роз'єму, що супроводжується умовою захисту, подібною до тієї, що використовується в схемі «ОПЦІЇ». Захисні умови для варіантів мають бути взаємовиключними. Коли виконується умова захисту, замість типу варіанта використовується пов'язаний з ним тип компонента або роз'єму.

Давайте ще більше диверсифікуємо нашу лінійку телевізійних продуктів, розширивши її на кілька міжнародних ринків. Телевізори, що постачаються до Північної Америки чи Японії, потребуватимуть тюнерів NTSC, тоді як телевізори, що постачаються до більшої частини Європи, потребуватимуть

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						44
Змн.	Арк.	№ докум.	Підпис	Дата		

тюнерів PAL. Ми можемо виразити це, замінивши існуючий тип тюнера варіантним типом, який пов'язаний з двома можливими конкретними типами: типом тюнера NTSC та типом тюнера PAL. У цьому прикладі структурний опис архітектури взагалі не змінюється: як основний, так і тюнери "картинка в картинці" зберігають посилання на один і той самий тип (tuner_type). Однак тепер їхній тип став варіантним: вони можуть бути тюнерами NTSC або PAL, залежно від ситуації.

2.4 Висновки по розділу

У другому розділі розглянуто методи та інструменти проєктування модульних програмних систем. Проаналізовано основні архітектурні підходи, що лежать в основі побудови модульних рішень, а також методи декомпозиції системи на незалежні компоненти. Розглянуто підходи до моделювання архітектурної еволюції та розвитку лінійок програмних продуктів. У результаті сформовано системне уявлення про методологію проєктування модульної архітектури.

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						45
Змн.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ МОДУЛЬНИХ ПРОГРАМНИХ СИСТЕМ

3.1 Інструментальні засоби підтримки модульної архітектури

Корисність ADL тісно пов'язана з обсягом інструментальної підтримки, доступної для цієї ADL. Інструменти необхідні для створення описів архітектури, їх редагування, аналізу, зіставлення з реалізаціями системи тощо. Інструменти можуть також допомагати захистити архітектора від неприємних синтаксичних деталей (таких як розмітка XML та простори імен).

Ми розділяємо інструменти, доступні в нашій інфраструктурі, на дві групи: інструменти, засновані на синтаксисі, та інструменти, засновані на семантиці. Інструменти, засновані на синтаксисі, є універсальними, а їхні - інтерфейси (графічні, текстові або програмні) базуються на синтаксисі ADL, як зазначено в XML-схемах [37]. Оскільки ADL, створені в нашій інфраструктурі, є розширюваними та легко змінюваними, інструменти, засновані на синтаксисі, є важливими, оскільки вони адаптуються в міру розвитку та змін мови. Інструменти, засновані на семантиці, роблять деякі припущення щодо значень конкретних конструкцій у побудованих ADL та можуть вимагати або підтримувати використання певних схем. Ці інструменти надають додаткову цінність користувачам нашої інфраструктури, які застосовують деякі або всі схеми xADL 2.0 та для яких семантичні припущення є дійсними. Зв'язки між інструментами в нашій інфраструктурі зображено на рисунку 10. Кожен інструмент у нашій інфраструктурі описано в цьому розділі.

Однією з переваг використання XML є підтримка готових синтаксичних інструментів для маніпулювання документами та схемами. Крім того, ми використали ці готові інструменти для створення додаткових синтаксичних інструментів, специфічних для нашої інфраструктури.

Готові інструменти для роботи з XML: Sun Crimson Parser, XSV та XML Spy. Оскільки ADL, створені в нашій інфраструктурі, визначені в XML-схемах, а

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						46
Змн.	Арк.	№ докум.	Підпис	Дата		

описи архітектури – це XML-документи, що відповідають цим схемам, користувачам доступно багато готових інструментів для роботи з XML загалом. Три такі інструменти особливо важливі, оскільки вони формують основу для всіх наших інших інструментів.

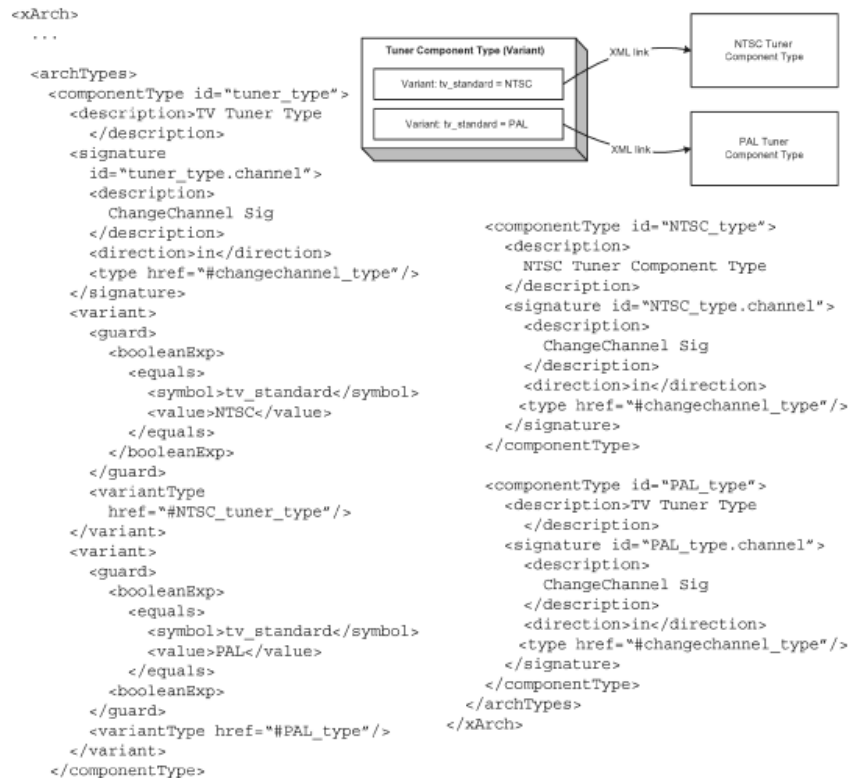


Рисунок 3.1 - Розширення раніше конкретного типу (tuner_type) до варіантного типу.

Парсер Sun Crimson - це реалізація на Java API DOM та SAX для розбору та маніпулювання XML-документами. Crimson надає можливість програмного доступу до структур у XML-документах (як описів архітектури, так і схем) на рівні синтаксичних конструкцій XML (елементів, тегів, атрибутів, коментарів тощо). Цей API формує основу для потужніших інструментів на основі синтаксису, які будуть описані далі в цьому розділі.

XSV - це валідатор XML-схем з відкритим кодом та документів-екземплярів. Він виконує дві основні функції: перевірку синтаксичної правильності XML-схем, що є частиною ADL, та перевірку відповідності опису

архітектури набору схем. Хоча XSV не може перевіряти складні властивості архітектури, такі як узгодженість типів, він може забезпечити деяку рудиментарну синтаксичну перевірку узгодженості на рівні XML-конструкцій.

XML Spy від Altova – це комерційне середовище розробки для XML. XML Spy можна використовувати для проектування та перевірки нових схем. Найголовніше, що він надає спосіб графічного дослідження та проектування конструкцій схем, що є цінним інструментом для розробників схем для перевірки та передачі функцій моделювання у своїх схемах без необхідності читати необроблені схеми.

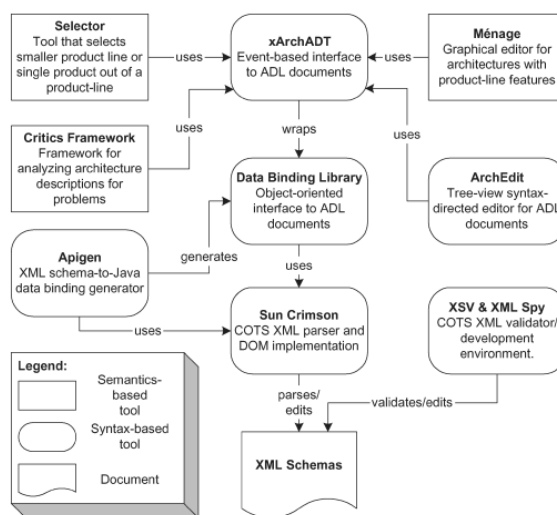


Рисунок 3.2 - Взаємозв'язки між інструментами в нашій інфраструктурі.

Повторне використання готових інструментів XML є ледь помітною, але важливою перевагою нашого підходу. Хоча ці інструменти не працюють на рівні архітектурних конструкцій (наприклад, компонентів та конекторів), вони дозволяють нам уникнути буденних завдань, таких як створення парсерів, серіалізаторів, перевірок синтаксису, редакторів графічних схем тощо. Наш підхід також автоматично виграє від удосконалень цих технологій: покращення продуктивності в базовій реалізації DOM призводить до швидшої роботи застосунків.

Бібліотека зв'язування даних. Готові інструменти XML, як правило, підтримують маніпулювання документами з точки зору низькорівневих

концепцій XML, таких як елементи, атрибути та текстові сегменти [38]. Створення інструментів, які працюють безпосередньо з цими низькорівневими конструкціями, є громіздким та схильним до помилок, оскільки це вимагає від самих інструментів безпосереднього керування елементами, такими як простори імен, та забезпечення відповідності документів схемам XML.

Розглянемо наступне XML-визначення компонента, взяте зі схеми структури та типів xADL 2.0 :

Лістинг 3.1 - XML-визначення компонента

```
<complexType name="Component">
  <sequence>
    <element name="description" type="Description"/>
    <element name="interface" type="Interface"
      minOccurs="0" maxOccurs="unbounded"/>
    <element name="type" type="XMLLink"
      minOccurs="0" maxOccurs="1"/>
  </sequence>
```

Коли схеми доступні, можна створити більш зручний програмний інтерфейс для XML-документів на основі мови, що визначається схемами. У нашій інфраструктурі цей інтерфейс надається через бібліотеку зв'язування даних [39]. Наша бібліотека зв'язування даних надає набір класів Java, що відповідають елементам та атрибутам, зазначеним у схемах xADL 2.0. Ці класи приховують деталі XML, такі як простори імен, теги заголовків, порядок послідовностей тощо. У той час як загальний XML API, такий як DOM, надає такі функції, як `addElement(...)` та `getChildElements(...)`, класи в нашій бібліотеці зв'язування даних надають такі функції, як `addComponentInstance(...)` та `getAllInterfaces(...)`. Внутрішньо бібліотека використовує реалізацію DOM, що постачається з `Crimson`, для маніпулювання базовим XML-документом. Бібліотека зв'язування даних автоматично генерується іншим інструментом у нашій інфраструктурі, `Arigen`.

Для цього типу бібліотека зв'язування даних містить клас Java, який надає наступний інтерфейс:

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						49
Змн.	Арк.	№ докум.	Підпис	Дата		

Лістинг 3.2 - Бібліотека зв'язування даних

```
void setDescription(IDescription value);
void clearDescription();
IDescription getDescription();
void addInterface(IInterface newInterface);
void addInterfaces(Collection interfaces);
void clearInterfaces();
IInterface getInterface(String id);
Collection getInterfaces(Collection ids);
Collection getAllInterfaces();
void removeInterface(IInterface interface);
void setType(IXMLLink link);
void clearType();
IXMLLink getType();
void setId(String id);
String getId();
void clearId();
```

Це демонструє, що, незважаючи на відсутність знань про семантику ADL, бібліотека зв'язування даних надає функції, які ближчі (за рівнем абстракції) до концепцій, що стосуються архітектора програмного забезпечення. Це робить створення інструментів архітектури за допомогою бібліотеки зв'язування даних більш інтуїтивним, ніж їх створення за допомогою інструменту XML, такого як Crimson. Крім того, це значно зменшує кількість рядків коду, необхідних для маніпулювання описом архітектури на основі XML, приблизно у співвідношенні 5:1. Тобто, кожен виклик бібліотеки зв'язування даних (наприклад, `addInterface`) інкапсулює близько 5 рядків коду DOM.

Арigen. Якщо бібліотеку зв'язування даних потрібно переписувати щоразу, коли схема додається, змінюється або видаляється з ADL, то перевага її наявності зводиться нанівець. На щастя, синтаксичної інформації, присутньої в схемах ADL, достатньо для автоматичної генерації бібліотеки зв'язування даних. Ми створили інструмент під назвою «Арigen» (скорочення від API generator), який може автоматично генерувати бібліотеку зв'язування даних Java, описану раніше, для xADL 2.0 та схем розширення. Коли схеми ADL змінюються, розробники інструментів просто повторно запускають Арigen для зміненого набору схем, щоб згенерувати нову бібліотеку зв'язування даних. Звичайно, зв'язування для елементів, які не змінилися, залишаються незмінними, що мінімізує вплив на існуючі інструменти, які використовують бібліотеку. Через складність мови схем XML, Арigen не є універсальним генератором зв'язування

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						50
Змн.	Арк.	№ докум.	Підпис	Дата		

даних, він не підтримує повний спектр конструкцій, доступних у схемах XML (наскільки нам відомо, жодне зв'язування схеми XML (генератор наразі це робить). Однак він підтримує великий набір функцій схеми і досі був достатнім для створення прив'язок даних для всіх схем xADL 2.0, а також схем, написаних третіми сторонами [40]. Повний список підтримуваних і непідтримуваних конструкцій в Arigen виходить за рамки цієї роботи, але включено до документації інструменту Arigen.

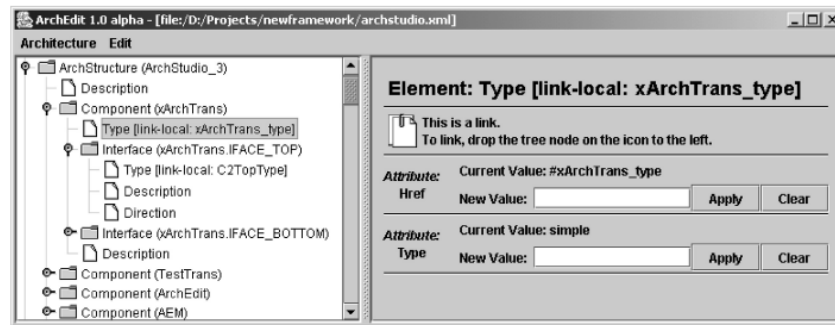
Data Binding Library Call:	xArchADT Message:
<pre>IComponent component; IInterface interface; ... component.addInterface(interface);</pre>	<pre>ObjRef componentRef; ObjRef interfaceRef; message{ name = "add"; source = componentRef; type = "interface"; target = interfaceRef; }</pre>

Рисунок 3.3 - Приклад виклику бібліотеки зв'язування даних та еквівалентне повідомлення xArchADT.

xArchADT. Бібліотека зв'язування даних надає традиційний об'єктно-орієнтований інтерфейс для редагування описів архітектури. Це вимагає від викликаючих програм бібліотеки підтримки багатьох прямих посилань на об'єкти. Загалом, розподілені та подієві системи припускають, що компоненти не мають спільного адресного простору і, отже, не можуть підтримувати посилання на об'єкти між компонентами. Через це використання такої бібліотеки як незалежного компонента в розподіленій або подієвій системі є складним. Щоб вирішити цю проблему, ми створили обгортку під назвою «xArchADT» для бібліотеки зв'язування даних, яка надає подієво-орієнтований інтерфейс замість об'єктно-орієнтованого. Замість викликів процедур, доступ до xArchADT здійснюється через асинхронні події. Приклад того, як виклик бібліотеки зв'язування даних виражається як структура даних подій, показано на рисунку 11. Вона використовує першокласні непрямі посилання на об'єкти, а не прямі вказівники, для посилання на елементи в документах xADL 2.0. Тобто,

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						51
Змн.	Арк.	№ докум.	Підпис	Дата		

xArchADT призначає ідентифікатори елементам xADL 2.0, і ці ідентифікатори використовуються для посилання на елементи. Коли опис базової архітектури змінюється одним інструментом, xArchADT генерує подію, яка інформує всі інструменти прослуховування про зміну. Це надає бібліотеці зв'язування даних додаткову властивість слабкого зв'язку.



xArchADT, як і Arigen та бібліотека зв'язування даних, є рефлексивним. Він використовує вбудовані можливості рефлексії Java для автоматичної адаптації до змін у бібліотеці зв'язування даних. Тобто, якщо бібліотека буде регенерована Arigen, xArchADT працюватиме без змін. Однак найбільшим внеском xArchADT у наш набір інструментів є розширення діапазону контекстів, у яких можна використовувати бібліотеку зв'язування даних. Сама по собі бібліотека добре підходить для використання в розробці тісно пов'язаних об'єктно-орієнтованих систем, але обгортка xArchADT надає їй інтерфейс, придатний для віддаленого доступу через межі процесів (що забезпечується проміжним програмним забезпеченням) та включення в середовище на основі подій.

ArchEdit. Бібліотека зв'язування даних та xArchADT надають різні - програмні інтерфейси для маніпулювання описами архітектури. Наш Інфраструктура також включає синтаксичний інструмент із графічним інтерфейсом користувача під назвою «ArchEdit». Знімок екрана цього інструменту показано на рисунку 3.4.

ArchEdit графічно зображує опис архітектури у форматі дерева, де кожен

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						52
Змн.	Арк.	№ докум.	Підпис	Дата		

вузол можна розгортати, згортати або редагувати. Це схоже на багато візуальних XML-редакторів, за винятком того, що ArchEdit приховує XML-деталі документа від користувача. XML-схеми ADL керують структурою відображеного деревоподібного подання, роблячи структуру XML-документа та структуру відображеного дерева ідентичними. Це надає архітекторам прямий доступ до описів архітектури без абстрагування деталей. ArchEdit - це програмний компонент, що працює на основі подій, і отримує доступ до описів архітектури через xArchADT. Зміни до опису архітектури, внесені через xArchADT за допомогою ArchEdit або інших інструментів, негайно відображаються в інтерфейсі користувача ArchEdit .

ArchEdit - це ще один рефлексивний інструмент у нашій інфраструктурі. ArchEdit керується синтаксисом: він не розуміє семантику відображених елементів. Він динамічно будує своє представлення та інтерфейс на основі XML-схем, що використовуються для визначення ADL. Тому його не потрібно змінювати, коли схеми додаються, змінюються або видаляються. Ця гнучкість цінна, оскільки вона надає архітекторам простий графічний редактор для документів ADL автоматично, навіть якщо нові функції ADL були додані нещодавно.

Рефлексивність ArchEdit має два недоліки. По-перше, він не застосовує стилістичні обмеження чи інші правила до опису архітектури, які неможливо вказати в XML. По-друге, ArchEdit не може відобразити структуру архітектури програмного забезпечення інтуїтивно зрозумілим чином, наприклад, у вигляді діаграми з рамкою та стрілками. Ці недоліки притаманні будь-якому інструменту на основі синтаксису, але переваги наявності безкоштовного редактора для нових функцій ADL переважають ці недоліки. Для доповнення ArchEdit можна створити більш семантично усвідомлені редактори та інтегрувати їх за допомогою xArchADT, як ми покажемо в наступному розділі.

Інструменти на основі семантики

Інструменти на основі семантики відрізняються від інструментів на основі синтаксису тим, що вони несуть у собі певне уявлення про семантику

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						53
Змн.	Арк.	№ докум.	Підпис	Дата		

базової нотації. Це обов'язково означає, що використання певних інструментів на основі семантики буде залежати від включення певних специфічних XML-схем до їхніх цільових ADL, а також від деяких припущень щодо семантики, пов'язаної з цими схемами. На відміну від інструментів на основі синтаксису, інструменти на основі семантики не обов'язково можуть автоматично оновлювати свої інтерфейси та поведінку для підтримки нових схем. Це робить розширюваність першочерговим завданням під час розробки нових інструментів на основі семантики. Тому ми прагнули зробити наші існуючі інструменти максимально толерантними до нових схем, наприклад, ігноруючи невідому інформацію, а не даючи збоїв.

У цьому підрозділі ми розглянемо три зрілі семантичні інструменти, які ми розробили для нашої інфраструктури: фреймворк critics, який використовується для розробки та інтеграції компонентів аналізу; Menage, графічний редактор для описів архітектури; та Selector, інструмент для моделювання лінійок продуктів до менших лінійок продуктів або окремих продуктів. Крім того, ми надамо короткий огляд в кінці розділу поточних, незавершених проектів, які ми здійснюємо для створення більшої кількості семантичних інструментів.

Структура критиків. Інструменти перевірки узгодженості та аналізу є важливими частинами будь-якого набору інструментів ADL. Фактично, багато інших ADL були створені спеціально для експериментування з новими методами аналізу. У розширюваній ADL один інструмент аналізу є непрактичним. Наявність модульної структури критиків є кращою альтернативою. Критики - це програмні компоненти, які перевіряють властивості описів архітектури, виявляють помилки та потенційні недоліки, а також повідомляють про них іншим компонентам. Критики відрізняються від традиційних інструментів аналізу тим, що вони можуть оновлювати свій аналіз у міру змін документа, що дозволяє постійно проводити повторну перевірку. Архітектура структури критиків, виражена в архітектурному стилі, показана на рисунку 3.5.

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						54
Змн.	Арк.	№ докум.	Підпис	Дата		

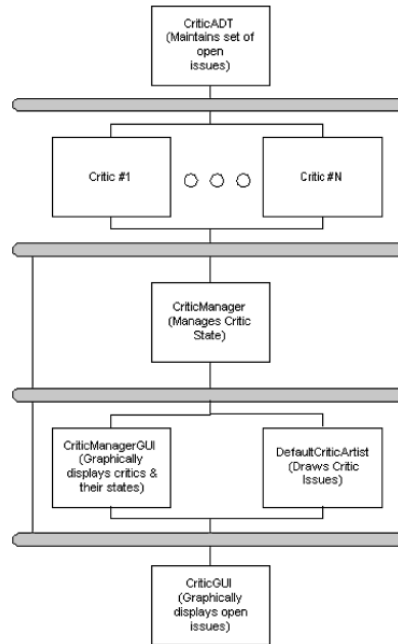


Рисунок 3.5 - Архітектура фреймворку critics.

У стилі C2 компоненти можуть робити запити до компонентів, що знаходяться вище них, але не можуть робити припущення щодо компонентів, що знаходяться нижче них. Таким чином, події, що запитують обслуговування, надсилаються вгору, а зміни стану - вниз. Наприклад, щоразу, коли внутрішнє сховище даних CriticADT щодо відкритих задач змінюється, він надсилає сповіщення всім підпорядкованим йому компонентам.

Критики дизайну – це незалежні компоненти, які можна включити до фреймворку та активувати, якщо їхній аналіз стосується базової ADL. Критики в нашому фреймворку або виконують детальний аналіз, перевіряючи лише одну властивість або набір пов'язаних властивостей, або спираються на інші критики для виконання складніших аналізів. Наприклад, *критик посилань* перевіряє, чи всі кінцеві точки посилань прив'язані до інтерфейсів, тоді як критик *менеджера еволюції архітектури* використовує *критика посилань* (та інші), щоб вирішити, чи є опис архітектури достатньо повним для створення екземпляра зазначеної системи.

CriticManager дозволяє користувачам вибирати, яких критиків активувати та деактивувати, а кілька компонентів художника та графічного інтерфейсу відображають відкриті проблеми для ознайомлення кінцевими користувачами.

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						55
Змн.	Арк.	№ докум.	Підпис	Дата		

Користувачі можуть використовувати елементи графічного інтерфейсу (у нашій поточній системі, кнопку «Focus Open Editors»), щоб повідомити редакторів відкритої архітектури про необхідність зосередитися на певному(их) елементі(ах), що стосуються проблеми. Знімки екрана зразка.

Вихідні дані цих критиків та їх візуалізація в менеджері критиків показані на рисунку 3.6.

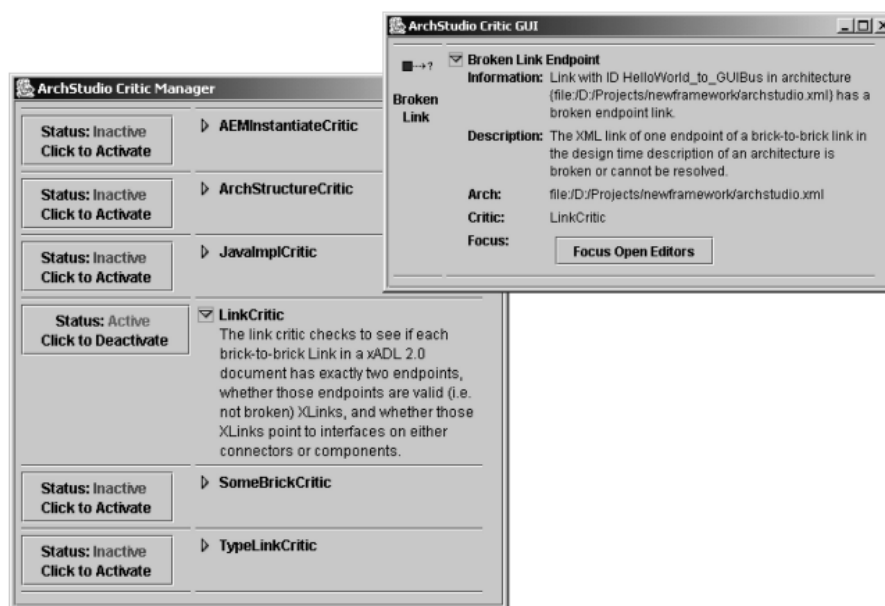


Рисунок 3.6 - Знімки екрана менеджера критиків та проблем із критиками.

Наш фреймворк включає зростаючий набір критичних елементів для базової перевірки узгодженості документів xADL 2.0. Поточні критичні елементи в нашому фреймворку перелічені в Таблиці 3.1. Конкретний набір критичних елементів, який ми створили на сьогодні, перевіряє деякі з найпоширеніших помилок, які можуть виникнути під час визначення архітектури xADL 2.0. Хоча досі ми зосереджувалися на створенні фреймворку, ми маємо намір використати це як відправну точку для майбутнього дослідження методів побудови та написання критичних елементів, наприклад, використання готових інструментів аналізу на основі XML, таких як xlinkit.

Menage. Menage – це середовище графічного дизайну для архітектур одного продукту та лінійки продуктів. Menage вимагає, щоб його цільова ADL

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						56
Змн.	Арк.	№ докум.	Підпис	Дата		

включала СХЕМИ СТРУКТУРИ та ТИПІВ , версій , опцій , варіантів ТА логічного ЗАХИСТУ з xADL 2.0. Menage дозволяє візуалізувати архітектури та виражати їх у графічній структурі. Додаткові та варіантні конструкції зображуються чітко. Додаткові конструкції мають пунктирну рамку. Варіантні компоненти та з'єднувачі позначені як варіанти ;

Таблиця 3.1

Критики та їхні описи

Критик	Опис
ArchStructure Critic	Перевіряє валідність елементів структури архітектури (ArchStructure).
Java Implementation Critic	Перевіряє правильність анотацій Java-реалізації для типів компонентів, конекторів та інтерфейсів.
Link Critic	Перевіряє валідність архітектурних зв'язків — наявність двох коректних кінцевих точок тощо.
Type Link Critic	Гарантує, що кожен компонент, конектор та інтерфейс має валідний зв'язок із відповідним типом.
XML Link Critic	Перевіряє валідність XML-посилань, знайдених в описах архітектури.
Version Graph Critic	Перевіряє, чи всі типи компонентів, конекторів та інтерфейсів мають коректні посилання на вузол у відповідному графі версій.
Signature Critic	Перевіряє, чи мають компоненти та конектори правильну кількість і типи інтерфейсів згідно з їхніми сигнатурами.
Signature-Interface Mapping Critic	Перевіряє, чи правильно компоненти та конектори з субархітектурами підключені до внутрішніх елементів.
Interface Direction Critic	Перевіряє сумісність напрямків підключених інтерфейсів (наприклад, «вхідні» підключаються тільки до «вихідних»).
Floating Elements Critic	Шукає «плаваючі» елементи — компоненти та конектори, що не підключені до решти архітектури.
Duplicates Critic	Шукає елементи з ідентифікаторами, що дублюються.
Architecture Evolution Manager Critic	Визначає, чи містить опис архітектури достатньо інформації для створення її екземпляра (інстанціації).

Подвійне клацання на варіантному елементі показує набір можливих варіантів. Кожен варіант або додатковий елемент пов'язаний з редагованою логічною умовою захисту, щоб визначити, чи включено його до архітектури чи ні.

Menage добре працює в поєднанні з ArchEdit та нашими іншими інструментами, орієнтованими на синтаксис. Описи архітектури, створені в Menage, можна за потреби точно налаштувати в ArchEdit, або ж ArchEdit можна використовувати для доступу до елементів схеми, які не підтримуються

безпосередньо Menage.

Селектор архітектури. Архітектура лінійки продуктів являє собою набір можливих архітектур продуктів. Оскільки параметри проектування стають фіксованими, кількість точок варіації в архітектурі лінійки продуктів зменшується. Коли всі точки варіації видаляються, результатом є єдиний продукт. Наша інфраструктура включає інструмент під назвою «Селектор», який дозволяє автоматично виконувати скорочення лінійки продуктів для описів архітектури лінійки продуктів, що використовують схеми « СТРУКТУРА та типи » , « Опції » , « ВАРІАНТИ » та « ЛОГІЧНИЙ захист » .

Нагадаємо, що кожен необов'язковий або варіантний елемент супроводжується логічним захисним виразом, який визначає, коли він включений в архітектуру.

Булевий вираз містить змінні, так само як і булевий вираз у мові програмування, але ці змінні не пов'язані зі значеннями в описі архітектури. Інструмент «Селектор» надає користувачам графічний інтерфейс користувача, в якому вони можуть вказати набір значень, які будуть пов'язані зі змінними в булевих виразах в архітектурі лінійки продуктів. Потім користувач натискає кнопку, щоб розпочати процес вибору. Селектор оцінює всі булевий захист для необов'язкових та варіантних елементів в архітектурі, використовуючи значення змінних, надані користувачем. Для виразів, які можна вирішити, необов'язкові компоненти або робляться постійними (включаються) , або видаляються (виключаються), а вибрані варіанти робляться постійними. Для виразів, які неможливо повністю вирішити (наприклад, через незв'язану змінну), вираз максимально скорочується, а елемент залишається необов'язковим або варіантним. Коли всі опції та варіанти вирішено, результатом є єдина архітектура продукту. Якщо деякі опції та варіанти залишаються, результатом є підмножина оригінальної лінійки продуктів. Знімок екрана Menage показано на рисунку 3.7.

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						58
Змн.	Арк.	№ докум.	Підпис	Дата		

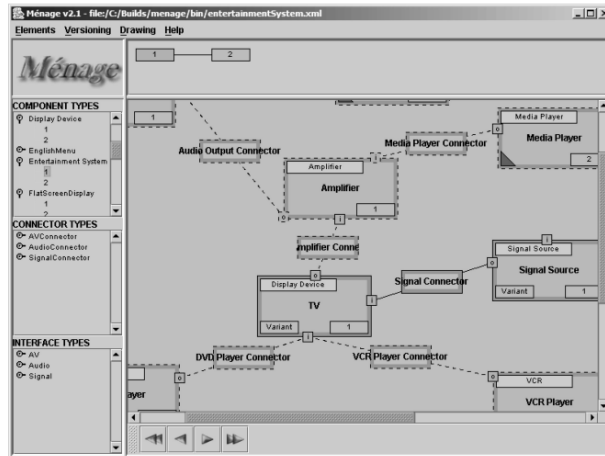


Рисунок 3.7 - Знімок екрана управління.

Додаткові проекти. Окрім перелічених у цьому розділі, ми постійно розробляємо нові семантичні інструменти для розширення нашої структури та забезпечити більш повну підтримку схем xADL 2.0. Ці проекти перебувають на різних стадіях розробки, і більше інформації про всі з них можна знайти на наших вебсайтах. Нижче наведено частковий список:

Розрізнення та об'єднання архітектур. Інструменти для автоматичної генерації документів, що описують різницю між двома архітектурами, та об'єднання документів про відмінності з існуючими архітектурами. Це корисно для управління еволюцією та поширення ознак.

Диференціація та об'єднання лінійки продуктів. Застосування методів диференціації та об'єднання до членів лінійки продуктів, що дозволяє диференціювати окремі члени лінійки продуктів та поширювати характеристики від одного продукту-члена до іншого.

Інтеграція з Eclipse. Зусилля щодо забезпечення та підтримки зіставлень з xADL компоненти до їх реалізації у вихідному коді в середовищі розробки Java Eclipse.

Archipelago. Графічний редактор xADL 2.0 другого покоління, що зосереджений на підтримці дослідницького дизайну та нових метафор користувацького інтерфейсу для архітектурного проектування. Archipelago побудовано на високогнучкому фреймворку плагінів, що робить його добре

поєднуваним з розширюваною мовною інфраструктурою.

Type Wrangler. Інструмент, який допомагає користувачам переконатися, що компоненти та роз'єми правильно зіставлені з їхніми типами (наприклад, інтерфейси відповідають сигнатурам, типи інтерфейсів узгоджені тощо).

3.2 Реалізація та впровадження модульної програмної системи

Наша інфраструктура застосовувалася для вирішення проблем у низці областей. У цьому розділі ми виділимо чотири приклади, кожен з яких демонструє різні сильні сторони інфраструктури. По-перше, ми показуємо, як наша інфраструктура підтримувала моделювання та симуляцію архітектури великої військової системи, демонструючи масштабованість інфраструктури. По-друге, ми показуємо, як наша інфраструктура підтримувала розробку ADL, яка зараз використовується в експериментах з архітектурного моделювання для систем космічних апаратів, демонструючи адаптивність інфраструктури до нових архітектурних областей з унікальними вимогами моделювання. По-третє, ми показуємо, як наша інфраструктура підтримувала розробку ADL для Koala та Mae, двох представлень, що використовуються для моделювання архітектур лінійки продуктів, демонструючи розширюваність XML-схем та інструментів у нашій інфраструктурі. Нарешті, ми показуємо, як інфраструктура використовувалася для розробки власного середовища розробки програмного забезпечення, демонструючи її здатність описувати та підтримувати себе. Два з цих прикладів демонструють, як xADL. Схеми вже можуть самостійно підтримувати розробку на основі архітектури, і дві з них описують досвід створення нової ADL в нашій інфраструктурі з використанням наших базових схем та інструментів. Загалом, цей досвід демонструє, як різні аспекти нашої інфраструктури (розширюваність на основі XML, базовий набір схем та гнучка підтримка інструментів) сприяють її ефективному використанню.

Літак системи повітряного попередження та управління (AWACS) США підтримується великою розподіленою програмною архітектурою для передачі

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						60
Змн.	Арк.	№ докум.	Підпис	Дата		

повідомлень. Однак формальної архітектурної моделі цього програмного забезпечення не існує. Щоб виправити це та допомогти оцінити масштабованість нашої інфраструктури, ми змодельовали архітектуру програмних систем літака AWACS у xADL 2.0 та використали цю модель як основу для архітектурного моделювання AWACS. Модель AWACS була створена на основі доступної документації та власного симулятора поведінки компонентів AWACS, наданого нам промисловим партнером.

Опис AWACS складається з понад 10 000 рядків (майже один мегабайт) XML. Опис AWACS описує 125 компонентів та 206 роз'ємів, розподілених по 28 процесорах, а також типи компонентів, роз'ємів та інтерфейсів. Опис було перевірено на відповідність схемам xADL 2.0 за допомогою XSV та візуалізовано за допомогою ArchEdit. Ми використовували ArchEdit для перевірки архітектури та внесення подальших удосконалень, доки модель не стала точною, та провели критичну роботу щодо архітектури, щоб перевірити різні її аспекти.

Різні інструменти, які ми використовували, були відносно ефективними, враховуючи великий розмір AWACS. XSV може проаналізувати XML-документ розміру AWACS за кілька секунд. ArchEdit амортизує час, необхідний для візуалізації архітектури, зчитуючи дані з дерева DOM лише під час його перегляду; розгортання найбільш заповненого вузла в описі AWACS вперше займає близько 5-8 секунд на сучасному ПК. Продуктивність критиків значною мірою залежала від того, скільки даних критикам потрібно було проаналізувати. Критики, які перевіряють значну частину Перевірка архітектури (наприклад, Link Critic) на ПК займала близько 2 секунд. Ми вважаємо, що цей час є прийнятним, враховуючи розмір архітектури, що перевіряється.

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						61
Змн.	Арк.	№ докум.	Підпис	Дата		

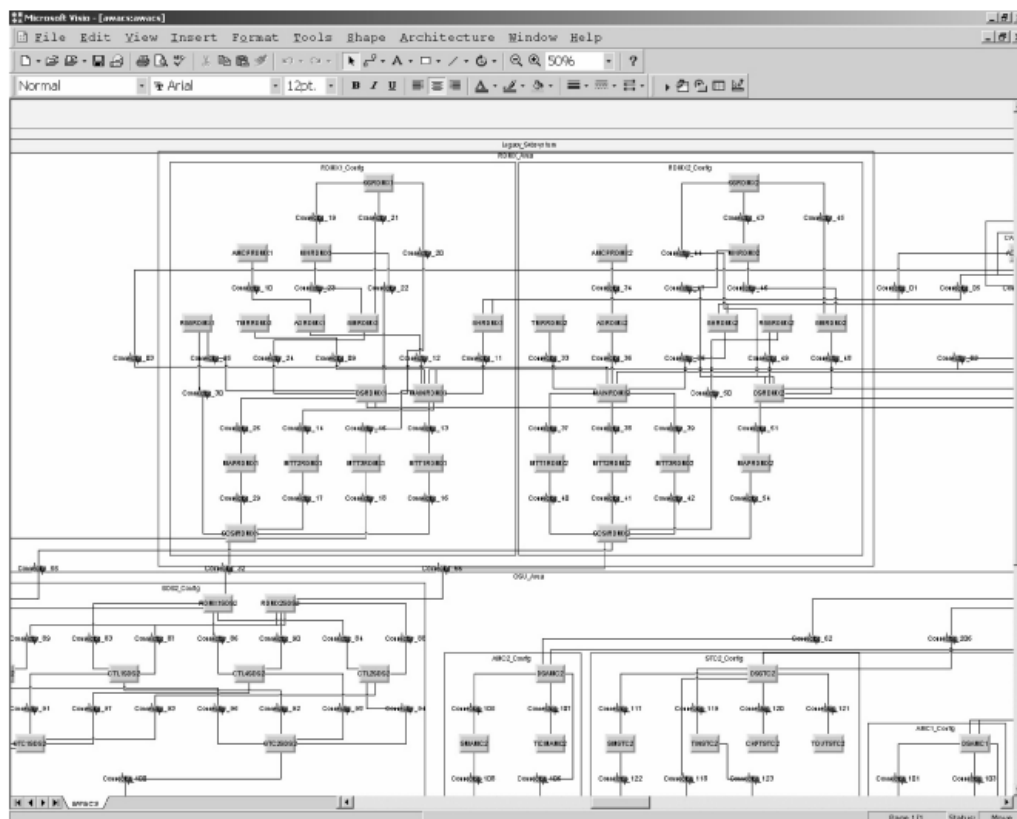


Рисунок 3.8 - Знімок екрана симулятора AWACS.

Поряд із початковим описом архітектури, ми розробили симуляцію зв'язку між компонентами літака на основі архітектури. Симулятор складається з реалізацій кожного типу компонента та роз'єму в архітектурі на Java. До опису архітектури було додано зіставлення реалізацій між типами в описі архітектури та цими класами Java. Програма початкового завантаження зчитує опис архітектури в xArchADT та автоматично створює екземпляри та пов'язує компоненти та роз'єми. Для візуалізації симуляції в окремому проєкті було створено розширення для Microsoft Visio, яке може відображати архітектуру xADL 2.0 як граф компонентів, роз'ємів та зв'язків. Повідомлення, що надсилаються між компонентами та роз'ємами AWACS, анімовані на цьому графіку. Знімок екрана цього симулятора показано на рисунку 3.8.

Окрім масштабованості, наш досвід роботи з AWACS демонструє кілька додаткових позитивних характеристик нашої інфраструктури. По-перше, це показує, що наші базові схеми мають ефективні можливості моделювання самі по собі, оскільки для моделювання архітектури AWACS не було потрібно

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
Змн.	Арк.	№ докум.	Підпис	Дата		62

жодних розширень. По-друге, це показує гнучкість інструментальної підтримки інфраструктури, оскільки ми змогли використовувати xArchADT (і, як наслідок, нашу бібліотеку зв'язування даних) для створення опису архітектури та використання її як основи для нашого симулятора. Нарешті, це показує цінність COTS та інструментів на основі користувацького інтерфейсу в нашій інфраструктурі, оскільки ми змогли використовувати XSV для перевірки нашого опису та ArchEdit для його уточнення.

Як демонстрацію адаптивності інфраструктури до нової області з її унікальними архітектурними вимогами, ми описуємо наш досвід роботи з Лабораторією реактивного руху NASA (JPL). JPL розробляє програмні системи для космічних зондів та наземних систем. Ця область породжує унікальні потреби моделювання; зокрема, ці архітектури моделюються як системи на основі станів. Щоб врахувати ці обмеження моделювання, група систем даних місії JPL та їхні дослідницькі партнери з Університету Південної Каліфорнії (USC) створили розширення для xADL 2.0. Хоча конкретний вміст цих схем не може бути поширений через їх власницьку та конфіденційну природу, вони приблизно характеризуються наступним чином:

Схема статичної поведінки. Ця схема розширює основні схеми xADL 2.0 для фіксації статичних поведінкових властивостей системи. Ці моделі поведінки додають попередні умови, постумови та інваріанти до типів компонентів xADL 2.0, використовуючи набір змінних, також визначених у розширенні. Крім того, сигнатури xADL 2.0 розширені вхідними та вихідними параметрами, що дозволяє їм визначати функції, подібні до мов програмування.

Схема типів MDS. Ця схема розширює схему статичної поведінки для захоплення просторів імен та складної інформації про успадкування для компонентів.

Схема реалізації MDS. Ця схема пов'язує архітектурні артефакти з аналогами на рівні реалізації, як це виражено у власному фреймворку MDS JPL. Вона використовується переважно для генерації коду.

JPL інтегрувала свою розширену версію xADL 2.0 у процес розробки,

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						63
Змн.	Арк.	№ докум.	Підпис	Дата		

створивши транслятори до та з існуючих нотацій, таких як UML, та власні текстові нотації. Вони також застосували наші інструменти для візуалізації та маніпулювання описами архітектури. Крім того, дослідники з USC використовували розширену версію xADL 2.0 для моделювання архітектури SCrover, мобільного робота, заснованого на фреймворку MDS JPL. Вони модифікували існуючий інструмент аналізу архітектури DRADEL, щоб використовувати розширені JPL описи xADL, а не власні власні моделі, та використовували DRAD EL для аналізу архітектури SCrover. Аналіз DRADEL виявив 21 помилку в архітектурі SCrover, 6 з яких також не були виявлені в процесі експертної оцінки. Кінцевою метою цих зусиль є використання xADL 2.0 як частини масштабних зусиль щодо сприяння генерації коду на основі моделей. Крім того, успіх цих зусиль нещодавно спонукав JPL розпочати другий проект, використовуючи нашу інфраструктуру, цього разу зосереджуючись на моделюванні космічних місій та архітектурі систем.

Цей досвід підтверджує, що універсальність нашої інфраструктури та її механізми розширюваності корисні в раніше недосліджених областях, особливо стосовно базових схем xADL 2.0. JPL змогла повторно використовувати базові схеми xADL 2.0, створюючи відносно невеликі нові схеми для моделювання специфічних для предметної області деталей програмного забезпечення космічних апаратів. Використання JPL Arigen та пов'язаної з ним бібліотеки зв'язування даних для маніпулювання описами архітектури ще раз демонструє цінність інструментальної підтримки нашої інфраструктури. Нарешті, їхнє зіставлення описів архітектури з іншими представленнями показує адаптивність нашого підходу до інших, незапланованих ситуацій.

Щоб продемонструвати здатність нашої інфраструктури фіксувати концепції з нової дослідницької області та ефективно додавати інструментальну підтримку для цих концепцій, ми створили схеми, які додають унікальні моделі Koala та Mae до наших базових схем xADL 2.0. Koala та Mae – це два формати представлення для фіксації архітектур лінійки продуктів. Як описано, схеми xADL 2.0 вже забезпечують базову підтримку архітектур лінійки продуктів за

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						64
Змн.	Арк.	№ докум.	Підпис	Дата		

допомогою схем Versions , Options та Variants . Однак, як Koala, так і Mae мають унікальні характеристики моделювання, які відрізняються від характеристик у схемах xADL 2.0 та один від одного.

Існує кілька відмінностей між Koala та xADL 2.0. По-перше, Koala не підтримує поняття явних конекторів або версій. По-друге, Koala має дві конструкції, яких немає в xADL 2.0: інтерфейси різноманітності та комутатори. Інтерфейс різноманітності, що представляє точку варіації в архітектурі, повинен бути присутнім на варіантних компонентах і має бути "вихідним" інтерфейсом. Коммутатор - це явна конструкція, що застосовується в точці варіації, яка створює з'єднання з варіантним компонентом, який буде використовуватися.

Представлення Mae дещо ближче до xADL 2.0. Між xADL 2.0 та Mae існують дві ключові відмінності. По-перше, типи компонентів у Mae доповнені рядком, що описує їхній архітектурний стиль. По-друге, типи компонентів у Mae також мають відношення підтипу та посилання на їхній супертип. Ми усунули ці відмінності, розширивши визначення типу компонента за допомогою нової схеми.

Гнучкість xADL 2.0 на рівні окремих елементів була корисною кілька разів. Наприклад, Koala не має підтримки явних конекторів, тому ми просто - виключили конектори з нашого ADL, оскільки xADL 2.0 їх не вимагає, демонструючи модульність схем xADL 2.0 на рівні окремих елементів. Коли потрібні були нові конструкції, такі як інтерфейси різноманітності Koala та підтипи Mae, ми створювали прості схеми, які додавали ці сутності до xADL 2.0. Цей досвід ще раз демонструє ефективність механізму розширюваності на основі XML, який ми обрали для нашої інфраструктури.

Розглянемо схему, показану на рисунку 3.9, яка використовується для додавання інтерфейсів різноманітності в стилі Koala до xADL 2.0. Ця схема підтипує визначення інтерфейсу для створення інтерфейсу різноманітності та розширює визначення компонента для додавання такого інтерфейсу. Зверніть увагу на відносну простоту цієї схеми; інші схеми, повністю показані в, також є простими та зрозумілими.

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						65
Змн.	Арк.	№ докум.	Підпис	Дата		

Для зіставлень Koala та Mae ми успішно застосували весь спектр методів розширюваності на основі XML-схем (створення нових елементів, розширення існуючих елементів, обмеження елементів тощо).

```
<schema xmlns="diversity.xsd">
  <complexType name="DiversityInterface">
    <complexContent>
      <restriction base="Interface">
        <sequence>
          ...
          <!--This is the only element that changes-->
          <element name="direction" type="Direction"
            minOccurs="0" maxOccurs="1"
            fixed="out"/>
          ...
        </sequence>
        <attribute name="id" type="Identifier"/>
      </restriction>
    </complexContent>
  </complexType>

  <complexType name="DiversityComponentType">
    <complexContent>
      <extension base="ComponentType">
        <sequence>
          <element name="diversity" type="DiversityInterface"
            minOccurs="1" maxOccurs="1"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
</schema>
```

Рисунок 3.9 - Схема розширення xADL 2.0 для інтерфейсів різноманітності в стилі Koala.

Цей досвід також підсилює ефективність інструментів інфраструктури. Ми використовували XML Spy для перевірки та створення наших схем розширень та Arigen для створення нової бібліотеки зв'язування даних, яка їх підтримує. ArchEdit зміг автоматично підтримувати ці схеми. Таким чином, після написання коротких схем розширень, необхідних для відображення Koala та Mae в нашій інфраструктурі, наші інструменти автоматично забезпечили можливості парсингу, перевірки синтаксису та редагування графічного інтерфейсу.

Зрештою, ми використовували нашу інфраструктуру для підтримки її власного розвитку та еволюції. Як обговорювалося вище, наша інфраструктура включає низку взаємодіючих інструментів, які маніпулюють, аналізують та іншим чином працюють з описами архітектури. Більшість цих інструментів

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						66
Змн.	Арк.	№ докум.	Підпис	Дата		

покладаються на базовий набір сервісів для взаємодії з описами архітектури (парсування, читання, модифікація, збереження), що надаються компонентом xArchADT. Деякі інструменти покладаються на сервіси, що надаються іншими інструментами; наприклад, Menage покладається на дизайнерських критиків для виконання перевірки узгодженості описів архітектури, перш ніж маніпулювати ними. Ми створили середовище розробки на основі архітектури під назвою ArchStudio 3, яке інтегрує більшість наших інструментів як компоненти. Як і симулятор AWACS, архітектура ArchStudio описана в xADL 2.0. Завантажувач ArchStudio зчитує опис ArchStudio в xADL 2.0, створює екземпляри та з'єднує всі компоненти та з'єднувачі.

Набір інструментів ArchStudio 3 наразі включає xArchADT, фреймворк critics (і всі critics), ArchEdit, Menage, Selector та багато інших інструментів. Він складається приблизно з 80 000 рядків коду Java, не враховуючи код, згенерований Arigen. Наш досвід у створенні, підтримці та розвитку ArchStudio 3 демонструє, що наша інфраструктура може підтримувати розробку та дизайн нетривіального продукту, а також здатна описувати та підтримувати себе.

3.3 Аналіз ефективності та результатів використання модульного підходу

Під час створення та розвитку нашої інфраструктури ми засвоїли кілька важливих уроків щодо розробки мови опису модульної архітектури, як загалом, так і в конкретному контексті нашої інфраструктури на основі XML.

Як ми вже зазначали, наша інфраструктура не намагається автоматично вирішити проблему взаємодії функцій у розробці ADL. Тобто, вона не забезпечує інструментальної підтримки для забезпечення семантичної сумісності між функціями ADL. Тим не менш, в такій інфраструктурі, як наша, взаємодію функцій необхідно вирішувати під час створення нових модулів ADL. Ми отримали значний досвід у вирішенні проблем взаємодії функцій та запобіганні майбутнім труднощам, коли розробляли схеми xADL 2.0 та працювали з нашими

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						67
Змн.	Арк.	№ докум.	Підпис	Дата		

партнерами над подальшим розширенням мови для конкретних областей. Наші висновки щодо розробки нових функцій ADL, деякі з яких відображають загальноприйняту думку, узагальнено тут.

Відокремте основні концепції від деталей. Під час розробки модулів ADL концепції важливіші за конкретну реалізацію цих концепцій. Наприклад, наша абстрактна схема реалізації фіксує концепцію відображення реалізації та вказує, куди такі дані повинні йти, але не уточнює деталі того, які дані слід включати. Конкретні схеми, такі як схема реалізації Java, виконують цю функцію. Аналогічно, схема структури та типів визначає основні елементи архітектури (компоненти, з'єднувачі, інтерфейси та посилання), але не включає деталі, які потрібно вказати в розширеннях.

Зберігайте дерево залежностей неглибоким. Щоразу, коли між схемами вводиться семантична залежність, це означає, що ті, хто приймає залежні схеми, також повинні неявно приймати батьківські схеми. Іноді це бажано: xADL. Система типів забезпечує зручне місце для визначення варіантів. Однак, підтримуючи сумісність без залежностей між схемами Options, Variants та Versions, ми спрощуємо впровадження цих схем окремо.

Створюйте специфічні для домену функції поверх універсальних. Кожну додаткову схему необхідно ретельно семантично порівнювати з усіма іншими схемами в цільовій ADL для визначення сумісності. Щоб максимізувати можливість повторного використання схем, краще спочатку створювати та інтегрувати універсальні для домену схеми (схеми, функції яких корисні в багатьох доменах), а потім створювати специфічні для домену схеми.

Наприклад, ми спочатку вбудували такі функції, як підтримка лінійки продуктів та зіставлення реалізацій, у xADL 2.0. Оскільки схеми, загальні для домену, не залежать від специфічних для домену, їх можна легше перенести на інші проекти або ADL, оскільки вони не вимагатимуть перенесення або впровадження більш специфічних схем.

Усуньте надмірність, де це можливо. Дані, які необхідно вказувати або копіювати в кількох місцях, створюють надмірне навантаження на інструменти,

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						68
Змн.	Арк.	№ докум.	Підпис	Дата		

що підтримують ADL, для забезпечення узгодженості між копіями. Усунення або мінімізація надмірності може зменшити це навантаження. У схемах xADL 2.0 типи надаються як спосіб зберігання спільних даних про подібні компоненти, з'єднувачі та інтерфейси в одному місці.

Використовуйте елементи даних першого класу замість декорування існуючих елементів. Хоча природно розширювати визначення існуючих елементів додатковими даними, часто з точки зору розширюваності краще моделювати дані як набір сутностей першого класу та надавати посилання від існуючих елементів до даних. Наприклад, замість того, щоб включати дані про версії та посилання між пов'язаними версіями всередині типів компонентів та з'єднувачів, ми створили дерева версій першого класу та пов'язали типи з вузлами в дереві. Це дозволяє часткові специфікації (що вказує на існування версії без обов'язкового наявності повністю визначеного типу для її представлення). Це також допомагає відокремити дані та інформацію про версії від інших елементів, які *декорують* типи, таких як інформація про варіанти. Нарешті, використання елементів першого класу дозволяє багатьом сутностям посилатися на ту саму інформацію, допомагаючи мінімізувати надлишковість.

Окрім цих загальних рекомендацій, важко запропонувати розуміння найкращого способу інтеграції будь-якої нової функції. Зрештою, лише досвід може підтвердити, чи була певна функція інтегрована добре чи погано. Наприклад, ми вважаємо, що наша модуляризація xADL 2.0 є відносно хорошим та керованим розподілом завдань. Ми не можемо достовірно стверджувати, що наша розбивка функцій є найкращою, або що немає рефакторингу схем, який би призвів до більш елегантної, розширюваної або модульної нотації. Тим не менш, наш досвід використання нотації підвищує нашу впевненість у тому, що наша модуляризація є ефективною.

Наша інфраструктура пропонує деякі засоби для полегшення цієї труднощі. По-перше, вона заохочує експерименти з новими рефакторингами та змінами до існуючих нотацій легше, ніж монолітна мова. Тож, хоча вона не може допомогти користувачам у тому, як найкраще інтегрувати нову функцію, вона

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						69
Змн.	Арк.	№ докум.	Підпис	Дата		

може принаймні допомогти їм експериментувати з різними альтернативами. По-друге, вона заохочує поширення та повторне використання існуючих інтеграцій функцій, як ми це зробили з xADL 2.0. xADL несе в собі інженерні знання та досвід, які ми (та інші) мали в інтеграції багатьох корисних функцій; можливість повторно використовувати цю конкретну добре застарілу модуляризацію функцій набагато краща, ніж її переосмислення.

У постійному розвитку розширюваної мови, або будь-якої мови, яка змінюється з часом, не можна недооцінювати важливість рефлексивних та синтаксичних інструментів. Досвід показує, що за відсутності розумного методу розширення мови та її інструментальної підтримки, розширення зазвичай з'являються у вигляді хаків. Для мов програмування розширення зазвичай мають форму спеціально відформатованих коментарів, що зловживає цією особливістю мови просто тому, що вона ігнорується існуючими мовними інструментами, такими як компілятори та середовища. Для інших нотацій проектування, таких як UML, існуючі концепції перевантажені для підтримки нових функцій, щоб уникнути розширення визначення мови та інструментів. Іноді, коли нові функції занадто відрізняються від існуючої мови для інтеграції, користувачі намагаються використовувати кілька нотацій одночасно, або, що ще гірше, розробляють абсолютно нову, конкуруючу нотацію. Рефлексивні інструменти дозволяють розширювати мовні елементи та додавати до мови нові першокласні елементи, не вдаючись до хаків чи надмірного перевантаження.

Досвід роботи з нашою інфраструктурою, описаний у розділі 6, показав, що наявність інструментів на основі синтаксису та рефлексії, таких як Arigen, xArchADT та ArchEdit, допомагає зробити розширення існуючої нотації та її інструментів прийнятним. Вони надають можливість експериментувати з новими функціями моделювання, перш ніж брати на себе значні інвестиції у створення та адаптацію інструментів на основі семантики. Вони заохочують користувачів розширювати нотацію раціональним, принциповим способом, а не злому. Коли розширення зрештою будуть підтримуватися потужнішими інструментами на основі семантики, інструменти на основі синтаксису можуть

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						70
Змн.	Арк.	№ докум.	Підпис	Дата		

взяти на себе другорядну роль та доповнити інструменти на основі семантики.

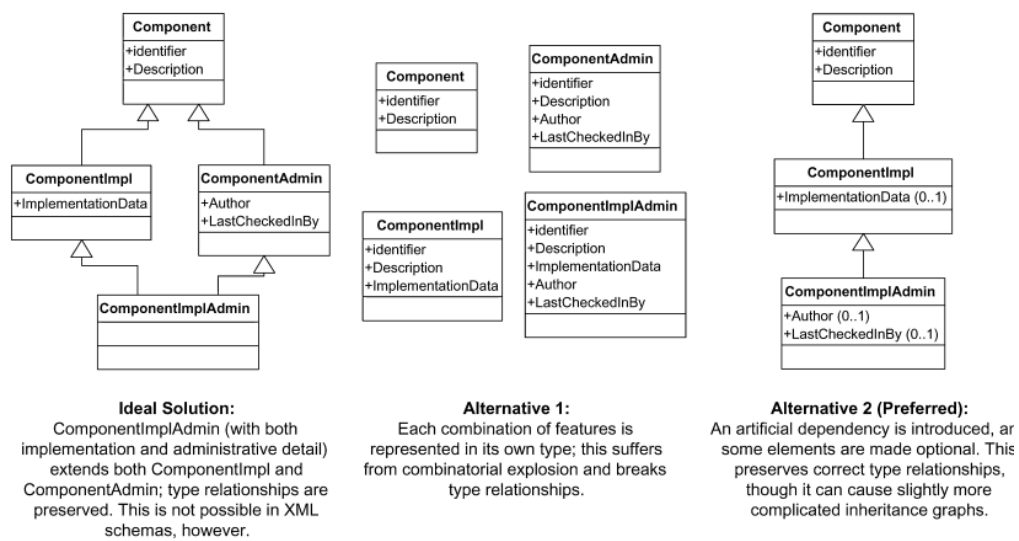


Рисунок 3.10 - Зображення проблеми множинного успадкування в XML-схемах та потенційні рішення.

XML-схемами показали задовільний результат, оскільки вони забезпечують виразну, але не надто складну метамову, що підтримується значним набором готових інструментів, що звільнило нас від необхідності писати парсери, серіалізатори та засоби перевірки синтаксису. Для нас найбільш проблематичним обмеженням XML-схем є те, що вони дозволяють лише одинарне успадкування типів даних для елементів та атрибутів, тобто підтип може мати лише один супертип. Тому підтипи, які змішують модель вмісту кількох супертипів, не дозволені в мовах на основі XML-схем. Існує кілька альтернативних метамов, які полегшують цю складність, таких як XInterfaces та RDF, але жодна з них не має підтримки доступних інструментів та комітетів зі стандартизації XML-схем. За нашим досвідом, переваги XML-схем переважають серйозність обмеження одинарного успадкування.

Щоб дослідити, як одинарне успадкування впливає на розробку мов моделювання програмного забезпечення, розглянемо базовий тип «Component», який описує програмний компонент. Тепер розглянемо два незалежних розширення Component. Функція 1 додає інформацію про реалізацію до опису

компонента, щоб описати, як він реалізований певною мовою програмування. Функція 2 додає адміністративні дані до опису компонента, щоб описати, хто відповідає за компонент і хто працював над ним у минулому. Ця ситуація зображена на рисунку 3.10. Семантично жодна з функцій не залежить від іншої. Тому в ідеалі має бути можливим моделювати простий компонент, компонент з деталями реалізації, компонент з адміністративними деталями та компонент з деталями реалізації та адміністрування.

У XML схем, існує кілька способів досягти цього. По-перше, можна було б створити окремі, незалежні типи для кожного з чотирьох варіантів компонента. Це працює, але призводить до комбінаторного вибуху типів, оскільки нові, ортогональні ознаки розповсюджуються. Крім того, це порушує очікувані зв'язки між типами (компонент -з-адміністративними-деталами не є підтипом компонента).

Бажаним рішенням є введення штучної залежності між двома розширеннями. Наприклад, `component-with-administrative-and-implementation-details` розширює `component-with-implementation-details`, який, у свою чергу, розширює `component`. Щоб врахувати компоненти, які мають лише деталі реалізації, адміністративні деталі робляться необов'язковими в розширенні. Це зберігає очікувані зв'язки типів (`component-with-administrative-and-implementation-details` є компонентом) і здатний моделювати всі чотири типи компонентів.

Остання альтернатива - використовувати групи підстановок та абстрактні базові типи для розширень. Це аналогічно додаванню списку `void*` до кожного класу об'єктів у C++ або вектора об'єктів у кінець кожного класу об'єктів у Java. Це неадекватно зберігає зв'язки між типами та послаблює здатність перевіряти архітектурні документи за допомогою перевірки типів XML.

Штучні залежності - це недосконале рішення, але вони працездатні та не страждають від комбінаторного вибуху. Якщо і коли W3C вирішить додати підтримку множинного успадкування до XML-схем у майбутньому, схеми та -

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						72
Змн.	Арк.	№ докум.	Підпис	Дата		

документи екземплярів, що використовують штучні залежності, мають бути досить простими для рефакторингу.

Ідея розширюваної мови опису архітектури не є новою, а наша інфраструктура не є першою, яка підтримує описи архітектури на основі XML. Однак наш підхід забезпечує перший модульний опис архітектури мова з корисними загальними функціями, розбитою на модулі та супроводжуваною - процесом та інструментами, які явно підтримують нові розширення мови.

Наш підхід був натхненний кількома іншими напрямками досліджень. По-перше, існує великий обсяг робіт з розширюваних та модульних мов, що охоплюють три десятиліття, який продовжується й сьогодні в таких проектах, як ArchJava, AspectJ та LXWB. По-друге, з'явилося багато мета- та універсальних середовищ моделювання, які технічно можуть підтримувати багато синтаксичних можливостей нашої інфраструктури на основі XML. По-третє, попередні дослідження ADL призвели до появи ADL на основі XML, таких як ADML та xADL 1.0, двох ADL першого покоління з обмеженою розширюваністю. Нарешті, важливо порівняти наш підхід з підходом Уніфікованої мови моделювання (UML), яка рекламується як мова опису архітектури.

Ці мови програмування дозволяли доповнювати свій синтаксис, зазвичай за допомогою нових правил виробництва BNF, та свою семантику за допомогою семантичних модулів під час виконання.

Мови програмування продовжують розширюватися й сьогодні. ArchJava розширює мову програмування архітектурними конструкціями та семантикою, такими як компоненти та конектори. AspectJ дозволяє визначати перехресні проблеми для програм Java, які «вплетені» в існуючі класи Java. У цих випадках семантичні можливості, що надаються інструментами, можуть бути досягнуті в цільовій мові програмування (Java) без розширень мови. Однак обсяг (значною мірою ручної) роботи для цього був би надзвичайно складним. ArchJava надає компактний спосіб визначення та перевірки обмежень на програмах Java, які практично неможливо перевірити вручну для системи будь-якого значного

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						73
Змн.	Арк.	№ докум.	Підпис	Дата		

розміру. AspectJ додає значну цінність Java, надаючи єдине місце, де можна визначати широко розподілені проблеми; раніше реалізація таких перехресних проблем виконувалася за допомогою копіювання та вставки коду або спеціальних препроцесорів. Різниця між цими інструментами та простими препроцесорами полягає в тому, що такі інструменти, як ArchJava та AspectJ, додають (і перевіряють) семантику, окрім тієї, що доступна в цільовій мові (тут Java), але зручно виводять байт-код Java для сумісності з існуючим середовищем виконання. Більш складна семантика (наприклад, додавання паралельності до мови програмування, яка її не має) може вимагати суттєвішої модифікації мови та інструментів.

Інші дослідження зосереджувалися на модульних предметно-орієнтованих мовах. Work Bench для розширення мови (LXWB) був розроблений у середині 1990-х років, щоб дозволити опис модульної мови за допомогою правил продукування, подібних до BNF. LXWB дозволяє визначати мовні модулі та автоматично генерує парсери та інші інструменти для композицій модулів, подібно до нашої інфраструктури. Фактично, аспекти розширюваності метамови, що підтримується LXWB, дуже схожі на ті, що містяться в XML-схемах. Металомова LXWB визначає систему типів над елементами мови та використовує успадкування для розширення існуючих конструкцій.

Як альтернативний шлях реалізації наших дослідницьких внесків, ми могли б обрати будь-яку з низки альтернативних мов та середовищ моделювання. Прикладами таких середовищ є Lisp, DOME та GME.

Lisp пропонувався як альтернатива використанню XML з моменту його появи. Зокрема, було висунуто аргумент, що s-вирази Lisp є кращим способом кодування ієрархічних даних, ніж XML і що включення додаткових елементів Lisp може означати вбудовування семантики та семантичних перевірок безпосередньо в саму мову.

DOME, середовище моделювання предметної області, – це інструмент для створення середовищ для розробки програмного забезпечення на основі моделей. DOME дозволяє користувачам визначати візуальні граматики, а потім

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						74
Змн.	Арк.	№ докум.	Підпис	Дата		

розробляти моделі, які є екземплярами цих граматики. Пакетна версія DOME має спеціальну підтримку моделювання в кількох популярних нотаціях моделювання, таких як об'єктно-орієнтований аналіз Коуда-Йордона та діаграми потоків даних. Можливість користувачів DOME створювати власні графічні граматики приблизно така ж, як редагування специфікації MOF для UML. Код можна додавати мовами, специфічними для DOME (так званими Projector та Alter), для аналізу моделей у граматиках, визначених користувачем.

GME, Загальне середовище моделювання, подібне до DOME. Як і в DOME, користувачі можуть редагувати графічну метамодель, а потім створювати екземпляри цієї моделі. Як і XML-схеми, GME використовує парадигму типів та екземплярів: метамоделі є типами, а моделі – екземплярами цих типів. Як GME, так і DOME підтримують поняття успадкування на рівні метамоделі, DOME підтримує одиночне успадкування вузлів, а GME підтримує віртуальне множинне успадкування.

Усі ці середовища є ефективно життєздатними альтернативами XML та XML-схемам, пропонуючи різні переваги та недоліки, зокрема з точки зору можливостей, які можуть забезпечити готові інструменти, такі як підтримка постачальників тощо. Ми не стверджуємо, що наш конкретний вибір та використання XML як середовища метамодельювання для нашої інфраструктури є дослідницьким внеском як таким; радше, наш дослідницький внесок полягає у формі конкретної декомпозиції функцій ADL, знайдених у xADL 2.0, розуміння того, як створювати та компонувати модулі ADL, та ролі різних видів інструментів у підтримці розробки та використання модульного ADL.

Доменно-орієнтовані ADL та доменно-орієнтовані програмні архітектури (DSSA) опосередковано пов'язані з нашою інфраструктурою. Завдяки обсягу знань про предметну область, що містяться в DSSA та пов'язаній з нею мові моделювання, можливий певний рівень компактності та повторного використання, якого немає в традиційних універсальних ADL. Завдяки модульності нашої інфраструктури та можливості робити довільні розширення ADL, наша інфраструктура ідеально підходить для побудови та

					БР.ІІІ – 00.00.00.000 ПЗ	Арк.
						75
Змн.	Арк.	№ докум.	Підпис	Дата		

експериментування за допомогою мов опису архітектури, специфічних для предметної області. Крім того, методи моделювання архітектури лінійки продуктів, що підтримуються в нашій інфраструктурі, забезпечують інтуїтивно зрозумілі способи визначення точок варіації в еталонній архітектурі.

Ранні ADL на основі XML досліджували можливість кодування описів архітектури програмного забезпечення в XML, але не використовували його розширюваність. Прикладами є xADL 1.0 та ADML. xADL 1.0 є інтелектуальним попередником xADL 2.0 та включав нові функції для ADL, такі як можливість вказувати зміни під час виконання та детальні специфікації інтерфейсу. Синтаксис xADL 1.0 був визначений в DTD і, отже, був розширюваним за допомогою просторів імен XML, але його інструменти не підтримували розширюваність. Більше того, його набір підтримуваних функцій був набагато меншим, ніж у xADL 2.0.

ADML - це, по суті, XML-переклад ядра Асме, також визначеного в DTD. ADML дещо покращив Асме, ввівши поняття метавластивостей та типів властивостей, але також не зміг використати розширюваність XML. Фактично, ADML досі використовує властивості пари ім'я-значення як механізм розширюваності, причому імена та значення просто закодовані в XML. Наш підхід є першим, який повністю використовує механізми розширюваності XML і не вимагає створення або використання власної метамови, такої як метавластивості ADML. Окрім кращої розширюваності, наша існуюча підтримка таких функцій, як архітектури лінійки продуктів та зіставлення реалізацій, перевищує можливості ADML.

Уніфікована мова моделювання – це спроба створити стандартну, універсальну, графічну мову моделювання для програмних систем. Існують певні розбіжності щодо того, чи є UML мовою ADL чи ні. З нашої точки зору, UML 1.x може використовуватися як ADL, але її необхідно певною мірою розширити для моделювання основних конструкцій архітектури програмного забезпечення, таких як компоненти та з'єднувачі. Щоб усунути цей недолік, UML 2.0 додав новий тип діаграми, який називається діаграмою складеної структури.

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						76
Змн.	Арк.	№ докум.	Підпис	Дата		

Діаграми складеної структури включають поняття частин, з'єднувачів, портів та зв'язків, які відповідно відображаються на компоненти, з'єднувачі, інтерфейси та зв'язки xADL 2.0.

Наш підхід суттєво покращує підхід UML у двох аспектах: функціональність та розширюваність. Що стосується функціональності, система типів xADL 2.0 та підтримка лінійки продуктів представляють можливості, яких немає в UML, включаючи UML 2.0.

Що ще важливіше, механізми розширюваності нашої інфраструктури та можливість додавання таких функцій набагато гнучкіші, ніж в UML. UML має два рівні розширюваності: вбудовані механізми розширюваності (стереотипи, теговані значення та обмеження) та редагування MOF. Вбудовані механізми UML дозволяють розширювати UML шляхом перевантаження існуючих конструкцій. Тобто, існуюча конструкція, така як клас або частина UML, може бути стереотипізована для використання як компонент або тип компонента архітектури програмного забезпечення. Цей підхід використовувався в кількох спробах адаптувати UML до ADL. Перевантаження може стати проблематичним, якщо потрібні конструкції, які природним чином не відповідають одному з існуючих типів діаграм UML. Наприклад, розглянемо таку конструкцію, як графи версій xADL 2.0. Можна було б створити граф вузлів, використовуючи існуючий тип діаграми UML, такий як діаграма класів, але метою діаграми класів є показ зв'язків між артефактами програмного забезпечення в проекті системи, а не відстеження еволюції компонента з часом. Альтернативою використанню вбудованих механізмів розширюваності є перевизначення UML у власній метамові, метаоб'єктному фреймворку (MOF). Це дозволяє довільну розширюваність, але результуюча мова не є UML, і тому інструменти UML зазвичай не підтримуватимуть цю нову мову. На противагу цьому, наш підхід забезпечує розширюваність, аналогічну редагуванню на рівні MOF, але надає інструменти, які очікують і підтримують цей рівень розширюваності.

Зв'язок UML з XML підтримується через формат XMI, формат обміну на основі XML, в якому можна зберігати та завантажувати діаграми UML. XMI

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						77
Змн.	Арк.	№ докум.	Підпис	Дата		

спрощує сумісність між інструментами UML, надаючи їм спільну текстову *лінгва франка* (графічний формат UML не має стандартного методу серіалізації), але не сприяє можливостям моделювання або розширюваності UML.

У цій роботі описано як дослідницький, так і технічний внесок. Дослідницький внесок включає перше розкладання мови опису архітектури на модулі, розуміння того, як розробляти нові мовні модулі та процес їх інтеграції, а також розуміння ролей різних видів інструментів у модульній інфраструктурі на основі ADL. Це супроводжується технічним внеском життєздатної інфраструктури для створення та розширення мов опису архітектури на основі XML. Інфраструктура значно зменшує обсяг зусиль, необхідних для експериментування та розробки нових архітектурних концепцій. Це зменшення є результатом трьох частин нашої інфраструктури: механізму розширюваності на основі XML, набору загальних базових схем та набору гнучких інструментів. Вона надає критично важливі інструменти, такі як парсери, редактори, засоби перевірки синтаксису та прив'язки даних для ADL, що дозволяє розробникам витрачати більше часу на створення високоцінних інструментів, що зосереджені на вирішенні відкритих питань.

Ми вважаємо, що наша інфраструктура виконала всі цілі. У Таблиці 3.2 наведено ці цілі та описано конкретні механізми інфраструктури, що використовуються для досягнення цих цілей.

Крім того, досвід, який ми та інші отримали у застосуванні нашого підходу, вказав на багато позитивних якісних аспектів цього підходу. Ми продемонстрували масштабованість нашої інфраструктури та гнучкість наших інструментів, моделюючи та імітуючи архітектуру програмного забезпечення AWACS. Адаптивність нашої інфраструктури до нової, раніше недослідженої області (програмне забезпечення космічних апаратів) та ефективність наших універсальних схем xADL 2.0 були продемонстровані роботою, виконаною в JPL. Ми продемонстрували, що нашу інфраструктуру можна використовувати для охоплення аспектів нової дослідницької галузі (архітектури лінійки

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						78
Змн.	Арк.	№ докум.	Підпис	Дата		

продуктів) за допомогою та Має. Наша інфраструктура підтримує власний розвиток та еволюцію в середовищі дизайну ArchStudio 3.

Таблиця 3.2

Цілі, досягнуті нашою інфраструктурою

Ціль	Характеристики інфраструктури
Встановити якомога менше обмежень на те, що може бути виражено на рівні архітектури.	Механізм розширення на основі XML дозволяє користувачам визначати довільні нові конструкції та розширювати будь-які існуючі. Досвід використання розширень JPL та Koala/MAE підтверджує ефективність цього механізму.
Дозволити додавання нових функцій моделювання та модифікацію існуючих з часом.	(Див. опис вище про механізм розширення XML).
Дозволити експериментування з новими функціями та їхніми комбінаціями.	Нові мови опису архітектури (ADL) можуть бути створені просто шляхом компонування набору схем.
Забезпечити розробників інструментів підтримкою для створення та маніпулювання моделями архітектури, навіть якщо базова нотація зміниться.	Синтаксичні інструменти автоматично надають миттєві парсери, прив'язки даних (data bindings) та редактори для нових схем без написання коду. Семантичні інструменти та середовища є компонентизованими та легкими для розширення (наприклад, інтеграція DRADEL).
Надати бібліотеку універсально корисних модулів, застосовних до широкого спектра областей.	Схеми xADL 2.0 є такими модулями. Деякі або всі ці схеми були повторно використані в кожному з етапів оцінки.
Дозволити повторне використання визначених функцій моделювання в інших проєктах.	Окрім повторного використання схем xADL 2.0, схеми розширення MDS, розроблені JPL, використовувалися як у генерації коду, так і в аналізі архітектури.

У майбутньому ми плануємо експериментувати з інтеграцією більшої кількості інструментів у нашу інфраструктуру через середовище ArchStudio 3, щоб додати корисні функції. Окрім цього, ми вважаємо, що інструмент xlinkit потенційно може бути використаний для вираження та перевірки обмежень на XML-посилання в документах xADL 2.0. Також набір інструментів SmartTools потенційно може забезпечити мову на основі XML, таку як xADL 2.0, з альтернативними редакторами та інструментами семантичного аналізу.

Для досягнення наших довгострокових дослідницьких цілей ми плануємо розширити схеми xADL 2.0, включивши нові конструкції моделювання, зокрема ті, що підтримуватимуть специфікацію розподілених та динамічних архітектур. Ми також хочемо дослідити застосування наших існуючих інструментів до нових проблем. Наприклад, ми вважаємо, що наші інструменти для лінійки продуктів можна застосовувати для підтримки специфікації та аналізу архітектурних компромісів, де варіанти продукту представляють собою точки в просторі доступних компромісів, а не фактичні описи продукту.

3.4 Висновки по розділу

У третьому розділі представлено реалізацію модульної програмної системи та оцінку її ефективності. Розглянуто інструментальні засоби підтримки модульної архітектури, а також практичне впровадження модульного підходу в розробку програмного забезпечення. Проведено аналіз результатів використання модульної структури, який підтвердив її переваги у масштабованості, підтримуваності та гнучкості системи. Отримані результати свідчать про ефективність модульного підходу в сучасній програмній інженерії.

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено архітектурні підходи до побудови модульних програмних систем, які є одним із ключових

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						80
Змн.	Арк.	№ докум.	Підпис	Дата		

напрямів сучасної програмної інженерії. У роботі проаналізовано сучасні принципи модульного проектування, досліджено переваги та недоліки модульної архітектури, а також розглянуто методи забезпечення масштабованості, підтримуваності та гнучкості програмного забезпечення. Процес дослідження охопив аналіз архітектурних підходів, моделювання структури програмної системи, оцінювання метрик якості та аналіз інструментальних засобів підтримки модульної архітектури.

На початковому етапі роботи було проведено аналіз предметної області та визначено основні проблеми, пов'язані з розробкою великих програмних систем. У результаті було визначено основні функціональні та нефункціональні вимоги до модульних програмних систем, серед яких масштабованість, повторне використання компонентів, зручність супроводу, незалежність модулів та забезпечення стабільної взаємодії між компонентами системи.

У першому розділі роботи було досліджено теоретичні основи модульної архітектури програмного забезпечення. Проведено аналіз принципів модульного проектування, зокрема розділення відповідальностей, низької зв'язаності та високої згуртованості модулів. Також розглянуто переваги та недоліки модульного підходу, а також метрики оцінювання якості модульних систем.

У другому розділі було проаналізовано методи та інструменти проектування модульних програмних систем. Розглянуто сучасні архітектурні підходи, серед яких компонентно-орієнтована архітектура, мікросервісний підхід та Service-Oriented Architecture (SOA). Особливу увагу приділено моделюванню еволюції архітектури та лінійок програмних продуктів, що дозволяє забезпечити адаптивність системи до змін вимог та розширення функціональності. У результаті дослідження було визначено основні підходи до побудови модульної архітектури, які забезпечують баланс між гнучкістю, продуктивністю та складністю реалізації.

У третьому розділі було розглянуто реалізацію модульної програмної системи та оцінено ефективність використання модульного підходу. Досліджено інструментальні засоби підтримки модульної архітектури, процеси

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						81
Змн.	Арк.	№ докум.	Підпис	Дата		

впровадження модульних компонентів та організацію взаємодії між ними. Аналіз результатів показав, що використання модульного підходу дозволяє спростити тестування та супровід програмного забезпечення, зменшити залежності між компонентами та забезпечити можливість незалежного оновлення окремих модулів.

Загалом, результати виконаної роботи підтверджують доцільність використання модульної архітектури для побудови сучасних програмних систем. Використання модульного підходу забезпечує підвищення гнучкості, масштабованості та підтримуваності програмного забезпечення, а також спрощує процеси модернізації та інтеграції нових функціональних можливостей. До основних переваг модульної архітектури можна віднести можливість повторного використання компонентів, незалежність модулів, зменшення складності системи та підвищення якості програмного забезпечення.

Разом із тим, у роботі було визначено певні обмеження та виклики, пов'язані з використанням модульного підходу. До них належать складність проектування взаємодії між модулями, необхідність підтримки узгодженості компонентів та додаткові витрати на організацію інфраструктури для взаємодії між модулями. У майбутньому модульні програмні системи можуть бути вдосконалені за рахунок використання технологій автоматизованого аналізу архітектури, штучного інтелекту для оптимізації структури системи та інтелектуальних засобів моніторингу взаємодії між компонентами.

					БР.ІІ – 00.00.00.000 ПЗ	Арк.
						82
Змн.	Арк.	№ докум.	Підпис	Дата		

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Bass, L., Clements, P., & Kazman, R. (2022). Software Architecture in Practice. addison-wesley.com. <https://www.informit.com/store/software-architecture-in-practice-9780136886099>
2. Booch, G. (2021). Object-Oriented Analysis and Design with Applications. pearson.com. <https://www.pearson.com/en-us/subject-catalog/p/object-oriented-analysis-and-design-with-applications/P200000003186>
3. Clean Architecture Principles. (2024). devblogs.microsoft.com. <https://devblogs.microsoft.com/architecture/clean-architecture/>
4. Docker Documentation. (2025). docs.docker.com. <https://docs.docker.com/>
5. Fowler, M. (2002). Patterns of Enterprise Application Architecture. martinfowler.com. <https://martinfowler.com/eaCatalog/>
6. Fowler, M. (2021). Microservices Resource Guide. martinfowler.com. <https://martinfowler.com/microservices/>
7. Freeman, E., & Robson, E. (2021). Head First Design Patterns. oreilly.com. <https://www.oreilly.com/library/view/head-first-design/9781492077992/>
8. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. pearson.com. <https://www.pearson.com/store/p/design-patterns-elements-of-reusable-object-oriented/P100000252058>
9. Garlan, D., & Shaw, M. (1996). Software Architecture: Perspectives on an Emerging Discipline. pearson.com. <https://www.pearson.com/en-us/subject-catalog/p/software-architecture-perspectives-on-an-emerging-discipline/P200000003464>
10. GitHub Documentation. (2025). docs.github.com. <https://docs.github.com/>
11. HashiCorp Terraform Documentation. (2025). developer.hashicorp.com. <https://developer.hashicorp.com/terraform/docs>

					БР.ІІІ – 00.00.00.000 ІІЗ	Арк.
						83
Змн.	Арк.	№ докум.	Підпис	Дата		

12. Hohpe, G., & Woolf, B. (2004). Enterprise Integration Patterns. enterpriseintegrationpatterns.com. <https://www.enterpriseintegrationpatterns.com/>
13. IEEE Software Engineering Standards. (2024). ieee.org. <https://standards.ieee.org/>
14. ISO/IEC 25010:2011 Systems and Software Quality Requirements and Evaluation (SQuaRE). (2024). iso.org. <https://www.iso.org/standard/35733.html>
15. ISO/IEC 12207 Software Life Cycle Processes. (2024). iso.org. <https://www.iso.org/standard/63712.html>
16. Jenkins Documentation. (2025). jenkins.io. <https://www.jenkins.io/doc/>
17. Kazman, R., Bass, L., & Abowd, G. (2020). Software Architecture Analysis Method. sei.cmu.edu. <https://www.sei.cmu.edu/>
18. Kubernetes Documentation. (2025). kubernetes.io. <https://kubernetes.io/docs/>
19. Larman, C. (2004). Applying UML and Patterns. pearson.com. <https://www.pearson.com/en-us/subject-catalog/p/applying-uml-and-patterns/P200000003088>
20. Martin, R. C. (2017). Clean Architecture: A Craftsman's Guide to Software Structure and Design. pearson.com. <https://www.pearson.com/store/p/clean-architecture-a-craftmans-guide-to-software-structure-and-design/P100001465201>
21. Microsoft .NET Architecture Guides. (2025). learn.microsoft.com. <https://learn.microsoft.com/en-us/dotnet/architecture/>
22. Microsoft Azure Architecture Center. (2025). learn.microsoft.com. <https://learn.microsoft.com/en-us/azure/architecture/>
23. Microservices Architecture Guide. (2025). microservices.io. <https://microservices.io/>
24. Newman, S. (2021). Building Microservices. oreilly.com. <https://www.oreilly.com/library/view/building-microservices-2nd/9781492034018/>
25. Object Management Group UML Specification. (2024). omg.org. <https://www.omg.org/spec/UML/>

					БР.ІІІ – 00.00.00.000 ІІЗ	Арк.
						84
Змн.	Арк.	№ докум.	Підпис	Дата		

26. OpenAPI Specification. (2025). swagger.io.
<https://swagger.io/specification/>
27. Oracle Java Platform Documentation. (2025). docs.oracle.com.
<https://docs.oracle.com/en/java/>
28. OSGi Alliance. (2024). OSGi Specifications. osgi.org.
<https://www.osgi.org/developer/specifications/>
29. OWASP Top Ten Security Risks. (2024). owasp.org.
<https://owasp.org/www-project-top-ten/>
30. Parnas, D. L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. carnegie-mellon.edu.
<https://www.cs.cmu.edu/~ModProb/KennethVanParys/Papers/Parnas-1972.pdf>
31. Postman API Platform Documentation. (2025). learning.postman.com.
<https://learning.postman.com/docs/>
32. Pressman, R. S., & Maxim, B. (2020). Software Engineering: A Practitioner's Approach. mcgraw-hill.com.
<https://www.mheducation.com/highered/product/software-engineering-practitioner-s-approach-pressman-maxim/M9781259872976.html>
33. Richardson, C. (2019). Microservices Patterns. manning.com.
<https://www.manning.com/books/microservices-patterns>
34. Sommerville, I. (2020). Software Engineering. pearson.com.
<https://www.pearson.com/en-us/subject-catalog/p/software-engineering/P200000003382>
35. Spring Framework Documentation. (2025). spring.io.
<https://spring.io/projects/spring-framework>
36. The Open Group Architecture Framework (TOGAF). (2024). opengroup.org. <https://www.opengroup.org/togaf>
37. Vernon, V. (2016). Domain-Driven Design Distilled. addison-wesley.com. <https://www.informit.com/store/domain-driven-design-distilled-9780134434421>

					БР.ІІІ – 00.00.00.000 ІІЗ	Арк.
						85
Змн.	Арк.	№ докум.	Підпис	Дата		

38. Wang, Y., & Li, Z. (2024). Modular Software Architecture for Scalable Enterprise Systems. Journal of Systems and Software, 208, 111901. <https://doi.org/10.1016/j.jss.2024.111901>

39. Zimmermann, O. (2023). Microservices and Cloud-Native Application Architecture. IEEE Software, 40(2), 18–25. <https://doi.org/10.1109/MS.2022.3229981>

40. Zivkovic, A., & Thompson, P. (2025). Evaluating Maintainability in Modular Software Systems. ACM Computing Surveys, 57(4), 1–28. <https://doi.org/10.1145/3701124>

					БР.ІІ – 00.00.00.000 ІЗ	Арк.
						86
Змн.	Арк.	№ докум.	Підпис	Дата		

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: " Архітектурні підходи до побудови модульних програмних систем"

Обсяг пояснювальної записки: 82 аркушів

Дата закінчення дипломної роботи 10 червня 2026р.

Підпис студента _____