МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Кваліфікаційна наукова
праця на правах рукопису

Луцик Ілля Ігорович

УДК 004.[02+4`2+4`6]

ДИСЕРТАЦІЯ

Методи та засоби створення адаптивних програмних систем на основі онтологій

(назва дисертації)

121 «Інженерія програмного забезпечення»

(шифр і назва спеціальності)

12 «Інформаційні технології»

(галузь знань)

Подається на здобуття наукового ступеня доктора філософії

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

_____ /І. І. Луцик/

Науковий керівник
Федасюк Дмитро Васильович
доктор технічних наук, професор

Львів – 2024

АНОТАЦІЯ

Луцик І. І. Методи та засоби створення адаптивних програмних систем на основі онтологій. – Кваліфікаційна наукова праця на правах рукопису.

Дисертація на здобуття наукового ступеня доктора філософії в галузі знань 12 Інформаційні технології за спеціальністю 121 «Інженерія програмного забезпечення» – Національний університет «Львівська політехніка», Львів, 2024.

У дисертаційній роботі розв'язано актуальну науково-прикладну задачу з інженерії програмного забезпечення – удосконалення процесу проєктування та розроблення адаптивних програмних систем на основі уніфікованої онтологічної моделі з використанням розроблених методів динамічного визначення налаштувань та модифікації функціональності програмної системи та інтерфейсу користувача з урахуванням даних про вимоги та потреби користувача.

Дисертаційна робота складається зі вступу, чотирьох розділів, висновків, списку літературних джерел та додатків.

У **вступі** наведено актуальність теми дослідження та поточний стан вирішення задачі. На основі проведеного аналізу тематики дослідження: сформульовано мету роботи та основні завдання дослідження; наведено методи дослідження для вирішення поставлених у дисертаційній роботі завдань; визначено наукову новизну та практичне значення отриманих результатів дослідження. Продемонстровано перелік опублікованих наукових праць за тематикою дисертаційної роботи у яких наведено та апробовано основні результати дослідження.

У **першому розділі** відображено сучасний стан проблеми розроблення і супроводу багатомодульних адаптивних програмних систем і технологій їх реалізації. Встановлено, що істотне зростання кількості вимог до програмних систем призводить до ускладнення їх архітектури, а часта зміна вимог значно

ускладнює модифікацію її компонент, особливо на пізніх етапах її розроблення. Це зумовлює необхідність використання методів динамічної адаптації програмної системи, які дають змогу змінювати налаштування та характеристики програмного забезпечення без необхідності його повторного налаштування та розгортання. Розглянуто можливості адаптивності та самоадаптивності компонент складних програмних систем та особливості її реалізації для динамічної модифікації функціональності. Встановлено, що основною проблемою класичних методів адаптації на основі концептуальних моделей предметної області є статичність структури відповідної моделі. Ця проблема унеможливлює внесення динамічних змін про нові функціональні та графічні компоненти під час роботи програмного забезпечення, що водночас обмежує процес модифікації програмної системи. На основі проведеного аналізу встановлено, що вирішенням проблеми статичності структури моделі є її уніфікація, що дає можливість наповнювати базу знань новими даними без необхідності модифікації структури моделі.

У **другому розділі** наведено розроблені методи та засоби адаптації програмних систем на основі онтологічної моделі. Визначено принципи використання онтологічного підходу у процесі створення адаптивних програмних систем. Розроблено онтологічну модель адаптивної програмної системи, використовуючи абстракції об’єктів предметної області. Встановлено, що така реалізація онтологічної моделі дає змогу не тільки уніфікувати класи та зв’язки між об’єктами, але й забезпечує можливість динамічного розширення та наповнення онтологічної бази знань інформацією без необхідності створення нових класів та зв’язків. Порівняно з класичним підходом це дозволяє уникнути жорсткої та статичної структури моделі програмної системи, що також сприяє динамічному визначенню налаштувань нових функціональних компонент.

Розроблено метод визначення налаштувань програмної системи на основі онтологічних правил та зв’язків. Спроєктований метод дозволяє визначати необхідні параметри для створення модифікованої конфігурації на основі даних про сутності та екземпляри онтологічної моделі, а також онтологічних правил

адаптації. Встановлено, що використання онтологічних правил дозволяє виділити процес формування адаптивних налаштувань у окремий сервіс, що забезпечить універсальність застосування процесу адаптації. Визначено, що наявні механізми міркувань здійснюють опрацювання онтологічних правил у послідовному режимі, що значно сповільнює процес визначення нових налаштувань системи. Вирішенням цієї проблеми є використання принципів горизонтального та вертикального масштабувань, які забезпечують розподіл навантаження на ресурс за рахунок поділу однієї великої онтологічної моделі на кілька підмоделей. Використовуючи розроблені онтологічну модель та метод визначення налаштувань програмної системи, спроєктовано метод динамічної адаптації функціональності програмної системи та графічного інтерфейсу користувача.

У **третьому розділі** наведено спроєктовані архітектурні рішення для реалізації багатомодульних адаптивних програмних систем. Визначено необхідні структурні рівні та компоненти, що забезпечують можливість динамічної адаптації програмного забезпечення. Встановлено, що використання об’єктно-орієнтованого підходу під час проєктування сервісу взаємодії з онтологічною моделлю дає змогу ефективно співставити об’єкти онтології з конкретними елементами програмного забезпечення. На основі вимог до адаптивних систем спроєктовано її архітектуру. Визначено, що поєднання елементів трирівневої клієнт-серверної та компонентно-орієнтованої архітектури забезпечує достатній рівень абстрагування та надає можливість виконання динамічної модифікації програмної системи. Можливість розширення функціональності та графічного інтерфейсу програмного продукту забезпечується за допомогою використання додаткових (plugin) компонент, які можуть динамічно доєднуватися до основного модуля та при цьому не впливають на роботу інших її компонент.

Розроблено механізм взаємодії між елементами програмного забезпечення з використанням брокерів повідомлень. Встановлено, що використання брокерів повідомлень дасть змогу розподілити навантаження між

сервісами, а також забезпечити можливість подальшого масштабування програмного забезпечення залежно від мережевого трафіку. Проаналізовано особливості використання технологій брокерів повідомлень та наведено основні групи та типи обмінів (“Exchange”). Встановлено, що у випадку динамічної адаптації програмного забезпечення доцільним є використання типу обміну “dead letter exchange”. Така реалізація взаємодії між сервісами дасть змогу не тільки розподілити навантаження на програмні ресурси, але й надасть можливість контролю виняткових ситуацій у разі виникнення помилки у опрацюванні онтологічних правил.

Розроблено алгоритм динамічної адаптації інтерфейсу та функціональності програмної системи з використанням онтологічної моделі та принципів рефлексії. Проаналізовано переваги рефлексії та випадки її застосування у процесі розробки програмного забезпечення. Встановлено, що рефлексія дасть змогу отримувати інформацію про збірку та конфігурацію програмної системи. Це дасть змогу здійснювати динамічну адаптацію компонентів програмної системи відповідно до визначених налаштувань. Запропонований алгоритм забезпечує можливість динамічної адаптації без необхідності статичного створення усіх можливих конфігурацій. Окрім цього, наявність кешування та брокера повідомлень дасть змогу зменшити кількість запитів до системи у разі, якщо конфігурація уже існує.

У **четвертому розділі** відображено реалізацію адаптивної програмної системи та апробацію результатів дослідження. Здійснено опис особливостей використаних технологій проєктування та розроблення програмної системи. Продемонстровано часткову реалізацію розробленої бібліотеки адаптації з використанням технології C#/.NET та елементів рефлексії для виконання адаптації під час роботи програмного забезпечення. Визначено та наведено налаштування брокера повідомлень на основі використання технології RabbitMQ. Подано використання технології SignalR для надсилання користувачам результатів визначення адаптивних налаштувань системи у режимі реального часу без необхідності здійснення повторних запитів до

вебсерверу. Продемонстровано реалізацію взаємодії сервісу бази даних та знань зі спроєктованою онтологічною моделлю за допомогою використання технології Owlready2 та мови програмування Python.

Здійснено порівняння абстрактного та класичного підходів до проєктування онтологічних моделей на основі метрик якості. Встановлено, що при абстрактному підході зменшуються показники структурно-ієрархічних метрик, що підтверджує кращий рівень розуміння структури онтології та більш коректний процес опрацювання онтологічних правил. Порівняння значень метрик наповнення бази знань демонструють для абстрактного підходу більшу гнучкість розподілу та вищу насиченість екземплярів сутностей, що доводить загальну ефективність використання абстракцій об’єктів предметної області.

Проаналізовано тривалості адаптації програмної системи залежно від її конфігурації. Визначено, що тривалість конфігурування програмних систем на основі проєктування онтології з використанням абстрактного підходу в середньому на 20 % нижча порівняно з класичним підходом. Окрім цього, при опрацюванні більше 3000 елементів онтології, абстрактний підхід забезпечує майже вдвічі більшу швидкодію адаптації, що підтверджує його загальну ефективність. Встановлено, що використання принципів горизонтального масштабування онтологічної моделі при абстрактному підході дає змогу не тільки знизити навантаження на програмні та апаратні ресурси, але й також зменшити тривалість визначення налаштувань в середньому на 19 %, що забезпечує більшу швидкодію опрацювання запитів на адаптацію програмної системи та формування її конфігурації.

Ключові слова: онтологія, онтологічна модель, онтологічний підхід, онтоорієнтовані системи, методи адаптації програмного забезпечення, конфігурація програмних систем, архітектура програмних систем, адаптивні програмні системи, самоадаптивні програмні системи, процес адаптації, модифікація компонентів системи, метрики оцінки онтології, брокери повідомлень, горизонтальне масштабування.

ABSTRACT

I. I. Lutsyk. Methods and means of creating adaptive software systems based on ontologies. –Qualifying scientific work on the rights of the manuscript.

Dissertation for obtaining the scientific degree of Doctor of Philosophy in the field of knowledge 12 Information technologies in the specialty 121 Software engineering. – Lviv Polytechnic National University, Lviv, 2024.

The thesis solves a current scientific and applied problem in the field of software engineering – improvement of the process of design and development of adaptive software systems based on a unified ontological model using the developed methods of dynamic determination of settings and modification of the functionality of the software system and user interface, taking into account data on user requirements and needs.

The thesis paper consists of introduction, four chapters, conclusions, list of references and appendices.

The **introduction** presents the relevance of the research topic and the current state of solving the problem. Based on the analysis of the research topic: the purpose of the work and the main tasks of the research are formulated; research methods for solving the tasks set in the dissertation work are given; the scientific novelty and practical significance of the obtained research results are determined. The list of published scientific works on the subject of the dissertation is demonstrated, in which the main results of the research are presented and tested.

The **first chapter** reflects the current state of the problem of development and support of multi-module adaptive software systems and technologies for their implementation. It has been established that a significant increase in the number of requirements for software systems leads to the complexity of their architecture, and the frequent change of requirements significantly complicates the modification of its components, especially at the later stages of its development. This necessitates the use of methods of dynamic adaptation of the software system, which make it possible to change the settings and characteristics of the software without the need for its

reconfiguration and deployment. The possibilities of adaptability and self-adaptability of components of complex software systems and the peculiarities of its implementation for dynamic modification of functionality are considered. It was established that the main problem of classical adaptation methods based on conceptual models of the subject area is the static structure of the corresponding model. This problem makes it impossible to make dynamic changes about new functional and graphical components during the operation of the software, which at the same time limits the process of modifying the software system. On the basis of the conducted analysis, it was established that the solution to the problem of the static structure of the model is its unification, which makes it possible to fill the knowledge base with new data without the need to modify the model structure.

The **second chapter** presents the developed methods and tools for adapting software systems based on the ontological model. The principles of using the ontological approach in the process of creating adaptive software systems are defined. An ontological model of an adaptive software system has been developed, using abstractions of domain objects. It was established that this implementation of the ontological model allows not only to unify classes and relationships between objects, but also provides the possibility of dynamic expansion and filling of the ontological knowledge base with information without the need to create new classes and relationships. Compared to the classical approach, this avoids the rigid and static structure of the software system model, which also facilitates the dynamic determination of the settings of new functional components.

A method of determining software system settings based on ontological rules and relationships has been developed. The designed method allows you to determine the necessary parameters for creating a modified configuration based on data about entities and instances of the ontological model, as well as ontological adaptation rules. It was established that the use of ontological rules allows the process of forming adaptive settings to be separated into a separate service, which will ensure the universality of the application of the adaptation process. It was determined that the available reasoning mechanisms process ontological rules in a sequential mode,

which significantly slows down the process of determining new system settings. The solution to this problem is the use of the principles of horizontal and vertical scaling, which ensure the distribution of the load on the resource due to the division of one large ontological model into several sub-models. Using the developed ontological model and the method of determining the settings of the software system, a method of dynamic adaptation of the functionality of the software system and the graphical user interface was designed.

The **third chapter** presents the designed architectural solutions for the implementation of multi-module adaptive software systems. The necessary structural levels and components that provide the possibility of dynamic adaptation of the software have been determined. It has been established that the use of an object-oriented approach during the design of an interaction service with an ontological model makes it possible to effectively match ontology objects with specific software elements. Its architecture was designed based on the requirements for adaptive systems. It was determined that the combination of elements of the three-level client-server and component-oriented architecture provide a sufficient level of abstraction and enable dynamic modification of the software system. The possibility of expanding the functionality and graphical interface of the software product is provided by the use of additional (plugin) components that can be dynamically added to the main module and at the same time do not affect the operation of its other components.

A mechanism of interaction between software elements using message brokers has been developed. It has been established that the use of message brokers allows to distribute the load between services, and also provides the possibility of further scaling of the software depending on the network traffic. The use of message broker technologies is analyzed and the main groups and types of exchanges are presented. It has been established that in the case of dynamic software adaptation, it is advisable to use the "dead letter" type of exchange. This implementation of interaction between services will allow not only to distribute the load on software resources, but also provide an opportunity to control exceptional situations in the event of an error in the

processing of ontological rules.

An algorithm for dynamic adaptation of the interface and functionality of the software system using the ontological model and principles of reflection has been developed. The advantages of reflection and cases of its application in the process of software development are analyzed. It is established that reflection will allow to obtain information about the assembly and configuration of the software system. This will make it possible to dynamically adapt the components of the software system in accordance with the specified settings. The proposed algorithm provides the possibility of dynamic adaptation without the need to statically create all possible configurations. In addition, the presence of caching and a message broker will reduce the number of requests to the system if the configuration already exists.

The **fourth chapter** reflects the implementation of the adaptive software system and the testing of the research results. A description of the features of the used technologies for designing and developing the software system was made. A partial implementation of the developed adaptation library using C#/.NET technology and reflection elements to perform adaptation during software operation is demonstrated. The configuration of the message broker based on the use of RabbitMQ technology is defined and given. The use of SignalR technology is presented to send the results of the determination of adaptive system settings to users in real time without the need to make repeated requests to the web server. The implementation of the interaction of the database and knowledge service with the projected ontological model using the Owlready2 technology and the Python programming language is demonstrated.

A comparison of abstract and classical approaches to the design of ontological models based on quality metrics was made. It was established that with an abstract approach, the indicators of structural and hierarchical metrics decrease, which confirms a better level of understanding of the structure of the ontology and a more correct process of developing ontological rules. The comparison of the values of the knowledge base filling metrics shows for the abstract approach a greater flexibility of distribution and a higher saturation of entity instances, which proves the general efficiency of using domain object abstractions.

The duration of adaptation of the software system depending on its configuration was analyzed. It was determined that the duration of configuring software systems based on ontology design using an abstract approach is on average 20% lower compared to the classical approach. In addition, when processing more than 3000 elements of the ontology, the abstract approach provides almost twice the speed of adaptation, which confirms its overall effectiveness. It was established that the use of the principles of horizontal scaling of the ontological model with an abstract approach allows not only to reduce the load on software and hardware resources, but also to reduce the duration of determining settings by an average of 19%, which ensures a faster processing of adaptation requests software system and formation of its configuration.

Keywords: ontology, ontological model, ontological approach, ontological systems, methods of software adaptation, configuration of software systems, architecture of software systems, adaptive software systems, self-adaptive software systems, adaptation process, modification of system components, ontology evaluation metrics, message brokers, horizontal scaling.

СПИСОК ОПУБЛІКОВАНИХ ПРАЦЬ ЗА ТЕМОЮ ДИСЕРТАЦІЇ

Статті у періодичних наукових виданнях, які включені до наукометричних баз даних

1. Fedasyuk D., Lutsyk I. Approach to implementation of configuration process for Adaptive Software Systems based on Ontologies. *International Journal of Computing*, 2023. Vol 22, No. 3, p. 381–388. https://doi.org/10.47839/ijc.22.3.3234

Статті у наукових фахових виданнях України

2. Федасюк Д. В., Луцик І. І. Адаптивна програмна система на основі онтологічного підходу для людей з когнітивними порушеннями. *Вісник Національного університету «Львівська політехніка». Серія Інформаційні системи та мережі*, 2021, № 9, с. 61–74. https://doi.org/10.23939/sisn2021.09.061

3. Fedasyuk D., Lutsyk I. Analysis of approaches to design ontological models of an adaptive software system. *Computer systems and information technologies*, *202*4. No. 3, p. 13–20. https://doi.org/10.31891/csit-2024-3-2

Публікації в матеріалах конференцій, що проіндексовані у наукометричних базах даних

4. Fedasyuk D., Lutsyk I. Tools for adaptation of a mobile application to the needs of users with cognitive impairments. 2021 IEEE 16th International Conference on Computer Sciences and Information Technologies (CSIT), (Lviv, Ukraine, September 22–25, 2021). IEEE, 2021, p. 321-324. https://doi.org/10.1109/CSIT52700.2021.9648702 (НБД SCOPUS).

5. Fedasyuk D., Lutsyk I. Method of modification of self-adaptive software systems based on ontology. 2022 IEEE 16th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET), (Lviv-Slavske, Ukraine, February 22–26, 2022). IEEE, 2022, p. 530-533. https://doi.org/10.1109/TCSET55632.2022.9766856 (НБД SCOPUS).

6. Fedasyuk D., Lutsyk I. The Use of Ontology in the Process of Designing

Adaptive Software Systems. 2022 IEEE 17th International Conference on Computer Sciences and Information Technologies (CSIT), (Lviv, Ukraine, November 10–12, 2022). IEEE, 2022, p. 503-506. https://doi.org/10.1109/CSIT56902.2022.10000528 (НБД SCOPUS).

Публікації в матеріалах конференцій

7. Луцик І. І., Федасюк Д. В. Програмна система для допомоги людям старшого віку з когнітивними порушеннями. Матеріали конференції Інформаційні технології – 2020: VII Всеукраїнська науково-практична конференція молодих науковців. (м. Київ, 21 травня 2020 р.). Київ, 2020, с. 123–125. URL: https://informationtechn2020.blogspot.com/2020/05/blog-post_19.html

8. Луцик І. І., Федасюк Д. В. Онтологічна модель адаптивної програмної системи для людей з когнітивними порушеннями. Матеріали конференцій Молодіжної наукової ліги «Теоретичне та практичне застосування результатів сучасної науки». (м. Запоріжжя, 27 листопада 2020 р) Запоріжжя, 2020, с. 49–52. https://doi.org/10.36074/27.11.2020.v2.04

9. Луцик І. І., Луцик І. Б. Використання онтологій для програмних модулів адаптивних систем керування на базі нечіткої логіки. Матеріали VIII міжнародної науково-технічної конференції КМОСС-2023. (м. Дніпро, 1-3 листопада 2023 р). 2023, с. 115–116. URL: https://udhtu.edu.ua/wp-content/uploads/2023/11/zbirnyk-tez-kmoss-2023.pdf

ЗМІСТ

ВСТУП

Актуальність дослідження. Розвиток сучасних інформаційних технологій спонукає до поширення використання програмних систем різної складності у всіх сферах людської життєдіяльності. Враховуючи змінні потреби та вимоги користувачів, такі системи потребують постійного вдосконалення. Проте, модифікація наявної функціональності та створення нових її функцій є ресурсовитратними процесами, що часто потребують повторного виконання етапів розробки, враховуючи необхідність інтеграції нових компонент у вже наявне програмне забезпечення. Окрім цього, така зміна функціональності системи часто вимагає залучення експертів з відповідної предметної області, які не є фахівцями інженерії програмного забезпечення. Тому актуальним є вдосконалення процесів проєктування та розроблення програмних систем, що забезпечують можливість динамічної адаптації її компонент до нових вимог користувача.

Ефективним вирішенням визначеної проблеми є застосування механізмів адаптації та самоадаптації програмних систем. Для ідентифікації необхідних змін та аналізу системних характеристик доцільним є використання діаграм управління та зворотного зв’язку. Принципи використання механізмів самоадаптації у поєднанні з діаграмами управління та зворотного зв’язку при проєктуванні адаптивних програмних систем для забезпечення можливості динамічної зміни характеристик програмного засобу подано у дослідженнях А. Філієрі, Г. Мюллера [51, 113]. Однак, як на нашу думку, використання класичних підходів до проєктування адаптивних програмних систем не завжди забезпечує можливість динамічної реконфігурації системи без потреби її повторного налаштування та розгортання.

Іншим вирішенням даної проблеми є використання онтологічних моделей для представлення об’єктів та зв’язків між ними в предметній області. Принципи та технології використання онтологічного підходу в розробці адаптивних програмних систем висвітлено у дослідженнях Є. Бурова,

В. Пасічника [2, 26,], К. Ангелопулоса, А. Пападопулоса [13], К. Песанья Б. Дуарте, В. Соуза [89], В. Литвина [3, 79]. На відміну від класичних підходів моделювання, онтологічний підхід дає змогу проєктувати формальну модель предметної області, яку можна змінювати та повторно використовувати в процесі розробки програмного забезпечення без необхідності виконання повторної реконфігурації системи.

Незважаючи на загальну ефективність застосування онтологічного підходу до проєктування адаптивних програмних систем, вважаємо, що наявні рішення не повною мірою забезпечують динамічність модифікації функціоналу програмної системи та інтерфейсу користувача. Окрім цього, при створенні такої моделі для програмного забезпечення враховують статичну інформацію про предметну область, що також негативно впливає на її можливість швидкого розширення чи уточнення.

Враховуючи наявні проблеми, вважаємо, що актуальним є створення методів та засобів розробки адаптивних програмних систем на основі онтологій, які передбачають можливість динамічної зміни їх функціональності програмної системи та інтерфейсу користувача залежно від середовища виконання та потреб користувача.

Зв’язок роботи з науковими програмами, планами, темами.

Дисертацію виконано на кафедрі програмного забезпечення Національного університету «Львівська політехніка». Тема дисертації відповідає науковому напрямку кафедри – програмне та математичне забезпечення автоматизованих систем. Зокрема, у 2024 – 2025 рр. дисертаційні дослідження виконувалися в межах кафедральної науково-дослідної роботи «Розробка адаптивних програмних систем з використанням онтологічного підходу», номер держреєстрації 0124U004349.

Мета дослідження: удосконалення процесів проєктування та розроблення адаптивних програмних систем на основі використання уніфікованої онтології, компонентно-орієнтованої архітектури, а також методу динамічної адаптації функціональності програмної системи та інтерфейсу

користувача.

Об’єкт дослідження: процес розроблення адаптивних програмних систем.

Предмет дослідження: методи та засоби динамічної адаптації функціональності програмної системи та інтерфейсу користувача на основі онтологічного підходу.

Завдання дослідження:

1. Проаналізувати наявні методи проєктування та розроблення адаптивних програмних систем, а також технології динамічної модифікації функціональності та інтерфейсу користувача.
2. Розробити метод побудови моделі адаптивної програмної системи на основі онтологічного підходу та удосконалити метод визначення налаштувань програмної системи на основі її правил та зв’язків.
3. Розробити метод динамічної адаптації функціональності програмної системи та інтерфейсу користувача на основі онтологій.
4. Спроєктувати архітектуру адаптивної програмної системи, що забезпечить можливість динамічної модифікації її функціональності та інтерфейсу користувача.
5. Дослідити швидкодію та ефективність процесу динамічної адаптації функціональності та інтерфейсу користувача на основі створеного прототипу адаптивної програмної системи.

Методи дослідження: Для вирішення поставлених у дисертаційній роботі завдань використано такі методи дослідження: методи онтологічного моделювання та теорії множин – для створення онтологічних моделей, представлення та опрацювання знань з предметних областей; методи системного аналізу та математичного моделювання – для аналізу впливу параметрів програмного забезпечення та зовнішніх чинників на загальну продуктивність системи; принципи об’єктно-орієнтованого програмування, методи моделювання на базі мов UML, SWRL та OWL – для проєктування та

розроблення адаптивної програмної системи.

Наукова новизна отриманих результатів:

1. Вперше розроблено метод побудови моделі адаптивної програмної системи на основі онтологічного підходу, що дозволяє враховувати особливості понять та зв’язків предметних областей під час проєктування програмного засобу та забезпечує можливість динамічного формування його системних характеристик.

2. Вперше розроблено метод визначення налаштувань програмної системи на основі онтологічних правил та зв’язків, використовуючи інформацію про вимоги та потреби користувача, що дає змогу опрацьовувати семантичні правила для окремих понять онтологічної моделі, а також забезпечує зменшення тривалості опрацювання інформації та скорочує процес адаптації програмної системи;

3. Удосконалено метод динамічної адаптації функціональності програмної системи та інтерфейсу користувача, який на відміну від наявних методів, дає змогу враховувати інформацію про поточний активний пристрій та проводити модифікацію компонент відповідно до нових вимог чи потреб користувача без необхідності повторної конфігурації та розгортання програмних систем.

4. Удосконалено метод використання брокерів повідомлень у процесі адаптації програмного забезпечення, що, у поєднанні з горизонтальним масштабуванням онтологічної моделі, дозволяє зменшити навантаження на сервіс адаптації за рахунок розподілу запитів до бази знань.

5. Отримала подальший розвиток технологія створення адаптивних програмних систем на основі онтологічного підходу, яка, на відміну від уже наявних, усуває статичність структури онтологічної моделі та дає змогу уніфікувати процес адаптації програмних систем для різних предметних областей.

Практичне значення отриманих результатів:

1. Спроєктована онтологічна модель адаптивних програмних систем дає змогу уніфіковано здійснювати представлення предметної області та забезпечує високий рівень її абстрактності, що в свою чергу зменшує зв'язність моделі та дає можливість наповнювати базу знань новими сутностями у процесі роботи програмної системи без необхідності реконфігурації у разі зміни вимог чи потреб користувача.

2. Розроблено бібліотеку для створення адаптивних програмних систем: визначені основні компоненти та сервіси, які необхідні для реалізації динамічного процесу адаптації програмних систем; створено уніфікований та кросплатформовий інтерфейс для забезпечення процесу адаптації та модифікації конфігурації програмної системи.

3. Створено сервіс бази даних та знань, що реалізує метод визначення налаштувань програмної системи з використанням технології брокера повідомлень RabbitMQ. У разі виникнення помилок у процесі визначення адаптивних налаштувань програмного забезпечення сервіс конфігурації перенаправляє первинний запит до спеціалізованої черги, яка через визначений проміжок часу знову надсилає запит на адаптацію. Це дозволяє не лише зменшити навантаження на ресурси системи але й забезпечити можливість автоматизованої повторної адаптації у разі виникнення помилок.

Результати дисертаційної роботи впроваджені у навчальному процесі кафедри програмного забезпечення НУЛП для підготовки студентів спеціальності 121 «Інженерія програмного забезпечення» освітньо-кваліфікаційного рівня магістр в лекційному курсі та практичних заняттях дисципліни «Наукові дослідження та семінари за їх тематикою».

Дослідне випробування результатів дисертаційного дослідження проведено на ПП «ЛІНК АП СТУДІО» (м.Львів).

Особистий внесок здобувача.

Основні наукові результати та висновки дисертаційної роботи отримані автором самостійно та викладені у 9 наукових працях, що були написані у

співавторстві. Зокрема, у працях, які написані у співавторстві, дисертантові належать наступні напрацювання:

- [5–7, 47, 48, 77] – розробка онтологічних моделей адаптивних програмних системи на основі класичного та абстрактного підходів до проєктування онтологій.
- [45, 46] – проєктування методів динамічної адаптації функціональних та графічних компонент програмного забезпечення, що використовують дані про вимоги до ПЗ та інформацію про активний пристрій для автоматизованої генерації нової конфігурації ПЗ;
- [8, 45, 48] – проєктування комбінованої архітектури адаптивного програмного забезпечення з використанням механізму брокерів повідомлень та онтологічної моделі предметної області.
- [78]– проєктування механізму обміну повідомленнями на основі використання технології RabbitMQ, що враховує додаткові налаштування програмного забезпечення та використовує механізм опрацювання виняткових ситуацій.
- [45, 78] – розробка алгоритму адаптації програмного забезпечення відповідно до вимог користувача з використанням принципів рефлексії та технології RabbitMQ.

Постановка завдань та їхнє обговорення здійснено під керівництвом д.т.н., проф. Федасюка Д.В.

Апробація матеріалів дисертації. Результати дисертаційного дослідження апробовано на міжнародних наукових та науково-практичних конференціях, наукових школах та консорціумах, семінарах:

1. VII Всеукраїнська науково-практична конференція молодих науковців, 2020;
2. Міжнародна студентська наукова конференція «Теоретичне та практичне застосування результатів сучасної науки», 2020;
3. IEEE 16th International Conference on Computer Sciences and Information Technologies, 2021;

4. IEEE 16th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering, 2022;

5. IEEE 17th International Conference on Computer Sciences and Information Technologies, 2022;

6. VII Міжнародна науково-технічна конференція «Комп'ютерне моделювання та оптимізація складних систем», 2023.

7. 19th International Conference on ICT in Education, Research, and Industrial Applications, ICTERI 2024, 2024

Публікації. За матеріалами дисертаційної роботи опубліковано 9 наукових праць, з яких 2 статті у наукових фахових виданнях України [8, 77], 1 стаття в науковому періодичному виданні, яке включено до міжнародної наукометричної бази Scopus [45], 6 – у матеріалах і тезах конференцій [5–7, 46–48].

Структура й обсяг дисертації. Дисертаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел (123 найменувань) і 4 додатки. Основний зміст викладено на 116 сторінках друкованого тексту, містить 31 рисунок, 7 таблиць. Загальний обсяг роботи – 158 сторінок.

РОЗДІЛ 1. СУЧАСНИЙ СТАН ТА ТЕХНОЛОГІЇ СТВОРЕННЯ АДАПТИВНИХ ПРОГРАМНИХ СИСТЕМ

1.1. Особливості проблеми створення адаптивних програмних систем

Розробка програмного забезпечення це складний процес, який включає в себе послідовність конкретно визначених етапів: визначення вимог, проєктування, розробка, тестування, впровадження та підтримку [65]. Модифікація існуючої функціональності чи створення нових функцій є ресурсовитратним процесом та потребує повторного виконання етапів розробки, враховуючи необхідність інтеграції внесених змін в уже існуючу систему. Внаслідок цього виникає проблема ефективного планування та використання ресурсів розробки для впровадження нового функціоналу. Крім цього, модифікація функціональності та елементів графічного інтерфейсу системи до нових вимог вимагає залучення експертів з відповідної предметної області, які часто не є фахівцями в галузі програмної інженерії.

1.1.1. Поняття адаптивності програмних систем та основні принципи її реалізації

Вирішенням проблеми модифікації характеристик є розробка програмної системи, що надає можливість динамічного розширення функціоналу та додавання ресурсів для графічного інтерфейсу. Проте більшість наявних рішень не дозволяють виконувати модифікацію без необхідності здійснення статичної реконфігурації системи.

Для забезпечення можливості динамічної зміни характеристик створеної програмної системи зазвичай використовуються два її типи: *адаптивні* та *самоадаптивні*. Різниця у цих двох підходах полягає у процесі визначення необхідності проведення процесу модифікації характеристик та властивостей програмних рішень. На відміну від адаптивних, самоадаптивні системи здатні самостійно аналізувати конкретні характеристики і, в подальшому, змінювати

свою поведінку в залежності від отриманої оцінки ефективності чи продуктивності [73].

Підходи на основі методів адаптації та самоадаптації є одними з найбільш ефективних та спрямовані на вдосконалення процесу проєктування та розробки комплексного програмного забезпечення, що має складну архітектуру та динамічні вимоги користувача [106].

У ряді досліджень визначено загальні принципи та методи проєктування та впровадження самоадаптивних систем. Зокрема, у [73, 80] автори висвітлювали проблеми проєктування цих систем, а саме: проблеми визначення нових уніфікованих процесів адаптації програмного забезпечення та децентралізації елементів системи. В результаті аналізу подано нові напрями дослідження самоадаптивних систем та можливі труднощі, пов'язані з розробкою складного програмного забезпечення. Детальний аналіз рішень для створення самоадаптивних систем та їх таксономія (рис. 1.1) наведено в дослідженні [71].

Рис. 1.1. Таксономія підходів та принципів реалізації самоадаптивних програмних систем

На основі таксономії підходів дослідниками визначено нові напрями розвитку процесу адаптації програмних систем:

- контекстна адаптація – даний підхід полягає в тому, що активатори керованих ресурсів для зміни контексту повинні бути інтегровані в семантичний процес формування суджень;
- децентралізація логіки адаптації – передбачає використання децентралізованої системи управління. У разі, якщо логіка адаптації має бути розподілена, необхідно додатково визначити схему взаємодії;
- проактивна адаптація – мета полягає в тому, щоб провести адаптацію програмного забезпечення наперед, на основі передбачення. З точки зору користувача, цей підхід ефективніший, оскільки процес можна оптимізувати для конкретної послідовності подій.

Однією з проблем при розробці програмних систем є коригування функціональних можливостей, додавання нових функцій, модифікації графічного інтерфейсу шляхом зміни ресурсів або існуючих графічних елементів. При цьому ключовою особливістю адаптивних та самоадаптивних систем є використання розробленої архітектура програмного забезпечення на основі використання плагінів (plugin) та додаткових модулів [101].

Базова структура компонентно-орієнтованої архітектури програмної системи (рис. 1.2) складається з основної компоненти, програмних інтерфейсів для приєднання додаткових функціональних компонент та, власне, додаткових модулів.

Можливість динамічної зміни функціональних характеристик забезпечується завдяки програмним інтерфейсам, що дозволяють виконувати обмін інформацією та здійснювати взаємодію між модулями, знімаючи, при цьому, залежність від конкретної реалізації додаткових компонент системи.

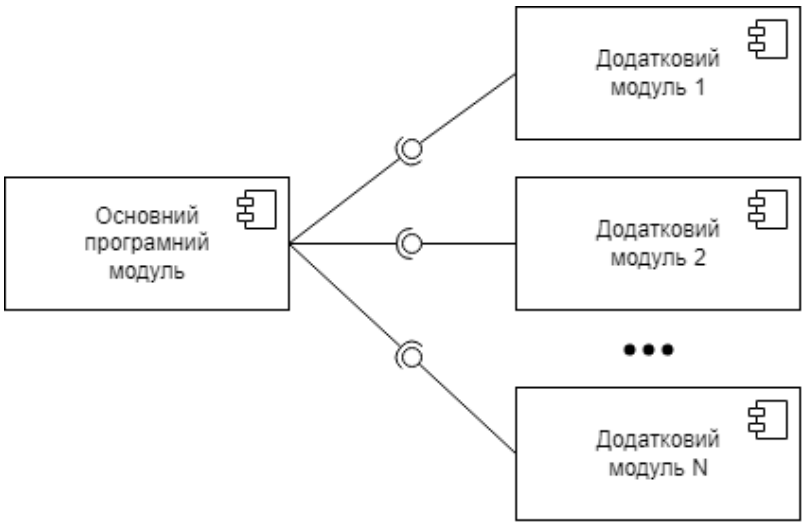

Рис. 1.2. Базова структура компонентно-орієнтованої архітектури програмної системи

Подане рішення є ефективним, оскільки забезпечує усунення високої зв'язності між основним та додатковими модулями системи. Крім того, така компонентно-орієнтована архітектура, за рахунок розподіленої структури, надає можливість виконання динамічної модифікації як елементів та ресурсів графічного інтерфейсу так і функціональних модулів без виконання реконфігурації та повторного налаштування програмної системи. Однак проєктування та використання plugin-архітектури вимагає попереднього визначення та реалізації додаткових структур та програмних інтерфейсів, що забезпечують механізм підключення нових модулів або сервісів. Одним із способів вирішення цієї проблеми є використання онтологічних моделей та динамічного завантаження програмних модулів.

1.1.2. Онтологічний підхід для розробки програмних систем

Онтологічний підхід під час проєктування програмної системи дозволяє в розгорнутому вигляді подавати семантичну модель предметної області, яка досліджується, у вигляді ієрархії концептів і множини відношень, які поєднують встановлені концепти і терміни [4, 42].

За своєю суттю онтологія є кортежем, що складається з множин: концептів, понять, атрибутів і відношень [2, 8]. Головною метою онтології є

класифікація індивідів (екземплярів) сутностей, що є фактичними елементами нижнього рівня представлення знань про предметну область. У якості понять обираються абстрактні набори або колекції об’єктів. Для зберігання специфічної інформації про об’єкт використовуються атрибути. Відношення дозволяють визначити залежності між об’єктами онтології. У результаті повного опису об'єктів і їх властивостей, предметна область буде представлена як складна ієрархічна база знань, над якою можна буде здійснювати «інтелектуальні» операції, такі як семантичний пошук і визначення цілісності та достовірності даних [3, 9].

Створення онтології базується на аналізі предметної області, що в загальному визначається послідовністю наступних етапів:

1. Визначення цілей і сфери застосування програми.
2. Проєктування загальної концептуальної структури предметної області. Даний етап передбачає визначення основних понять предметної області, їх властивостей, зв’язків між ними, а також створення абстрактних класів для підтримки наслідування властивостей і зв’язків, посилання чи включення допоміжних онтологій, віднесення екземплярів за концептами. Цей етап на даний час практично не піддається автоматизації, всі дії повинна здійснювати людина.
3. Збереження отриманої онтології як базової для подальшого розширення.
4. Розширення онтології предметної області шляхом додавання концептів, зв’язків та об’єктів до рівня деталізації, необхідного для забезпечення вимог, які ставлять перед онтологією, щоб використати її для розв’язування задач предметної області.
5. Верифікація створеної онтології та розгортання її в середовищі, де вона буде використовуватися.

Послідовне виконання цих кроків до проєктування онтологічних моделей допомагають формалізувати процеси, а також уникнути проблем і помилок на усіх етапах життєвого циклу програмного забезпечення: від початкового аналізу вимог (забезпечується комунікація та взаємодія між аналітиком та

клієнтом), у фазі проєктування та розробки (проєктуються формалізовані концепти, об’єкти та властивості елементів предметної області), а також на етапах впровадження та підтримки (забезпечення кращого розуміння запитів на модифікацію, вдосконалення та підтримки працездатності системи) [99, 75]. Враховуючи вказані особливості, рівні та сфери застосування онтологій, у наукових працях виділено цілий ряд класифікацій онтологічних моделей.

В першу чергу слід виділити класифікацію підходів до проєктування онтологічних моделей за рівнями залучення експертів домену [70]:

- Колаборативний підхід – полягає у активному залученні експертів баз знань, розробників та експертів домену на усіх етапах проєктування онтологічної моделі. Це забезпечуватиме безперервну взаємодію між усіма учасниками проєкту та формування узгоджених знань [36, 85].
- Підхід без співпраці – не виділяється чітка співпраця між учасниками розробки та зацікавленими сторонами, проте формалізовано процес, фази та завдання щодо побудови онтологічної моделі [54].
- Індивідуальний підхід – під час роботи не обов’язково визначаються формальні фази, завдання та процеси, але передбачається активне залучення фахової спільноти та використання спеціалізованих інструментів [15, 102, 109].

Інша класифікація підходів до проєктування онтологій полягає у визначенні сфери та рівня застосування створеної онтологічної моделі [19]:

- Онтологія верхнього рівня – визначаються найбільш загальні поняття та концепти, що є притаманними для усіх предметних областей. Такі онтологічні моделі забезпечують семантичну взаємодію з усіма іншими моделями нижчого рівня [28];
- Онтологія програмного процесу – даний тип моделей дозволяють формалізувати знання про різні моделі та методології життєвого циклу програмного забезпечення. Крім того під час проєктування у моделі визначаються конкретні фази розробки, їх залежність та їх послідовність [108];

- Онтологія домену – представляє знання та об’єкти конкретної предметної області та іншу допоміжну бізнес-інформацію, що необхідна у процесі розробки програмного забезпечення. Окрім об’єктів також ідентифікуються зв’язки, властивості та функціональні залежності, що допомагають краще формалізувати знання про необхідний домен [56, 95];
- Онтологія фази життєвого циклу – представляє та формалізує знання про конкретну фазу життєвого циклу програмного забезпечення: визначення вимог, проєктування, розробка, тестування, впровадження та підтримка. Дані моделі дозволяють не лише документувати знання про конкретні етапи, але й особливості використання методів, засобів та технологій на кожному з них [122, 20, 50]. Наприклад, під час проєктування онтології патернів – визначається інформація про існуючі шаблони проєктування, їх можливості, ризики та переваги використання. Це в свою чергу дає змогу прийняти рішення про їх впровадження в результуюче програмне забезпечення [53].

Вибір методології проєктування програмних систем на основі онтологічного моделювання обумовлено наступними чинниками. По-перше, створення програмного продукту на основі фіксованого набору вимог згідно традиційних методів проєктування програмних систем призводить до створення систем, в яких не передбачена реакція на зміну вимог та зовнішніх чинників [79]. По друге, врахування оновлених вимог, як правило, призводить до необхідності випуску нової версії програмного продукту, що передбачає виконання тривалих та ресурсоємних етапів аналізу, дизайну, кодування, тестування та впровадження нової версії.

Недоліки використовуваних модельно-орієнтованих підходів до побудови програмних систем значною мірою пояснюються складністю як предметної області, так і відповідних моделей. Ці недоліки можна усунути, реалізуючи програмну систему як набір простих інтерпретованих моделей, що взаємодіють між собою. Тому, створення програмних систем на основі онтологічного

моделювання в порівнянні з традиційними методами матиме наступні переваги:

- зникає необхідність проводити перекомпіляцію та повне розгортання програмної системи у випадку зміни бізнес-логіки. В такому разі необхідно модифікувати тільки ті моделі, що реалізують цю логіку;
- використання онтологічних моделей дасть змогу накопичувати та повторно використовувати знання про методи та варіанти адаптації програмної системи до зміни вимог.

1.1.3. Використання брокерів повідомлень при розробці адаптивних програмних систем

Використання окремих онтологічних моделей у процесі проєктування програмних систем, які формуються з різнотипних слабко зв’язаних компонентів, загострює проблему коректного обміну повідомленнями між різними сервісами. В такому разі доцільним є реалізація цього обміну на основі брокерів повідомлень, що дозволяють забезпечувати асинхронну комунікацію між сервісами. Реалізація такої взаємодії забезпечується можливістю використання методу публікації / підписки, що в свою чергу дає змогу постійно відстежувати та фільтрувати повідомлення, в залежності від сервісу [98].

Загальні принципи використання брокерів повідомлень та концепції service bus для модульного та адаптивного програмного забезпечення подано в роботі [16]. Автори зазначають, що архітектура програмного забезпечення та його компоненти відіграють життєво важливу роль у гнучкій, модульній та адаптивній системі. Запропоновано використання концепції service bus, для забезпечення можливості використання модульного та адаптивного програмного забезпечення, що дозволятиме виконувати керування модульними елементами. У результаті визначено перелік вимог до реалізації концепції модульного програмного забезпечення.

Окрім програмних систем, використання брокерів обміну повідомлень доцільне також і у об’єднаних програмно-апаратних комплексах, а також системах Інтернету речей. Велика складність їх реалізації пояснюється тим, що

керування middleware (проміжним програмним забезпеченням) стає неможливим через величезну кількість та різнотипність інформаційних сервісів і протоколів зв’язку [30].

Використання безпечного та розподіленого проміжного програмного забезпечення, що орієнтоване на повідомлення, для IoT додатків та розумних будівель наведено в роботах [74, 87]. Враховуючи існуючі виклики та проблеми створення таких програмних рішень, дослідниками запропоновано використання програмного забезпечення, що працює у режимі реального часу та орієнтоване на обмін повідомленнями, задля взаємодії та з’єднання інформаційних сервісів і фізичних пристроїв. Автори зазначають, що це дозволятиме масштабувати архітектуру програмно-апаратного комплексу незалежно від продуктивності пристроїв і обмежень підключення.

У роботі [30] авторами обґрунтовано важливість використання самоадаптивних систем у поєднанні з брокерами та чергами повідомлень, оскільки це дозволить покращити ефективність розроблюваних систем. Окрім того, у статті також подано запропоновані метрики масштабування, які дозволяють оцінювати час опрацювання/відгуку на певне повідомлення. Застосування даних метрик, у поєднанні з метриками використання ресурсів, дозволить краще оцінювати ефективність системи в цілому.

Метод використання самоадаптивної обробки даних для покращення процесу обслуговування динамічних систем подано в роботах [31, 63]. Автори досліджують проблему опрацювання даних під час динамічних навантажень на програмно-апаратну систему. Для досягнення кращих результатів механізм самоадаптації обробки даних інфраструктури модифікується шляхом доповнення до схем обчислення компонента обробки даних. Дослідники зазначають, що таке рішення дозволятиме мінімізувати втрати у процесі обслуговування та супроводу розроблюваних програмно-апаратних систем.

Таким чином, наявні дослідження та напрацювання дозволяють стверджувати про доцільність використання брокерів повідомлень у програмних системах з компонентно-орієнтованою архітектурою та слабко

зв’язаними компонентами для забезпечення асинхронної комунікації між сервісами. Проте залишається відкритим питання щодо підходів використання технології брокерів повідомлень у адаптивних програмних системах реального часу для коректного опрацювання даних, що актуалізує необхідність вирішення даного завдання.

1.2. Підходи до створення адаптивних інтерфейсів користувача

Одним із ключових і важливих завдань у процесі створення інтерактивних програмних систем є розробка інтерфейсу користувача. Дана компонента повинна коректно відображати доцільну інформацію та надавати доступ до функціоналу системи. Проте, під час проєктування програмної системи потрібно враховувати як різні користувачі використовують програмний засіб. Наприклад, для окремих класів користувачів необхідно обмежити доступ до певних ресурсів, а для інших, навпаки, потрібно надати доступ до всіх ресурсів. Враховуючи такі особливості, доцільним є використання моделей та методів для забезпечення можливості динамічної адаптації інтерфейсу до нових вимог або потреб користувача.

Концепцію створення контекстно-адаптивних інтерфейсів користувача на прикладі моделі програмної системи для комерційних транспортних засобів розкрито у роботі Л. Шелькопф, М. Вольф [103]. Автори зазначають, що використання контекстно-адаптивних інтерфейсів дозволяє виявити події, що повторюються найчастіше. На основі отриманої інформації стає можливою побудова різних варіацій інтерфейсу користувача та сценаріїв використання для основних «робочих фаз» водіїв комерційного транспорту. Незважаючи на загалом позитивну оцінку прототипу інтерфейсу, автори зазначають про необхідність вдосконалення наявної адаптивної моделі графічного інтерфейсу для підвищення ефективності роботи з системою.

Іншим прикладом проєктування адаптивного графічного інтерфейсу на основі контексту та досвіду користувача наведено у роботі Дж. Хусейн, А. Хасан, С. Лі [62]. У дослідженні автори представили нову методологію

побудови інтерфейсу користувача, що є незалежною від предметної області та пристрою, на якому виконуватиметься розроблене програмне забезпечення. Запропоноване рішення дозволяє модифікувати налаштування графічного інтерфейсу в залежності від контексту та інформації про обмежені можливості користувача та стан його здоров'я, враховуючи також рівень освітленості, шуму та інших факторів навколишнього середовища.

Окрім контексту під час адаптації враховується також і попередній досвід користувача, що отримується як явно – на основі зворотного зв'язку з особою, так і неявно – відбувається аналіз коректності виконання дій користувача у відповідь на зміни в системі. Такий рівень гнучкості під час адаптації забезпечується шляхом розбиття концептуальної моделі предметної області на три складові: «Користувач», «Пристрій», «Контекст» (рис. 1.3).

Рис. 1.3. Концептуальна трикомпонентна модель адаптації графічного інтерфейсу

Використовуючи ці моделі, інформацію про поведінку системи та зворотній зв'язок формуються правила адаптації інтерфейсу, що будуть інтерпретовані під час роботи програмного засобу. Проте дослідники

зазначають, що, зважаючи на механізм інтерпретації, наразі можливо додавати лише базові правила адаптації, що в свою чергу не дозволяє формувати остаточну версію графічного інтерфейсу [62].

Використання сучасних методів проєктування адаптивного вебінтерфейсу на основі мікросервісної архітектури програмного забезпечення подано у дослідженні А. Бугай та В. Олійник [1]. У роботі автори подали нову модель адаптивного інтерфейсу користувача, що, на відміну від стандартних підходів, дозволяє формувати правила модифікації графічного інтерфейсу, враховуючи зворотній зв’язок з користувачем та його попередній досвід.

Відповідно до дослідження автори виділили необхідність створення додаткових компонент, що забезпечують можливість динамічної зміни графічного інтерфейсу користувача. Реалізація запропонованої моделі передбачає використання таких модулів:

- модуль зберігання інформації про користувачів – забезпечує можливість зберігання та опрацювання інформації про користувача;
- модуль «інтерпретатор адаптацій» – дозволяє визначати необхідні параметри графічного інтерфейсу;
- модуль «рекомендаційна система» – використовує профіль користувача для визначення рекомендованих налаштувань;
- модуль з набором компонентів інтерфейсу користувача та модуль CSS стилів – містить елементи графічного інтерфейсу та їхнє представлення у вебзастосунку;
- модулі агрегації та допомоги роботи з системою – забезпечують можливість формування вебсторінки та зворотного зв’язку з користувачем.

Використання запропонованого підходу дозволяє спростити процес адаптації графічного інтерфейсу. У поєднанні з модулем «інтерпретатор адаптацій» даний підхід дозволяє формувати налаштування інтерфейсу відповідно до потреб користувача на основі зворотного зв’язку та попереднього досвіду.

Інший приклад використання адаптивного графічного інтерфейсу наведено у дослідженні О. Тищенко, Т. Онищенко та К. Писаренко [112]. У своїй роботі автори пропонують розширену модель доступності вебінтерфейсу для людей з обмеженими можливостями. В даному завданні дослідники виділяють чотири окремі цільові групи користувачів відповідно до їх порушень здоров'я: користувачі, що мають проблеми зі слухом; користувачі, що мають проблеми із зором; користувачі, в яких присутні розлади опорно-рухового апарату; користувачі з когнітивними порушеннями.

Відповідно до запропонованої розробки, стандартна модель формування графічного інтерфейсу, що містить правила оформлення посилань, тематичних блоків інформації, коректності відображення допомоги користувачу, розширюється завдяки впровадженню нових компонентів. Розширена модель додатково містить правила:

- використання атрибутів *alt* та *title*, а також надання субтитрів для медіа-контенту для кращого відображення графічних елементів;
- забезпечення можливості управління, використовуючи комбінації клавіш на клавіатурі;
- створення часових проміжків між виконанням дій над веб елементами;
- визначення прийнятних розмірів елементів;
- надання інформації про методи доступності з метою представлення наявних можливостей доступності для різних груп користувачів.

Дослідження показує, що використання запропонованої моделі дозволяє покращити ефективність адаптації вебзастосунків при проєктуванні графічного інтерфейсу для користувачів з попередньо визначеними типами порушень здоров'я. Проте, дана модель надає можливість модифікації графічних елементів лише для вебзастосунків. При проєктуванні програмного забезпечення для інших платформ будуть виникати труднощі у визначенні атрибутів графічного елементу для реалізації концепцій запропонованої моделі.

Ще одним засобом для динамічного формування та адаптації інтерфейсу користувача є використання методів формування логічного висновку та

Баєсових мереж [18]. Відповідно до запропонованого авторами підходу сформована Баєсова мережа відповідає за моделювання поведінки та взаємодії користувача з системою. Для валідації роботи спроєктованої Баєсової мережі визначено набір гіпотез, які дозволяють здійснити порівняння ефективності з методом *Frequentist Inference*. Як результат, це забезпечує кращу ефективність виконання адаптації, оскільки даний підхід дозволяє чіткіше визначити нові елементи, які потребують модифікації. Проте, незважаючи на представлені високі значення ефективності, автори зазначають, що запропонований метод є досить новим, що зумовлює необхідність його валідації у порівнянні з класичними методами адаптації інтерфейсу користувача.

1.3. Методи та технології динамічної адаптації функціональності програмної системи

1.3.1. Методи адаптації функціональності програмного забезпечення

Основою для забезпечення можливості динамічної адаптації функціональності програмних систем є використання компонентно-орієнтованої архітектури. Як вже зазначалося раніше, такий підхід дозволяє виділити основну компоненту програмного засобу та надати інтерфейс для приєднання додаткових модулів для розширення функціональних можливостей. Основні принципи використання компонентно-орієнтованої архітектури у поєднанні з політиками адаптації для реалізації самоадаптивної системи подано у роботі [35]. У дослідженні наведено підхід на основі автоматизованого генерування початкових налаштувань та станів, що використовуються адаптивною системою для ідентифікації процесу реконфігурації. Для генерування початкових конфігурацій автори запропонували використання комбінаторного алгоритму, що підтримує різні архітектурні елементи та зв’язки між ними, задовольняючи обмеження узгодженості, виражені інваріантами.

Іншим напрямом щодо створення самоадаптивних систем є використання діаграм зворотного зв'язку та керування, реалізація якого, зокрема, представлена в працях Філієрі, Мюллера [51, 113]. У роботі [113] дослідники

зазначають, що використання діаграм зворотного зв’язку (feedback-loop) є основою для динамічного управління поведінкою технічних систем у різних галузях. У результаті автори створили еталонну модель на основі циклу зворотного зв’язку, що забезпечила можливість зберігати та динамічно керувати налаштуваннями системи. Незважаючи на загальну ефективність запропонованого підходу, система додатково потребує визначення предметно-орієнтованої мови для повноти визначення об’єктів предметної області. Також, на нашу думку, запропоноване рішення не є універсальним, а лише еталонним, оскільки в роботі визначено систему в поняттях конкретної галузі застосування.

Дослідження процесу проєктування та розробки керованих систем на основі еталонних моделей відображено також у роботі [51]. Автори представили таксономію стратегій управління змінами, що дозволяють аналізувати їх застосовність для функціональних та нефункціональних властивостей. Проте, вважаємо, що не всі принципи та методи, які представлені у роботі можуть бути прямо застосовані для проблем адаптації програмного забезпечення без необхідності проєктування нових шаблонів чи структур даних.

Іншою проблемою, що виникає при створенні адаптивних та самоадаптивних систем є визначення вимог до програмного забезпечення. Проєктування вказаних систем вимагає ретельнішого опрацювання завдань та особливостей предметної області, оскільки, чим краще описані об’єкти предметної області, тим коректніше програма буде динамічно реагувати на зміну вимог.

Однак, використання стандартних підходів не завжди забезпечує можливість динамічної реконфігурації програмного забезпечення без необхідності повторного налаштування системи. Крім того, оновлення програмного засобу відповідно до нових вимог чи потреб користувача у стандартних модельно-орієнтованих методах буде означати повторне виконання етапів проєктування, розробки та тестування, що як вже зазначалося є ресурсозатратним процесом. Одним з ефективних рішень даної проблеми є

використання онтологічних моделей для представлення об’єктів та зв’язків в предметних областях. Такий підхід до створення програмного забезпечення не тільки дозволяє уникнути повторної конфігурації системи, але й надає можливість акумулювати знання про поведінку та налаштування програмного засобу.

Вирішенню проблеми адаптації програмної системи до змін середовища на основі використання онтологічного моделювання присвячені дослідження Є. Бурова, Б. Дуарте, В. Литвина [26, 79, 89]. На відміну від класичних підходів моделювання, онтологічний підхід дозволяє проєктувати формальну модель предметної області, яку можна змінювати та повторно використати під час розробки іншого програмного забезпечення.

Дослідження використання онтологічного підходу під час створення самоадаптивних систем розкрито у праці В. Дуарте, В. Соуза [89]. У своїй роботі автори представили комплексну онтологічну модель для самоадаптивних систем на основі вимог. В результаті дослідження автори провели експериментальну перевірку та верифікацію спроєктованої моделі на основі п’яти експертних питань:

- Що таке адаптивна система? – дане питання дозволяє ідентифікувати чи спроєктована система підпадає під концепції адаптивності;
- Які компоненти містить адаптивна система? – дозволяє визначити основні компоненти адаптивної системи та їх взаємодію;
- Як відбуваються процеси адаптації в адаптивній системі? – виконується опис основних процесів, що спрямовані на адаптацію програмної системи;
- Як визначається адаптивна система в специфікації вимог? – визначається зв’язок між процесами та компонентами адаптивної програмної системи та елементами специфікації вимог;
- Які вимоги задовольняє адаптивна система під час виконання? – демонструє зв’язок між виконуваними програмними компонентами та конкретними вимогами до програмного забезпечення;

Незважаючи на достатньо повний опис запропонованої онтологічної моделі предметної області, вона може використовуватися лише для підтримки та реінжинірингу адаптивних програмних систем на основі вимог.

Поєднання онтологічного підходу та методу адаптації програмної системи на основі моделей управління та зворотного зв'язку подано у праці К. Ангелопулоса, А. Пападопулоса, В. Соуза [13]. Використовуючи систему для планування зустрічей, автори дослідили процес адаптації програмних систем на основі зв'язків між динамічними вимогами та наявними адаптаціями програмного засобу. Запропонований метод заснований на використанні концептуальної моделі предметної області в поєднанні з конкретними показниками для визначення того, які зміни необхідно внести в систему. Проте, незважаючи на успішний процес адаптації конкретної системи, автори зазначають, що для покращення ефективності процесу адаптації слід розглянути реалізацію методу для різних предметних областей.

Незважаючи на загальну ефективність застосування онтологічного підходу при проєктуванні адаптивних програмних систем, як на нашу думку, наявні рішення не в повній мірі забезпечують динамічність модифікації функціоналу та інтерфейсу користувача. Також при створенні концептуальної моделі для програмного забезпечення враховується статична інформація про предметну область, що в свою чергу означає її неможливість швидкого розширення чи уточнення.

Інший підхід до динамічного формування та зміни компонент програмного забезпечення полягає у використанні нечітких множин та методів нечіткої логіки [5]. Методи адаптації функціональних можливостей на основі нечіткої логіки надають можливість динамічно визначити та опрацювати системні параметри відповідно до визначених правил. На відміну від класичної логіки при опрацюванні правил встановлюється ступінь істинності твердження, що дозволяє визначати бажані або необхідні налаштування відповідно до потреб користувача [39].

У порівнянні зі стандартними підходами побудова систем на основі

нечіткої логіки має кілька переваг:

1. можливість використання даних, що задано нечітко або інформації, яка динамічно змінюється в процесі роботи програмного забезпечення;
2. можливість проведення якісної та кількісної оцінки як вхідних даних так і отриманих результатів роботи;
3. можливість моделювання складних динамічних систем та проведення порівняльного аналізу, використовуючи певний попередньо заданий ступінь точності.

На дослідження використання методів нечіткої логіки при проєктуванні самоадаптивних програмних систем спрямована робота Янга, Тао та Хє [120]. Автори створили фреймворк, що дозволяє реалізувати процес самоадаптації програмного забезпечення на основі підходу нечіткого керування. Відповідно до запропонованого підходу процес розробки адаптивного програмного забезпечення відбуватиметься відповідно до визначених кроків:

1. Визначення вимог до нечіткої самоадаптації програмного забезпечення;
2. Створення нечітких змінних (вхідних та вихідних даних) для процесу нечіткої самоадаптації програмного забезпечення;
3. Редагування правил нечіткої самоадаптації програмного забезпечення;
4. Опрацювання правил нечіткої самоадаптації програмного забезпечення та аналіз отриманих результатів;
5. Створення нечітких аспектів самоадаптації програмного забезпечення;
6. Впровадження створених аспектів у програмне рішення.

Запропонована структура та принципи реалізації програмних систем на основі спроєктованого фреймворку дозволяють не тільки надавати зручний графічний інтерфейс для редагування та тестування стратегій нечіткої самоадаптації на зрозумілій предметній мові (DSL), але й автоматично перетворювати створені стратегії нечіткої самоадаптації у аспектно-орієнтовані

методи конкретної мови програмування.

1.3.2. Методи масштабування програмного забезпечення

Зростання та динамічність вимог до обчислювальних можливостей програмних систем та необхідність забезпечення варіативності використання програмного забезпечення призводить до збільшення навантаження на програмні компоненти. Складні програмні системи потребують постійного контролю за використанням ресурсів системи, оскільки у разі відмови обчислювального вузла можлива повна або часткова відмова у роботі програмного застосунку [17].

У результаті виникає проблема ефективного використання методів реагування та адаптації до змін мережевого трафіку, з урахуванням архітектури програмної системи [88]. Особливу увагу до зміни конфігурації та налаштувань програмної системи слід приділити у процесі проєктування адаптивного програмного забезпечення. Такі програмні застосунки потребують не тільки ефективного реагування на зростання кількості запитів на адаптацію, але й на відповідне збільшення затримки та часу формування модифікованої конфігурації програмного застосунку.

Враховуючи вказані проблеми, доцільно провести дослідження методів масштабування програмного забезпечення до зміни вимог та зростання мережевого трафіку для забезпечення кращого процесу формування адаптивної конфігурації програмного застосунку.

Ефективне реагування на збільшення мережевого трафіку полягає у застосування різних підходів та методів масштабування програмного забезпечення. Масштабованість програмної системи, водночас, визначає можливості опрацьовувати усі запити, що надходять без втрати продуктивності та без створення додаткових затримок. При цьому, масштабування програмної системи повинно враховуватися вже під час етапу проєктування. У разі некоректного проєктування архітектури програмного застосунку масштабування може потребувати додаткових змін (що спричинить затримки у

розробці або ж втрати на повторний випуск та вдосконалення програмного забезпечення на етапі впровадження) або може бути не застосовуваним, якщо система не відповідатиме поставленим вимогам [21, 59].

Масштабування програмного забезпечення не є уніфікованим в залежності від проблематики та вимог до збільшення пропускної здатності мережевого трафіку. Узагальнено методи масштабування ресурсів, на прикладі програмної-апаратної системи, можна поділити на два типи [60]:

- Горизонтальне масштабування – збільшення кількості фізичних ресурсів (рис. 1.4). При використанні цього методу масштабування до існуючої системи додаються нові обчислювальні вузли з однаковими або різними обчислювальними можливостями.

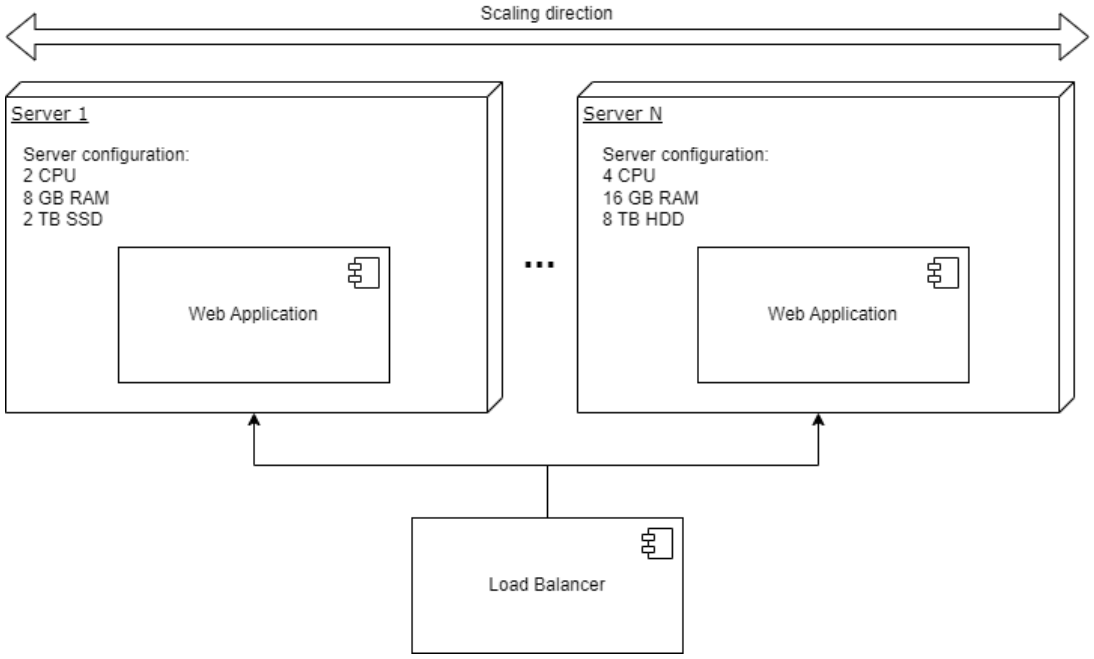

Рис. 1.4. Схема горизонтального масштабування програмно-апаратної системи

Проте, у такому випадку виникає необхідність використання спеціалізованих балансувальників навантаження, для рівномірного розподілу мережевого трафіку між створеними вузлами.

- Вертикальне масштабування – збільшення / покращення віртуальних та обчислювальних ресурсів (рис. 1.5).

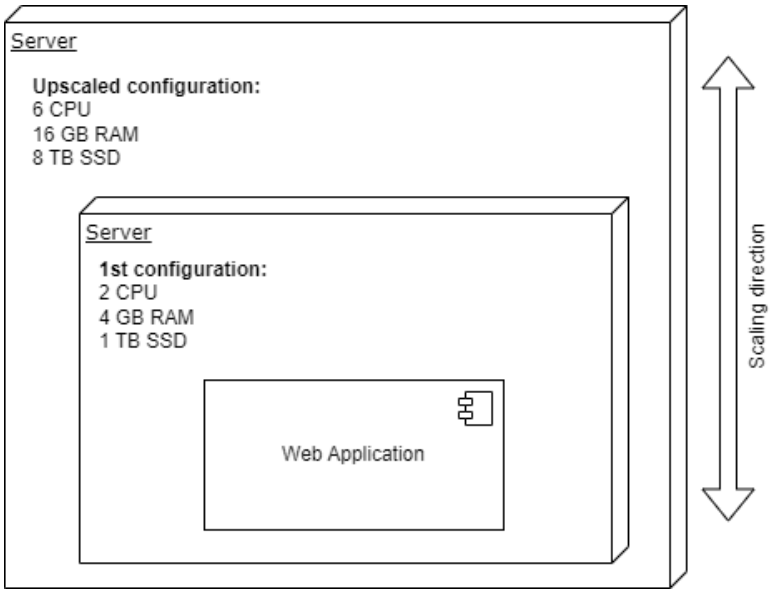

Рис. 1.5. Схема вертикального масштабування програмно-апаратної системи

Для реалізації цього методу передбачаються зміни апаратних властивостей відповідного серверу: покращення центрального процесору; збільшення кількості та розміру оперативної пам'яті; збільшення розміру пам'яті системи. Цей метод найчастіше використовується при неможливості застосування горизонтального масштабування.

Кожен з вказаних методів доцільно використовувати у ситуаціях, якщо архітектура програмної системи є компонентно-орієнтованою та дозволяє здійснити виділення та розподіл окремих сервісів, тоді доцільно застосувати горизонтальне масштабування. Окрім збільшення пропускної здатності, збільшиться час доступності програмного застосунку, оскільки у разі відмови одного вузла балансувальник навантаження планомірно перенаправить усі запити до робочих вузлів. Проте, у разі неможливості ефективного поділу функціональності або незначного збільшення кількості мережевого трафіку, тоді доцільно використати вертикальне масштабування, оскільки під час застосування цього методу збільшуються ресурси окремих компонентів [81].

Масштабування програмного забезпечення застосовується для проєктування програмних систем в різних прикладних галузях. Зокрема, у роботі [23] запропоновано фреймворк для розробки розподілених програмних систем для біоінформатики. Автори запропонували розроблену гнучку

платформу, що дозволяє покращити створення та розгортання багатоетапних робочих процесів, оптимізованих для високопродуктивних обчислювальних кластерів і хмар. При цьому, завдяки використанню принципів масштабування, програмне забезпечення дозволяє реалізувати можливість модифікації внутрішніх процесів до потреб дослідників.

Принципи масштабування та паралелізації ефективно використовуються у високопродуктивних та хмарних обчисленнях та системах. У дослідженні [82] автори зазначають, що сучасні підходи до аналізу великих даних потребують переходу на високопродуктивні програмні системи. Проте традиційні методи розпаралелення не забезпечують необхідної продуктивності. Враховуючи ці проблеми, у статті подано новий метод до масштабування програмних систем. Автори підкреслюють, що новий метод забезпечує не лише зменшення використання апаратних ресурсів але й значне пришвидшення роботи системи в процесі аналізу великих даних.

Переваги використання принципів масштабування також наявні під час виконання реконфігурації компонентів розподіленого програмного забезпечення [33]. У запропонованому аналізі автори зазначають, що одним з ефективних методів проєктування розподілених та сервіс-орієнтованих систем є використання компонентно-орієнтованої архітектури. Проте, у процесі розробки, окрім коректності реконфігурації, слід враховувати ще й час на опрацювання відповідних запитів.

Окрім класичних методів горизонтального та вертикального масштабування програмного забезпечення, для ефективної роботи системи доцільно також використовувати наступні підходи:

- Кешування – використання принципів кешування забезпечують зниження навантаження на бази даних та прикладні програмні інтерфейси за рахунок збереження раніше отриманої інформації [29];
- Балансування навантаження – розподіляє вхідний трафік між декількома серверами, щоб жоден сервер не перевантажувався [92];
- Шардинг бази даних – розділяє базу даних на менші, більш керовані

частини (шарди) для підвищення продуктивності та масштабованості [0];

Кожен із представлених методів дозволяє оптимізувати роботу програмного забезпечення за рахунок зменшення кількості запитів, зменшення часу на опрацювання або ж зменшення кількості даних, що опрацьовуються за один раз. Зокрема, у дослідженні [119] автори представляють тип кешування малих об’єктів, що дозволяє покращити роботу створеної системи за рахунок зменшення використання пам’яті. Крім того, зазначається, що використання такого підходу організації кешу використовує на 22-60% менше пам'яті, ніж найсучасніші конструкції для різних робочих навантажень. Проте, вважаємо, що принципи кешування доцільно використовувати коли результуючі дані оновлюються не з великою частотою або у разі, якщо застарілі дані не мають сильного впливу на подальшу роботу процесів ПЗ.

На нашу думку, хоч принципи кешування й дозволяють зменшити час опрацювання запитів, проте зі зростанням кількості користувачів проблема затримки мережевих запитів може відновитися. У таких випадках доцільно використовувати принципи балансування навантаження, яке має забезпечити не лише максимально можливу пропускну здатність запитів, але й забезпечити автоматичний їх розподіл між масштабованими сервісами [83, 104].

У роботі [115] автори зазначають, що балансування навантаження можна розділити на 2 основних методи: статичне та адаптивно-динамічне. Відповідно до зазначених підходів, статичний розподіл навантаження дозволяє збільшити пропускну здатність сервісів, при цьому не використовуючи додаткових складних алгоритмів [114]. Проте, у разі виконання складних операцій, для яких дані змінюються динамічно, доцільно використовувати адаптивний розподіл, що дозволяє постійно контролювати стан системи та коригувати розподіл ресурсів у режимі реального часу [14].

Враховуючи представлені проблеми та перспективи використання принципів та механізмів масштабування програмного забезпечення, виникає необхідність застосування комбінованих підходів до горизонтального

масштабування адаптивної програмної системи, що дозволить покращити швидкодію процесу визначення конфігурації та адаптації компонент, а також здійснити розподіл запитів за типом і критеріями адаптації.

1.4. Оцінка якості адаптивних програмних систем на основі онтологічного підходу

Підходи до оцінки якості адаптивного програмного забезпечення враховують різні аспекти роботи створеної системи. Враховуючи, що адаптивні та самоадаптивні програмні системи складаються з основної компоненти (основна поведінка та залежності системи) та додаткової функціональності (нові модулі, що розширюють можливість основної компоненти), аналіз якості та відповідності до вимог слід здійснювати як інтегрально так і поелементно. Відповідно до такої структури, підходи до оцінки адаптивних програмних систем можна поділити на дві основні групи [93]:

- Оцінка основного модуля – зосереджується на механізмах, процесах і продуктивності адаптації, що необхідно розглянути на ранніх стадіях процесу розробки. У цій категорії якість проєктування суттєво впливає на загальну якість і продуктивність програмної системи.
- Оцінка додаткових компонент – зосереджується на продуктивності програмної системи після адаптації та вимірюється під час виконання системи.

Однією з основних метрик оцінки процесу адаптації є визначення показника адаптивності програмного забезпечення. Метрика адаптивності не залежить від специфіки предметної області, орієнтована на поведінку адаптивної системи та дає кількісну оцінку адаптивності створеного програмного забезпечення [97]. При цьому, для використання метрики адаптивності попередньо потрібно визначити один або кілька показників ефективності для завдань, які реалізує адаптивна система. Ці показники відображають корисність адаптивної системи та дозволяють визначити виграш у використанні модифікованої конфігурації програмного забезпечення.

Метрика адаптивності є ефективним способом оцінки якості самоадаптивних та адаптивних систем, що реалізують механізм автоматичного моніторингу змін як частину процесу модифікації. Проте, якщо адаптація відбувається на основі статичного або динамічного набору правил, така метрика не дозволяє повністю охарактеризувати адаптивність створеного програмного забезпечення. В такому разі доцільно використовувати архітектурні та методологічні метрики.

У своїх дослідженнях Каддум та Райбулет [69, 68, 94] представили чотири категорії метрик:

1. методологічні – застосовуються до підходів розробки адаптивних систем на високому рівні абстракції;
2. архітектурні – фокусуються на зв'язку між основним компонентом та допоміжними модулями, а також на зростанні загальної системи за рахунок механізму адаптації;
3. внутрішні (метрики комплексності) – дозволяють оцінити складність процесу адаптації;
4. метрики часу виконання – застосовуються для оцінки часу формування та застосування конфігурації, а також навантаження на ресурси в процесі адаптації.

Такий поділ дозволяє здійснювати оцінку адаптивних програмних систем на усіх етапах життєвого циклу: від процесу формування вимог та проєктування до тестування ефективності процесу адаптації. Крім того, автори зазначають, що порівняння кількох механізмів адаптації потрібно здійснювати не лише на основі однієї метрики, але враховуючи кілька метрик разом. Це дозволить здійснити кращу оцінку роботи системи, оскільки незначні покращення однієї метрики можуть не повністю демонструвати якість розробленого адаптивного програмного забезпечення.

Іншу класифікацію архітектурних метрик адаптивних програмних системи розкрито у роботі Перес-Паласіна [90]. Автор демонструє використання модифікованих архітектурних метрик для програмних систем з

компонентно-орієнтованою архітектурою. Ці метрики дозволяють досліджувати процес адаптації з точки зору використання додаткових модулів.

Враховуючи загальну ефективність використання онтологічних моделей виникає необхідність перевірки та контролю якості створеної моделі. У процесі проєктування та розробки онтологій важливим є відслідковування її відповідності до визначеної мети. Для цього використовується певний набір критеріїв, які формують оцінку якості онтології [118]. Процес оцінки якості онтології є комплексним завданням, що полягає у визначенні ключових властивостей онтологічного графу та взаємодії між його вершинами.

Існуючі моделі якості програмного забезпечення зосереджені на дослідженні характеристик програмного коду, а також на процесах, що пов'язані з його розробкою, модифікацією, підтримкою та обслуговуванням. При цьому подані оцінки якості не можна застосувати безпосередньо до онтологічних моделей, оскільки вони принципово відрізняються від програм не тільки структурно чи функціонально, але і мають суттєві відмінності у процесах проєктування, розробки та підтримки [107, 123]. В таких випадках використовують уніфіковані методології якості, що базуються на існуючих критеріях, які визначені в інженерії онтологій у поєднанні з критеріями якості згідно теорії інженерії програмного забезпечення [117].

Враховуючи складність проєктування та розробки онтологічної моделі адаптивної програмної системи, для оцінки її якості доцільно використовувати спеціалізовані метрики, які, в загальному, можна поділити на дві групи: ієрархічні та реляційні. Перша група дозволяє оцінити онтології з точки зору ієрархії концептів предметної області та їх властивостей. Друга категорія дозволяє дослідити відношення між поняттями онтологічної моделі. Крім того, метрики також дозволяють здійснити аналіз конкретних аспектів онтології, оскільки, в більшості випадків, спосіб побудови онтології значною мірою залежить від сфери, в якій вона розроблена чи буде застосовуватися [107].

Процес оцінки якості спроєктованої онтологічної моделі доцільно виконувати відповідно до трьох основних груп метрик: структурних,

функціональних та метрик профілю зручності використання [41]. Ці групи метрик дозволяють проаналізувати онтологію відповідно до таких атрибутів як: синтаксична якість; відповідність правилам; насиченість і семантична якість; інтерпретабельність; послідовність; ясність і прагматична якість; вичерпність, точність та актуальність [76].

Аналіз характеристик онтологічної моделі слід розпочинати з базових структурних метрик, що дозволять у комбінації з метриками схеми визначити якісні атрибути створеної моделі. Початкова оцінка структурно-ієрархічних властивостей здійснюється завдяки метрикам глибини, ширини, заплутаності онтології та наявності циклів:

– Метрика абсолютної глибини (A_{depth}) – визначається як сума довжин всіх шляхів, де шляхом є послідовність вершин від кореневої до листової вершини графу онтологічної моделі:

$$A_{depth} = \sum_i^P l_i \,, \tag{1.1}$$

де: P – множина всіх шляхів онтологічного графу;

l_i – довжина і-ого шляху з загальної множини P.

– Середня глибина (Avg_{depth}) графу – визначає співвідношення абсолютної глибини онтологічного графу до загальної кількості шляхів:

$$Avg_{depth} = \frac{A_{depth}}{n_p}, \tag{1.2}$$

де: A_{depth} – абсолютна глибина онтологічного графу;

n_p – загальна кількість шляхів від кореневого елемента до листків.

– Метрика абсолютної ширини ($A_{breadth}$) – визначається як сума кількості класів на кожному рівні ієрархії онтологічної моделі:

$$A_{breadth} = \sum_i^L N_i \,, \tag{1.3}$$

де: L – множина всіх рівнів онтологічного графу;

N_i – кількість вершин на і-му рівні онтологічного графу.

– Метрика середньої ширини $(Avg_{breadth})$ – дорівнює співвідношенню абсолютної ширини онтологічного графу до загальної кількості рівнів ієрархії онтологічної моделі:

$$Avg_{breadth} = \frac{A_{breadth}}{n_L}, \quad (1.4)$$

де: $A_{breadth}$ – абсолютна ширина онтологічного графу;

n_L – загальна кількість рівнів ієрархії в онтологічному графі.

– Показник заплутаності онтології ($Tang$) – вказує на наявність множинного наслідування: чим менше значення заплутаності, тим краще спроєктована онтологія з точки зору когнітивної ергономіки. Метрика заплутаності визначається як відношення кількості класів, що мають більше ніж 1 батьківський клас до загальної кількості класів онтологічної моделі.:

$$Tang = \frac{t}{n_g}, \quad (1.5)$$

де: t – кількість вершин що мають більше однієї батьківської (множинний вхідний зв'язок is-a);

n_g – загальна кількість вершин в онтологічному графі.

– Показник циклічності онтології (A_{cycle}) – є індикатором наявності циклів, що використовується для визначення якості спроєктованої онтологічної моделі. Ступінь циклічності онтологічного графу визначається як відношення кількості циклічних залежностей / зв’язків до загальної кількості вершин у онтологічній моделі.

$$A_{cycle} = \frac{N_{cycle}}{n_g}, \quad (1.6)$$

де: N_{cycle} – кількість вершин що входять в будь який цикл онтологічного графу;

n_g – загальна кількість вершин в онтологічному графі G.

Таким чином, дана група метрик (1.1 – 1.6) забезпечує оцінку базових характеристик онтології, що на початкових стадіях проєктування дозволяє виявляти ергономічні та структурно-ієрархічні помилки.

Результатом процесу проєктування онтологічної моделі є створена таксономія понять предметної області. Оцінку якості утвореної таксономії доцільно здійснювати на основі метрик схеми (1.7) – (1.9), що включатиме наповненість екземплярів, глибини та ширини онтологічного графу, а також наявності множинного успадкування між концептами.

− Метрика насиченості атрибутів *(AR)* показує розподіл функціональних властивостей серед спроєктованих класів предметної області і визначається як відношення загальної кількості атрибутів онтології до загальної кількості концептів/класів онтології:

$$AR = \frac{|ATT|}{|C|}, \tag{1.7}$$

де: *ATT* – загальна кількість атрибутів онтології;

C – загальна кількість класів онтології.

− Метрика насиченості зв’язків (*RR*) характеризує розподіл властивостей, що встановлюють зв’язки між концептами онтологічної моделі серед спроєктованих класів предметної області. Насиченість зв’язків визначається як відношення загальної кількості зв’язків між об’єктами онтології до загальної кількості концептів/класів онтології:

$$RR = \frac{|P|}{|P| + |H|}, \tag{1.8}$$

де: *P* – кількість зв’язків, що не включають зв’язок is-a

H – кількість зв’язків спадкування (зв’язок is-a)

− Метрика співвідношення класів-атрибутів *(ACR)* визначається на основі співвідношення між класами, що містять властивості, і всіма класами у спроєктованій онтологічній моделі:

$$ACR = \frac{\left|N_{cl+att}\right|}{\left|C\right|}, \tag{1.9}$$

де: $|N_{cl+att}|$ – кількість класів, що містять властивості;

C – загальна кількість класів онтології

Базові метрики онтології та метрики схеми дозволяють оцінити структуру та зв’язки онтологічної моделі з метою визначення коректності побудови моделі предметної області. Проте дані метрики не дозволяють повністю оцінити інформативність та наповнення сутностей конкретними об’єктами. Враховуючи, що основною метою побудови онтологічних моделей є представлення знань про предметну область, подальше розширення бази знань дозволить оцінити якість онтології. Оцінку бази знань можна здійснити завдяки метрикам наповнення бази знань, які перевіряють як дані розміщені всередині онтології (1.10) – (1.11).

− Середнє наповнення класів (AP) – дозволяє оцінити розподіл екземплярів сутностей між усіма класами онтології на основі співвідношення загальної кількості екземплярів до загальної кількості класів онтологічної моделі,. Вище значення даної метрики вказує на краще наповнення бази знань. Середнє наповнення класів визначається як:

$$AP = \frac{\left|I\right|}{\left|C\right|}, \tag{1.10}$$

де: I – загальна кількість екземплярів сутностей;

C – загальна кількість класів онтології

− Метрика насиченості класів *(CR)* вказує на ступінь зв’язності концептів та екземплярів сутностей онтологічної моделі. Насиченість класів визначається як співвідношення кількості не пустих зв’язків до загальна кількість класів онтологічної моделі:

$$CR = \frac{\left|C_{non-empty}\right|}{\left|C\right|}, \tag{1.11}$$

де: $C_{non\text{-}empty}$ – кількість зв’язків, що мають хоча б один екземпляр

C – загальна кількість класів онтології

Таким чином, даний набір метрик дозволяє визначити синтаксичну та семантичну якість онтологічних моделей на основі оцінки базових характеристик онтології, а також оцінки якості утвореної таксономії та оцінки бази знань. Аналіз визначених метрик дозволяє виявляти ергономічні та структурно-ієрархічні помилки вже на початкових стадіях проєктування

1.5. Висновки до розділу

У розділі проведено аналіз методів та засобів проєктування адаптивних програмних систем. Визначено, що зі зростанням складності програмних систем ускладнюється процес модифікації функціональності та інтерфейсу користувача, що зумовлено необхідністю повторної конфігурації та інтеграційного тестування всієї системи. Вирішенням такої проблеми є використання методів адаптації та самоадаптації, що дозволяють змінювати властивості окремих компонент програмного забезпечення без необхідності повторної конфігурації та з урахуванням вимог конкретного користувача.

Проведений аналіз досліджень щодо підходів до проєктування та підтримки самоадаптивних систем підтвердив актуальність створення та вдосконалення методів адаптації програмних систем. Серед існуючих методів є характерним використання концептуальної моделі предметної області, яка дозволяє визначати зв’язки та співвідношення між новими вимогами до програмного забезпечення (ПЗ) та необхідними діями для адаптації функціональних та нефункціональних характеристик.

Встановлено, що основною проблемою під час такого процесу адаптації є статичність структури відповідної моделі, що, в свою чергу, не дозволяє вносити зміни в архітектуру чи враховувати конфігурацію та вимоги нових пристроїв, на яких буде встановлено відповідне ПЗ. Відповідно, доцільним є

використання узагальнених онтологічних моделей, що дозволять уніфікувати процес адаптації для програмних систем з різними предметними областями. Крім того, використання концепції самоадаптації при проєктуванні методу модифікації компонент програмної системи дозволить динамічно змінювати функціональність та графічний інтерфейс програмного засобу в залежності від вимог або потреб користувача.

Визначено групи метрик, що спрямовані на контроль якості адаптивної програмної системи та онтологічної моделі в процесі проєктування. Подані метрики дозволяють робити оцінку відповідності онтології до вимог з точки зору структури моделі, залежностей між компонентами а також наповнення бази знань. Крім цього, зміни значень метрик кожної групи дозволяють ідентифікувати як структурні недоліки спроєктованої моделі (наявність циклічних залежностей, наявність множинного наслідування), так і наповненість бази знань (насиченість атрибутів та зв’язків, наповненість класів).

РОЗДІЛ 2. МЕТОДИ АДАПТАЦІЇ ПРОГРАМНИХ СИСТЕМ НА ОСНОВІ ОНТОЛОГІЧНИХ МОДЕЛЕЙ

2.1. Особливості використання онтологічного підходу для створення адаптивних програмних систем

Онтологічний підхід до створення адаптивних програмних систем ґрунтується на принципах, що випливають з онтології та формального представлення знань у певній області. В процесі проєктування та створення онтологічної моделі проводиться концептуалізація об’єктів, зв’язків та екземплярів предметної області [43]. Це дозволяє представити об’єкти у більш абстрактному вигляді, що в свою чергу допомагає спростити проєктування та розробку об’єктів програмних систем [52].

В залежності від складності та типу інформаційних систем виділяють два види онтологій [55]:

1. Еталонна онтологія – дозволяє формувати найкращий опис предметної області в умовах реального використання, незалежно від її обчислювальних властивостей;
2. Операційна онтологія – гарантує бажані обчислювані властивості для програмних систем, що у свою чергу забезпечує краще автоматизоване опрацювання та редагування моделі.

Такий поділ онтологічних моделей дозволяє розподілити їх застосування між процесами проєктування програмних систем та використання моделей в процесі розробки та власне роботи програмної системи. Як на нашу думку, дані види онтологічних моделей дозволяють формувати опис для відносно невеликих предметних областей. У разі необхідності створення моделей для великих та складних доменів, зокрема тих що спеціалізуються на людино-машинній взаємодії, потрібно формувати окрему онтологічну мережу. В такому випадку буде забезпечуватися модульність системи, а зв’язки між окремими онтологічними моделями дозволять створити взаємопов’язані семантичні ресурси.

Процес проєктування онтології залежить від різних реалізацій компонентів певної предметної області [44]. В процесі проєктування важливим завданням є коректне визначення набору концептів та зв’язків, що в свою чергу дозволить якісно наповнити базу знань екземплярами та аксіомами. Окрім того, під час проєктування онтології слід переконатися, що результуюча модель відповідає характеристикам [27, 24]:

- модель повинна бути визначена в стандартизованій формальній мові (наприклад, OWL, RDF та ін.);
- концепції онтологічної моделі повинні виражатися через аксіоми;
- модель повинна відповідати попередньо сформованій специфікації вимог для відповідної предметної області.

Формалізовано процес проєктування та розробки онтологічної моделі можна поділити на кілька етапів [27]:

1. Вибір предметної області;
2. Визначення основних елементів предметної області та проєктування на основі них концептів та класів;
3. Визначення властивостей для кожного сформованого класу;
4. Створення екземплярів для спроєктованих концептів.
5. Проєктування зв’язків між концептами онтології та створення аксіом.

Проєктування онтологічної моделі відповідно до вказаного процесу дозволить контролювати її якість та характеристики. Окрім того сформована за таким методом модель забезпечить можливість в подальшому додавати семантичні правила. Наявність семантичних правил забезпечить виконання семантичного механізму прийняття рішень для отримання нових понять або семантичних зв’язків.

2.2. Розробка методу побудови моделі адаптивної програмної системи на основі онтологічного підходу

Однією з проблем проєктування та розробки адаптивних програмних систем є ефективне відображення елементів предметної області та їх

представлення в процесі розробки. Ефективним способом подання інформації про предметну область є проєктування онтологічної моделі програмної системи. Таке представлення дозволяє подати інформацію в структурованому вигляді та встановлює функціональні залежності між концептами та атрибутам об’єктів предметної області [48]. В стандартизованому вигляді онтологічна модель складається з трьох компонент:

1) концепти – відображають об’єкти предметної області;
2) відношення (властивості об’єктів) – представляють зв’язки між концептами онтологічної моделі;
3) функціональні атрибути – поля/атрибути, що розширюють інформацію про визначений об’єкт онтологічної моделі.

Проте, як на нашу думку, такий класичний метод проєктування онтології є ефективним лише для систем з чітко визначеними об’єктами предметної області. У випадку, коли онтологічна модель застосовується для адаптивних програмних систем її структура буде змінюватися відповідно до наповнення програмного забезпечення функціональними та графічними компонентами. Це погіршуватиме ефективність процесу адаптації, оскільки після додавання нових модулів потрібно додавати нові концепти та зв’язки між елементами, а для програмної системи робити повну реконфігурацію та повторне розгортання.

Дану проблему, яка виникає при використанні класичного підходу у проєктуванні онтологічної моделі для адаптивної програмної системи можна проілюструвати на основі прикладу програмного забезпечення для допомоги людям з когнітивними порушеннями (рис. 2.1). Відповідно до вимог таке програмне забезпечення повинно реагувати на зміни потреб користувача та враховувати різні види порушень здоров’я, зокрема: порушення зору, слуху, а також когнітивні зміни [48].

Отже, згідно з класичним підходом проєктування онтології, спершу виділяються об’єкти предметної області у вигляді концептів відповідної моделі: «Програмний модуль», «Тип порушення здоров’я», «Програмне забезпечення», «Інтерфейс користувача», та ін. Після створення об’єктів встановлюються

відповідні зв’язки, зокрема: uses – позначає зв’язок між користувачем та використовуваною конфігурацією ПЗ; need_element та need_module – позначає необхідний для встановлення або модифікації модуль ПЗ. Проте, подальші вдосконалення та зміни структури онтологічної моделі (додавання нових концептів онтологічної моделі – програмного модуля або елементу інтерфейсу користувача) спричинятиме розбіжності у схемі бази знань, що в свою чергу створюватиме проблему своєчасної та коректної міграції версій онтологічної моделі. Окрім цього, у випадку адаптації програмного забезпечення для кожного нового елемента потрібно створювати нове онтологічне правило, що теж спричинятиме проблеми на фазі підтримки програмного забезпечення.

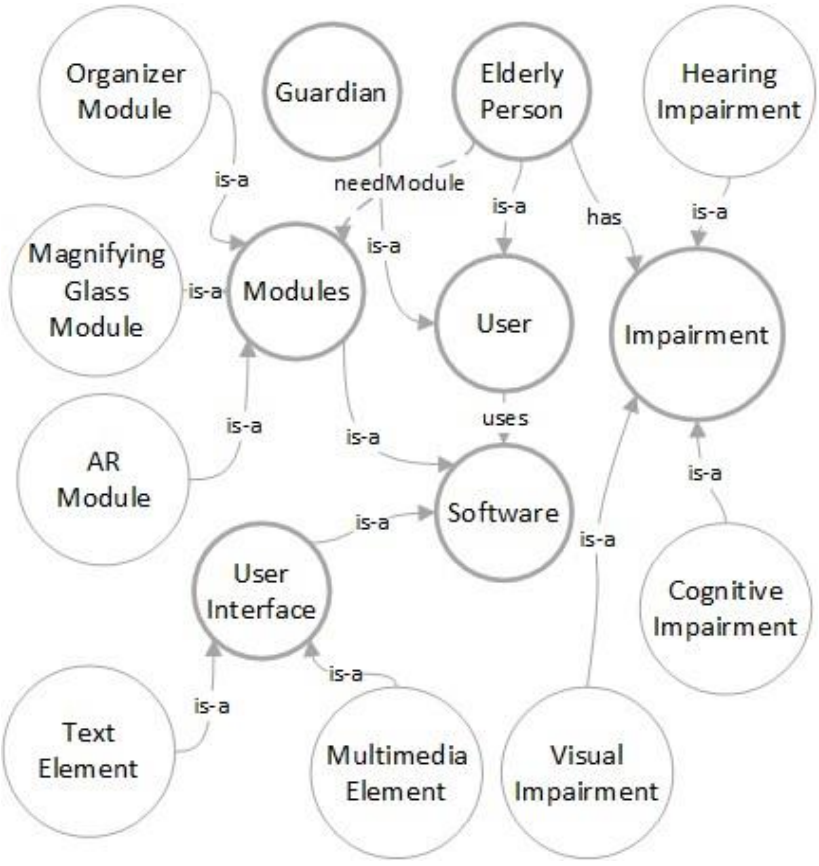

Рис. 2.1. Онтологічна модель адаптивної програмної системи на основі класичного підходу проєктування

Вирішенням проблеми адаптації на основі онтології є використання абстрактного підходу до проєктування онтологічної моделі [45]. Відповідно до абстрактного підходу під час створення концептів та зв’язків онтологічної

моделі між елементами виділяються абстракції елементів предметної області. Кожна з абстракцій може розширюватися за допомогою екземплярів сутностей, що в свою чергу забезпечує можливість динамічного додавання нових елементів до онтологічної моделі. У випадку адаптації програмного забезпечення виділяється три основних абстракції, що стосуються безпосередньо процесу адаптації: «Компонента», «Функціональна компонента» та «Графічна компонента». Відповідно до такого поділу основні зв'язки та правила адаптації встановлюються для концепту «Компонента» і поширюються на усі підкласи та екземпляри. Таким чином забезпечується динамічність процесу адаптації, без необхідності повної реконфігурації системи.

У формалізованому вигляді метаонтологію адаптивної програмної системи (рис. 2.2) можна представити як комбінацію трьох елементів [46]:

$$O_{meta} = \langle C_{meta},\ R_{meta},\ Prop_{meta} \rangle, \tag{2.1}$$

де: C_{meta} – множина сутностей предметної області адаптивної програмної системи, що містить інформацію про користувачів та можливі конфігурації програмного забезпечення;

R_{meta} – множина відношень між сутностями предметної області. До основних відношень / зв'язків адаптивних систем слід виділити:

1. *need configuration* – дозволяє визначити оптимальну конфігурацію компонент або системи в цілому для користувача;
2. *has requirement* – зв'язок встановлює набір вимог до програмного забезпечення, на основі яких буде здійснюватися аналіз та підбір необхідних компонент програмної системи;
3. *contains* – визначає набір функціональних та графічних компонент / модулів, що наявні для конкретної версії програмного забезпечення;
4. *requires solution* – зв'язок пов'язує конкретну вимогу користувача з її рішенням у вигляді програмних компонент.

$Prop_{meta}$ – множина властивостей, що характеризують сутності (поняття) предметної області.

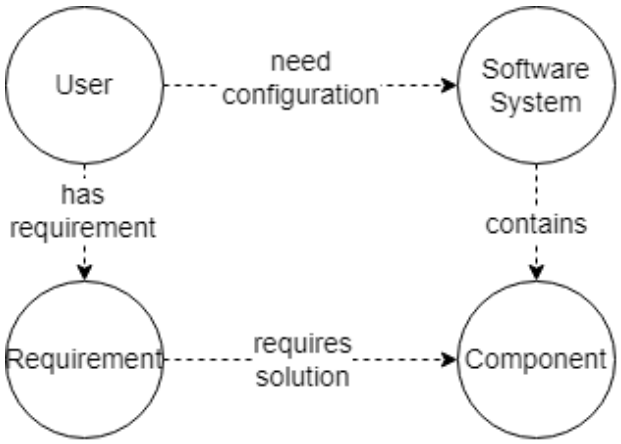

Рис. 2.2. Метаонтологія адаптивної програмної системи

Метаонтологія описує абстрактні компоненти програмної системи. Основними концептами метаонтології є:

1. *User (Користувач)* – концепт представляє основну інформацію про користувачів програмної системи;
2. *Software system (Програмна система)* – описує конкретну реалізацію програмної системи. Також цей концепт дозволяє відстежувати зміни версій програмної системи завдяки додатковим атрибутам;
3. *Requirement (Вимога)* – опис основних вимог до програмної системи з точки зору користувача. Окрім опису кожна вимога також визначає зв’язок з можливим рішенням проблеми у вигляді програмної компоненти;
4. *Component (Компонента системи)* – концепт описує інформації про функціональні та/або графічні компоненти програмної системи, з відповідною версією та залежностями.

Сформована метаонтологія адаптивних програмних систем містить основні концепти та зв’язки, що необхідні для визначення конфігурації програмного забезпечення. У деталізованому вигляді онтологічну модель адаптивної програмної системи наведено на рис. 2.3.

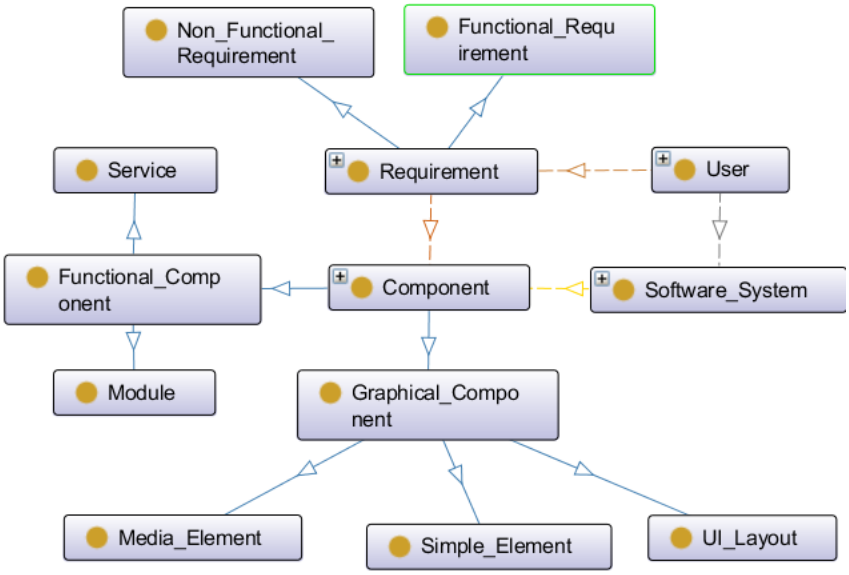

Рис. 2.3. Онтологічна модель адаптивної програмної системи на основі абстрактного підходу проєктування

Відповідно до створених моделей, представлених на рисунках 2.1 – 2.3, можна побачити що у абстрактному підході змінені зв'язки забезпечують більший рівень гнучкості. Наприклад, зв'язки «needModule» та «needElement» у класичній моделі було видалено та сформовано у абстрактній моделі уніфікований зв'язок «need configuration». Відповідні зміни забезпечують формування та збереження у базі знань різних конфігурацій програмної системи у залежності від наявності елементів у даній версії програмного забезпечення. Також це сприяє зменшенні кількості онтологічних правил, оскільки абстракції концептів та оновлені зв'язки дозволяють визначити єдине правило яке покриватиме усі можливі конфігурації (використовуючи зв'язки has requirement – для ідентифікації вимог користувача та requires solution – для визначення необхідних компонент).

Така структура дозволяє абстрагуватися від конкретної реалізації програмного забезпечення для різних предметних областей. Окрім цього, сформована модель надає можливість структурного розширення базових концептів шляхом використання процесу наслідування (зв'язок «has-a»), а також доповнення моделі новими зв'язками та правилами.

Для додаткового опису кожної із створених сутностей додано набір

властивостей, що описують відповідні характеристики.

Властивості сутностей формують ієрархію:

1. *ComponentDataProperties* – опис основних властивостей програмної компоненти:
    - *ComponentDownloadURI* – URI посилання на ресурс, що містить файли до завантаження програмних компонент;
    - *ComponentName* – назва програмної компоненти зареєстрованої в системі;
    - *ComponentVersion* – версія програмної компоненти, що доступна до завантаження;
    - *Component_UUID* – унікальний ідентифікатор програмної компоненти;
2. *FunctionalComponentDataProperties* – опис особливостей та властивостей функціональних компонент програмного забезпечення:
    - *AdditionalModulesURI* – URI посилання на ресурс, що містить файли для додаткової конфігурації функціональної компоненти;
    - *DataResource* – набір ресурсів та даних, які необхідні для роботи компоненти;
3. *GraphicalComponentDataProperties* – опис особливостей та властивостей елементів графічного інтерфейсу програмного забезпечення:
    - *AdditionalUIResourceURI* – URI посилання на ресурс, що містить файли для додаткової конфігурації графічного інтерфейсу;
    - *ResourceDictionary* – словник ресурсів та значень, що змінюють вигляд графічних елементів;
4. *SoftwareSystemDataProperties* – опис властивостей розробленої програмної системи:
    - *AssemblyVersion* – версія збірки програмного забезпечення;
    - *SoftwareName* – назва програмного забезпечення;
    - *SoftwareVersion* – версія програмного забезпечення;

- *Software_UUID* – унікальний ідентифікатор для активного програмного забезпечення;

5. *RequiremetsDataProperties* – опис властивостей та інформації про наявні вимоги до програмного забезпечення:
   - *LinkedDocumentLocation* – посилання на збережений документ, що описує конкретну вимогу;
   - *Req_UUID* – унікальний ідентифікатор вимоги для поточної реалізації програмного забезпечення;
   - *RequirementDescription* – короткий опис вимоги до програмного забезпечення;
6. *UserDataProperties* – опис інформації, що дозволяє ідентифікувати користувача в системі:
   - FullName – повне ім’я користувача, зареєстрованого в системі;
   - Role – роль/клас користувача в системі;
   - UUID – унікальний ідентифікатор користувача.

Сформовану діаграму класів отриману з онтологічної моделі предметної області, що враховує адаптивність компонент системи та графічного інтерфейсу користувача, наведено на рис. 2.4.

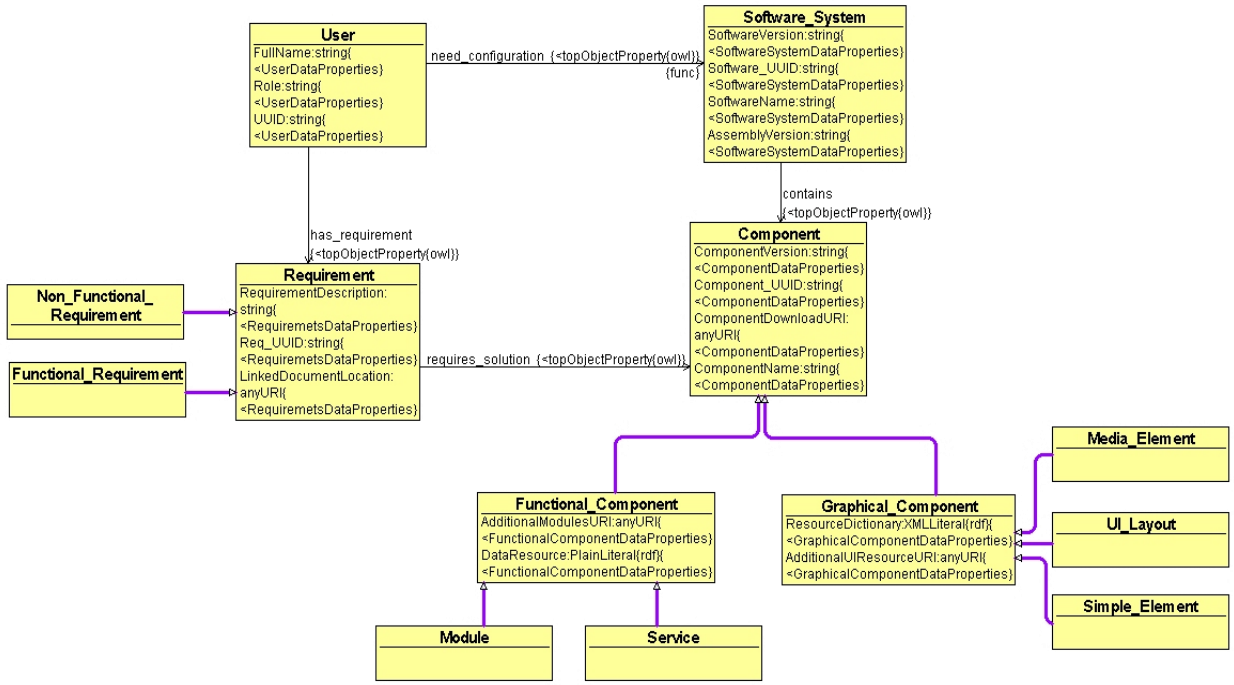

Рис. 2.4. Діаграма класів адаптивної програмної системи

2.3. Метод визначення налаштувань програмної системи на основі онтологічних правил та зв’язків

Онтологічні моделі дозволяють чітко та деталізовано представляти елементи предметної області використовуючи механізм формалізації та асоціації даних. Таке відображення сприяє використанню принципів онтологічного моделювання та технологій семантичних мереж у різних галузях: медицина, освіта, людино-машинна взаємодія, розробка програмного забезпечення та ін. Проте, незважаючи на загальну ефективність побудованих моделей, складність отримання та формалізації інформації для кожної галузі є різною, що в свою чергу викликає необхідність залучення експертів домену. Це в свою чергу ускладнює проєктування інформаційних систем, оскільки більшість доменних експертів не мають досвіду розробки програмного забезпечення, що створює нову проблему комунікації [22, 38].

Ефективним вирішенням такої проблеми є використання семантичних правил для побудови логічних зв’язків між компонентами онтології. Цей механізм дозволяє забезпечити визначення як функціональних так і причинно-наслідкових зв’язків між концептами та / або властивостями спроєктованої онтологічної моделі. Як результат під час побудови онтології предметної області доменні експерти матимуть змогу визначати не лише об’єктні властивості але й реальні правила опрацювання даних, що в подальшому можуть бути використані в процесі роботи програмного забезпечення [32, 37, 96].

Реалізація семантичних правил для онтологічних моделей відбувається завдяки механізмів дескриптивної логіки. Це дозволяє удосконалювати та формалізувати існуючий опис спроєктованих концептів та властивостей. Основою для механізму дескриптивної логіки у онтологіях є діалект OWL DL, що розширює стандарт опису OWL та забезпечує максимальну виразність, при цьому не втрачаючи обчислювальну повноту [25]. У свою чергу діалект OWL DL накладає деякі обмеження, зокрема:

- екземпляр повинен належати лише одній сутності (не зважаючи на

кількість батьківських концептів);

- поділ типів (наприклад, елемент не може бути одночасно екземпляром та властивістю);
- конкретизація домену та типів даних для властивостей.

Проте, незважаючи на особливості та переваги використання діалекту OWL, під час проєктування синтаксис залишається низькорівневим, що ускладнює його використання експертами домену. Для уникнення цією проблеми під час проєктування онтологічних доцільно застосовувати мову правил семантичної мережі (Semantic Web Rule Language – SWRL). SWRL використовується для створення логічної імплікації між тілом правила та його висновком, що складаються з окремих атомів. В свою чергу, атоми в SWRL правилі представлені як описи сутностей та їх властивостей, що складаються зі змінних, екземплярів сутностей чи, власне, даних [57, 61].

Загальна структура SWRL-правила складається з двох частин – атомів-аксіом та висновків:

$$atom1(?x) \wedge atom2(?y) \wedge atom3(?x, y) ... \Rightarrow consequent(?x, y) \wedge ... \qquad (2.2)$$

Опрацювання атомів та висновків відбувається послідовно. Після кожної операції виконується булева операція «і», яка комбінує результати тільки в разі, якщо попереднє значення було «істина». Такий метод дозволяє в процесі опрацювання ігнорувати концепти та екземпляри сутностей, які не підпадають під спроєктовану умову відповідного правила.

SWRL розширює можливості мови OWL DL та має кілька реалізацій:

- інтерпретація SWRL в логіку першого порядку (Hoolet) і демонстрація процесу опрацювання семантичних правил;
- інтерпретація OWL-DL в правила та подальше опрацювання в розширеному рушії правил Bossam;
- розширення існуючого механізму опрацювання семантичних правил – OWL-DL на основі алгоритму Tableaux (Pellet)

Враховуючи особливості проєктування онтологічних правил входом для процесу визначення оптимальних характеристик програмного забезпечення є вимоги певного користувача. Кожна вимога, що зареєстрована в системі, повинна посилатися на відповідне рішення у вигляді функціональної та/або графічної компоненти системи. У разі, якщо вимога, на даний час, не має вирішення вона не буде враховуватися у процесі адаптації.

Визначення характеристик системи відбувається на основі зазначених в онтологічній моделі класів, зв’язків та властивостей. Відповідно до створеної моделі, користувач буде отримувати актуальну конфігурацію програмного забезпечення в залежності від зміни вимог або потреб.

Інтегрування даного процесу дозволяє динамічно змінювати вміст та функціональність програмного забезпечення враховуючи попередньо визначені умови та тригери онтології. В узагальненому випадку процес визначення динамічних характеристик програмного забезпечення на основі SWRL можна поділити на такі етапи:

1. Визначення структури та наповнення онтології програмного забезпечення:
    a. Створення онтології за допомогою редактора онтології, наприклад Protege або TopBraid Composer;
    b. Визначення класів, індивідуумів, властивостей та зв’язків всередині створеної онтології.
2. Формування вимог до адаптації:
    a. Чітко визначити вимоги до адаптації, зрозуміти умови, за яких онтологічні правила необхідно додати, модифікувати або адаптувати
3. Створення SWRL-правил адаптації
    a. Створення правил SWRL, які фіксують умови та дії адаптації. Правила SWRL складаються з атомів, що позначають передумови, істинних значень та відповідні дії/наслідки після виконання усіх умов.

Реалізація онтологічних правил для адаптації програмного забезпечення є різною для класичного та абстрактного підходів до проєктування онтологій. У класичному підході кожна сутність предметної області виділена окремим класом та відповідним об’єктом у базі знань та онтологічному графі. Таке проєктування спонукає до визначення кількох правил адаптації у залежності від типу компоненти, а також вимог користувача.

Наприклад, для забезпечення доступу до модуля організації задач опікуну та особі з когнітивними порушеннями потрібно сформувати наступне правило:

$$\begin{array}{l} User(?user) \wedge Software(?software) \wedge uses(?user, ?software) \wedge \\ OrganizerModule(?organizer) \Rightarrow needModule(?user, ?organizer) \end{array}, \quad (2.3)$$

де: *?user, ?software, ?organizer* – змінні SWRL-правила;

uses, needModule – відношення між сутностями предметної області;

У випадку якщо користувачеві необхідний доступ до нових текстових налаштувань, на основі даних про користувача з порушенням зору, потрібно створювати нове правило, що передбачає використання даних про порушення зору користувача:

$$\begin{array}{l} ElderlyPerson(?\ person) \wedge VisualImpaiment(?impaiment) \\ \wedge (has(?\ person, ?impairment) \wedge Vision(?impairment, ?vision) \\ \wedge swrlb:greaterThan\ (?vision, 5) \wedge swrlb:lessThan\ (?vision, 8) \\ \Rightarrow needElement(?\ person,\ cog:BigText), \end{array}, \quad (2.4)$$

де: *?person, ?impairment, ?vision, ?mglassModule, ?textElement* – змінні SWRL-правила;

swrlb:greaterThan, swrlb:lessThan – вбудовані логічні операції SWRL-правила;

has, needElement – відношення між концептами предметної області;

BigText – екземпляр концепту TextElement.

У випадку використання абстрактного підходу до побудови онтологічних моделей основні концепти та класи виділені у абстракції з загальними атрибутами. Це дозволяє застосувати правило для всіх дочірніх елементів, без необхідності сильного зв'язування компонентів. Приклад правила адаптації для визначення конфігурації програмної системи на основі користувацьких вимог та наявних компонент програмного забезпечення наведено у формулі:

$$\begin{aligned} &User(?user) \wedge Requirement(?reg) \wedge Component(?comp) \\ &\quad \wedge Software\_System(?sys) \wedge has\_requirement(user, ?reg) \\ &\quad \wedge requires\_solution(?reg, ?comp) \wedge contains(?sys, ?comp) \\ &\quad \Rightarrow needconfiguration(user, ?sys), \end{aligned} \quad , \quad (2.5)$$

де: *?user, ?req, ?comp, ?sys* – змінні SWRL-правила, що позначають, відповідно, сутності *Користувач, Вимога, Компонента та Програмна Система*;

has*_requirement, requires_solution, contains, need_configuration* – відношення між сутностями предметної області;

4. Інтегрування правил в онтологію
    a. Первинне інтегрування правил здійснюється за допомогою імпорту записів в форматі SWRL у попередньо спроєктовану онтологію. Більшість редакторів онтології забезпечують механізми для включення або посилання на правила SWRL у файлі онтології;
    b. Після імпорту правил необхідно визначити яким семантичним механізмом прийняття рішень будуть опрацьовуватися результуючі правила адаптації. Використання таких механізмів (наприклад, Pellet, HermiT) дозволить динамічно формувати нові знання та нову адаптивну конфігурацію програмної системи.
5. Валідація, тестування та підтримка програмного забезпечення:
    a. Валідація та тестування створених правил відбувається на наборі тестових даних та з використанням семантичних механізмів прийняття рішення. Це забезпечить перевірку правильності

використання концептів, зв’язків та екземплярів онтологічної моделі у процесі опрацювання правил адаптації;

b. Окрім процесу перевірки коректності онтологічних правил також необхідно проводити моніторинг показників якості та здійснювати модифікацію онтології (наприклад, створення нових екземплярів, встановлення зв’язків між елементами, оновлення SWRL правил, та ін.) відповідно до зміни вимог до конкретного модуля або програмної системи у цілому.

Використання та дотримання цих кроків до проєктування та реалізації онтологічних правил забезпечить можливість використання SWRL правил у процесі адаптації програмного забезпечення. Крім того, цей процес дозволить гарантувати, що онтологія залишається сумісною та узгодженою з змінними вимогами.

Проте, використання та опрацювання SWRL правил під час адаптації програмного забезпечення може бути ресурсозатратним процесом. Збільшення кількості таких елементів онтології як концепти, зв’язки та властивості суттєво впливає на час роботи семантичного механізму прийняття рішень. Крім того, більшість рушіїв опрацьовують семантичні правила у однопотоковому та послідовному (обробка одного твердження за раз) режимі. Така реалізація також сприяє збільшенню часу опрацювання онтологічних правил та використання ресурсів системи.

Вирішенням цієї проблеми може бути поділ однієї онтології на кілька частин. В такому разі кількість елементів онтологічної моделі суттєво зменшиться, що сприятиме покращенню часу опрацювання правил. Поділ процесу опрацювання онтології можна здійснювати за допомогою методів:

- Вертикальне масштабування – додавання ресурсів до системи, щоб продуктивність системи відповідала попиту;
- Горизонтальне масштабування – розподіл єдиного файлу онтології на частини з метою подальшого опрацювання у багатопотоковому та паралельному режимах.

У випадку адаптивного програмного забезпечення поділ можна здійснити у два підходи:

1. На основі типу компонент – виділяються унікальні об’єкти за типом компоненти: функціональна та графічна. При такому поділі дублюється інформація про систему та користувачів, проте у окремому файлі онтології міститиметься лише інформація про функціональні або графічні компоненти;
2. На основі типу системи – виділяються унікальні об’єкти системи використовуючи її тип. Наприклад, одним з типів класифікації є поділ за середовищем виконання: мобільний застосунок, вебзастосунок та настільний застосунок. В разі такого поділу кожен окремий файл онтології міститиме лише інформацію про конкретний застосунок та наявні для нього компоненти.

Проте, у визначених методів горизонтального масштабування є ряд недоліків. У випадку розподілу онтології на основі типу компонент виникає проблема у синхронізації даних між частинами, оскільки інформація про користувачів дублюється. Крім того, якщо система спрямована на адаптацію лише функціональних або лише графічних компонент, то проблема високого використання ресурсів системи та повільного процесу опрацювання правил залишиться.

Недоліки синхронізації та використання ресурсів вирішуються використанням поділу на основі типу системи. Такий метод зберігає актуальну інформацію про користувачів та елементів системи та дозволяє робити опрацювання правил на компактному наборі елементів. На нашу думку, недоліком даного методу може бути розподіл онтології на велику кількість файлів-моделей, якщо адаптивне програмне забезпечення підтримує кілька типів систем. Це ускладнить процес визначення необхідної частини онтології в процесі адаптації. Проте, цей недолік можна виправити використанням механізму брокера повідомлень, що також сприятиме покращенню продуктивності процесу адаптації.

2.4. Метод динамічної адаптації функціональності програмної системи та інтерфейсу користувача

Розроблена онтологічна модель предметної області для адаптивних програмних систем дозволяє визначати необхідні дії для реагування на зміни у вимогах чи потребах користувача. Проте дана модель є лише частиною процесу генерації програмних налаштувань. Функціональна модель спроєктованого процесу динамічної адаптації програмної системи представлена на рис. 2.5. На верхньому рівні ієрархії наведено взаємодію загального процесу адаптації програмного забезпечення із зовнішніми сутностями.

Основними зовнішніми об’єктами для процесу слугують «Користувач» та «Активний пристрій». Власне для ініціації процесу зміни налаштувань користувач надсилає відібрану інформацію, яка слугує ідентифікатором про необхідні зміни в роботі чи вигляді системи. В процесі опрацювання система робить зворотній запит, щодо інформації про активний пристрій, яка допоможе ідентифікувати можливість змін конкретних характеристик. Сформовані налаштування враховуються під час процесу модифікації компонентів системи, а користувачу представляються адаптовані характеристики.

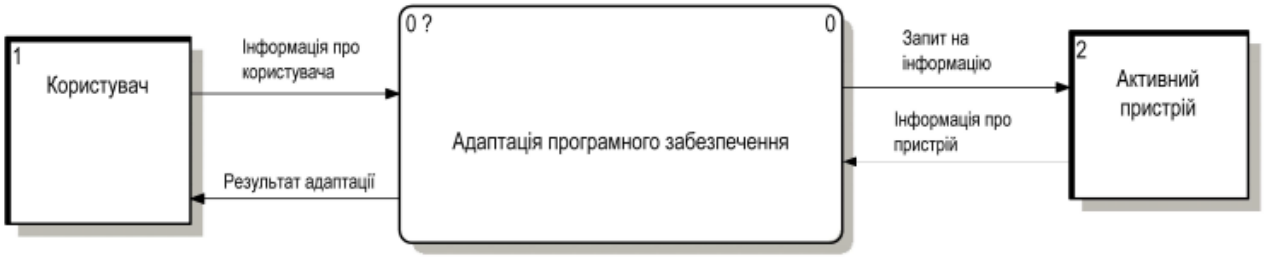

Рис. 2.5. Схема процесу адаптації програмної системи

Відповідно до діаграми можна побачити, що система отримує інформацію про користувача та надає користувачу результати модифікації в залежності від специфікації активного пристрою.

Для складних систем будується ієрархія контекстних діаграм. При цьому

контекстна діаграма верхнього рівня містить набір підсистем, з'єднаних потоками даних. Оскільки процес адаптації програмного забезпечення комплексний та містить в собі набір підпроцесів, тому необхідно зробити його декомпозицію (результат декомпозиції процесу наведено на рис. 2.6.). Відповідно до поданої декомпозиції процесу «Адаптація програмного забезпечення» інформація отримана від користувача опрацьовується та синхронізується з базою знань. Після синхронізації відбувається генерація налаштувань, що дозволяє здійснити адаптацію програмного продукту.

Виходячи із вказаного процесу, метод динамічної адаптації функціональності програмної системи та інтерфейсу користувача складається з таких етапів:

1. Опрацювання інформації, отриманої від користувача:
    a. ідентифікація користувача у системі та відповідна реєстрація у разі відсутності даних у базі знань;
    b. перевірка коректності заповнення інформації про конфігурацію програмного забезпечення, що включає: набір нових потреб користувача на основі наявних вимог до ПЗ;
2. Синхронізація даних між базою даних та онтологічною базою знань – даний процес дозволяє створити або ж модифікувати необхідні записи про користувача, а також про поточну конфігурацію використовуваного ПЗ;
3. Генерація налаштувань – етап опрацьовує дані про обраного користувача та формує необхідні налаштування системи, базуючись на зв’язках та властивостях концептів онтології:
    a. Опрацювання та генерація налаштувань здійснюється за допомогою методу визначення оптимальних характеристик відповідно до створених онтологічних правил;
    b. У разі успішної генерації – формування формалізованого представлення нової конфігурації ПЗ на основі модифікованих концептів та екземплярів онтологічної моделі;

4. Адаптація програмної системи включає модифікацію функціональних характеристик та графічного інтерфейсу на основі набору сформованих налаштувань.

Рис. 2.6. Діаграма потоку даних процесу адаптації програмної системи

Відповідно до зображеного процесу, онтологічна модель використовується в двох ключових етапах: синхронізація даних та генерація налаштувань для ПЗ. Перший етап дозволяє зберігати відібрану інформацію про користувача в онтологічній базі знань з метою полегшення подальших запитів щодо модифікації характеристик системи. Другий етап власне запускає семантичний механізм суджень, який починає опрацювання семантичних правил, що визначені в онтологічній моделі. Ці два етапи є взаємопов’язані оскільки результат синхронізації даних між базою даних та базою знань відповідає за коректне наповнення бази знань тільки оновленою інформацією.

Процес «Адаптація програмного забезпечення» показує загальну комунікацію між підпроцесами та сховищами даних від початку надходження запиту до формування відповіді. З метою деталізації процесу модифікації функціональності та графічного інтерфейсу користувача здійснено декомпозицію процесу «Адаптація програмного продукту» (рис. 2.7).

Результатом декомпозиції є діаграма потоків даних, що містить детальний опис послідовності передачі даних між елементами системи для процесу динамічної адаптації існуючого програмного рішення.

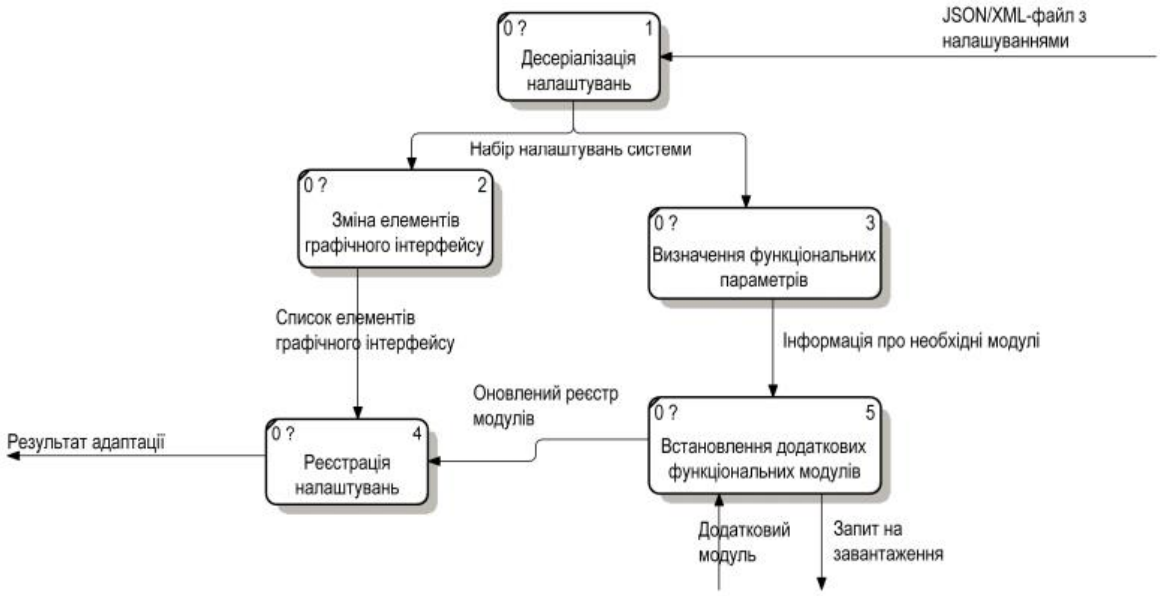

Рис. 2.7. Схема декомпозиції процесу А4 «Адаптація програмного продукту»

2.5. Висновки до розділу

У розділі розроблено методи адаптації програмних систем на основі онтологій. Визначено принципи використання онтологічного підходу та поетапність його реалізації, що дозволяють забезпечити виконання семантичного механізму прийняття рішень в динаміці для застосування моделі у разі отримання нових понять або семантичних зв’язків.

Розроблено метод побудови моделі адаптивної програмної системи на основі онтологічного підходу, що дозволяє представити основні елементи та зв’язки без прив’язки до структури предметної області. Визначені концепти та поняття дозволяють сформувати SWRL-правила для визначення оптимальної конфігурації системи в залежності від вимог до програмного забезпечення.

Розроблено метод визначення налаштувань програмної системи на основі онтологічних правил та зв’язків. Метод дозволяє формувати інформацію про адаптивні налаштування системи з використанням знань онтологічної моделі.

Використання абстрактного підходу до проєктування онтології у поєднанні з семантичним механізмом міркувань забезпечує можливість динамічної адаптації програмної системи для різних платформ. Застосування горизонтального масштабування дозволить розподілити навантаження на систему у процесі визначення нової конфігурації.

Удосконалено метод динамічної адаптації функціональності програмної системи та графічного інтерфейсу, що враховує уподобання та вимоги конкретного користувача, а також особливості поточно активного пристрою. Спроєктований процес використовує онтологічну модель, як базу знань про вимоги та структуру програмного забезпечення, та дозволяє визначати необхідні налаштування системи на основі семантичного механізму прийняття рішення. Також, процес зміни конфігурації дозволяє проводити модифікацію як вже існуючих компонент, які зареєстровані в програмному середовищі, так і додавати нові функціональні та графічні елементи для розширення можливостей програмного забезпечення.

РОЗДІЛ 3. АРХІТЕКТУРА ПРОГРАМНОЇ СИСТЕМИ ТА АЛГОРИТМІЧНА МОДЕЛЬ ПРОЦЕСУ ДИНАМІЧНОЇ АДАПТАЦІЇ

3.1. Основні рівні архітектури адаптивної програмної системи та їхні компоненти

Основою для реалізації методу адаптації програмних систем ми пропонуємо використання онтологічного підходу. Першочерговим завданням є визначення специфікації вимог:

1) наявність онтології, що дозволяє зберігати архітектуру програмних продуктів;
2) адаптація до специфіки предметної області з урахуванням контексту прикладних завдань;
3) надійність зберігання та швидкодія опрацювання даних;
4) наявність механізму логічного висновку, що забезпечуватиме опрацювання онтологічних правил та знань, а також формування налаштувань програмного забезпечення;
5) наявність механізму адаптації інтерфейсу для користувачів;
6) наявність механізмів імпорту даних із зовнішніх інформаційних ресурсів.

Для забезпечення виконання визначених вимог необхідно спроєктувати структуру сервісу синхронізації даних та знань (рис. 3.1). Оскільки онтологічна модель за своїм призначенням зберігає знання про всю предметну область, перед опрацюванням онтологічних правил необхідно пересвідчитися чи база знань містить оновлену інформацію. Проте, оскільки формат зберігання даних у базі знань та базі даних суттєво різниться один від одного, тому виникає проблема коректного співставлення полів та значень.

До вирішення проблеми синхронізації даних між онтологічною моделлю та базою даних використовують такі підходи:

- створення спеціалізованого засобу, що буде виконувати пряме співставлення об’єктів онтологічної моделі та елементів бази даних;

- використання принципів об’єктно-орієнтованого програмування та особливостей об’єктно-реляційного співставлення для створення єдиного інтерфейсу синхронізації даних.

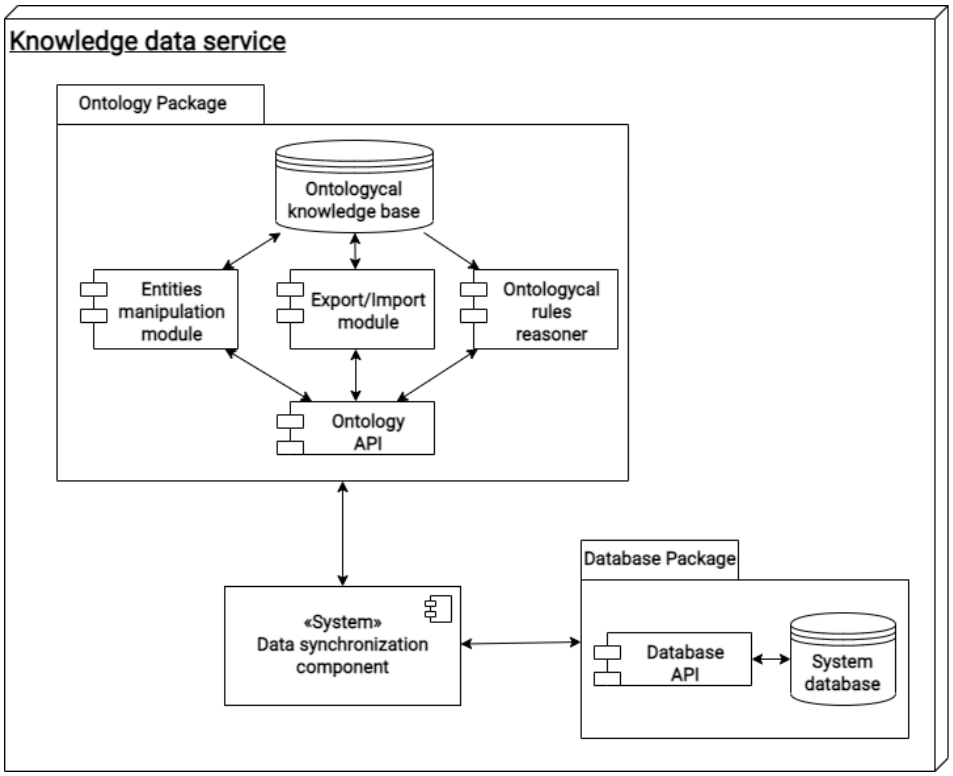

Рис. 3.1. Діаграма розгортання онтологічного сервісу даних та знань

Недоліком першого підходу є велика зв’язність між елементами та конкретними реалізаціями об’єктів онтології та баз даних. Це спричинить гіршу підтримуваність та ускладнить процес модифікації структури та функціональності процесу синхронізації. У такому разі найбільш ефективним є поєднання існуючих парадигм програмування та проєктування з онтологічним моделюванням.

Відповідно до принципів створення об’єктно-орієнтованого програмного забезпечення, кожна програма складається з об’єктів, що в свою чергу є екземплярами конкретного класу з типовою функціональністю та атрибутами. При цьому структура такого рішення є часто ієрархічною, а об’єкти в ній взаємодіють, використовуючи виклики функцій та повідомлення.

При опрацюванні онтології предметної області доцільним є використання

принципів об’єктно-орієнтованого програмування, оскільки при співставленні об’єкти структури моделі виражаються у конкретні елементи програмного рішення (рис. 3.2):

- концепт / поняття онтологічної моделі → клас;
- функціональні атрибути → поля та атрибути класу;
- відношення (властивості об’єктів) → атрибут класу, що містить посилання на інший клас;
- екземпляри або індивіди сутностей → об’єкти відповідного класу.

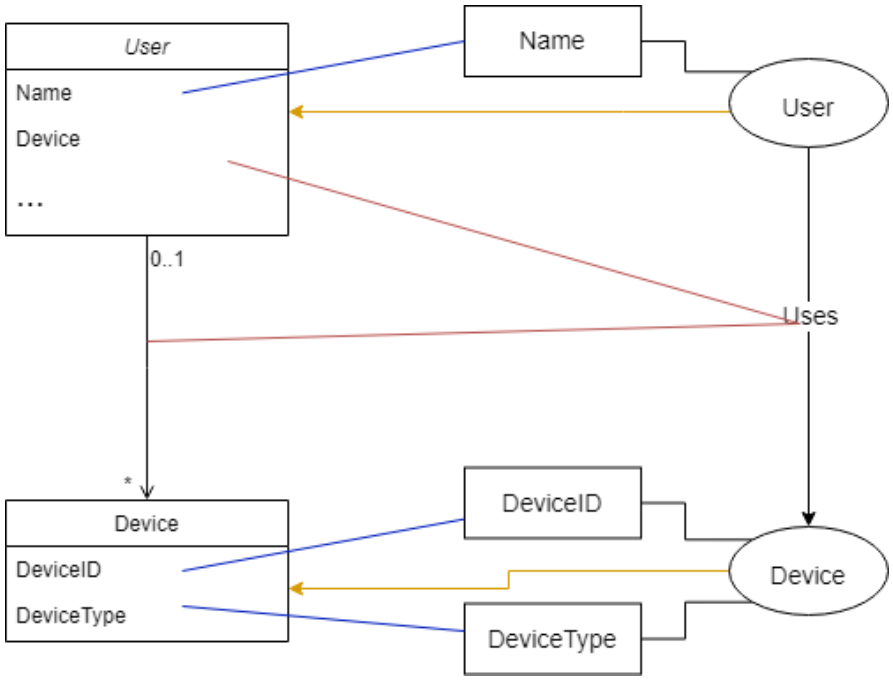

Рис. 3.2. Схема співставлення об’єктів ООП та елементів онтологічної моделі

Застосування принципів ООП дозволяє краще використовувати елементи онтологічної моделі [72]. Проте, для обміну інформацією з базою даних доцільно використовувати об’єктно-реляційне відображення (Object relational mapping –ORM). Дана технологія дозволяє не лише полегшити взаємодію з базою даних в процесі розробки, але й забезпечує представлення елементів бази даних у виглядів об’єктів. Така особливість забезпечує можливість зміни контексту та рушія бази даних, а також дає змогу проєктувати структуру та зв’язки бази даних напряму з коду за допомогою підходу code-first [110].

Окрім коректного представлення об’єктів та зв’язків предметної області,

іншою проблемою у розробці програмних систем є модифікація її функціональності. Ключовою особливістю адаптивних систем є спроєктована компонентно-орієнтована архітектура на основі використання компонент та модулів розширення [100, 101]. Типова структура багатомодульної програмної системи складається з основної (батьківської) компоненти, бібліотеки класів, властивостей та програмних інтерфейсів, що використовуються в процесі модифікації для приєднання додаткових функціональних компонент. Можливість динамічної зміни функціональних характеристик забезпечується завдяки реалізації визначених шаблонів та програмних інтерфейсів, які дають можливість обміну інформації та взаємодію між батьківською та завантаженою компонентами, при цьому знімаючи залежність від конкретної реалізації додаткових компонент системи.

Використання компонентно-орієнтованої архітектури є ефективним, оскільки усуває високу зв’язність між основним та додатковими модулями системи. Крім того, таке рішення надає можливість виконання динамічної модифікації як елементів графічного інтерфейсу так і функціональних модулів без виконання повторної реконфігурації системи. Однак, проєктування та використання plugin-архітектури вимагає попереднього визначення та реалізації додаткових шаблонів, структур даних, властивостей та програмних інтерфейсів, що забезпечують механізм підключення нових модулів або сервісів.

Використання лише компонентно орієнтованої архітектури не забезпечить достатнього рівня абстрактності, оскільки вимагатиме окремої реалізації процесу визначення конфігурації системи для кожного пристрою. В такому випадку доцільно використовувати поєднання компонентно-орієнтованої та клієнт-серверної архітектури. Для відображення запропонованої нами архітектури програмної системи використано діаграму розгортання, що представлена на рис. 3.3.

Принцип проєктування програмних засобів на основі трирівневої клієнт-серверної архітектури полягає у відокремленні вигляду системи від рівнів бізнес-логіки та даних [12]. Такий поділ дозволяє унеможливити втрату або

пошкодження даних, у разі, якщо буде сформовано хибний або некоректний запит. Крім того, таке рішення дозволяє замінити або вдосконалити кожен рівень незалежно один від одного.

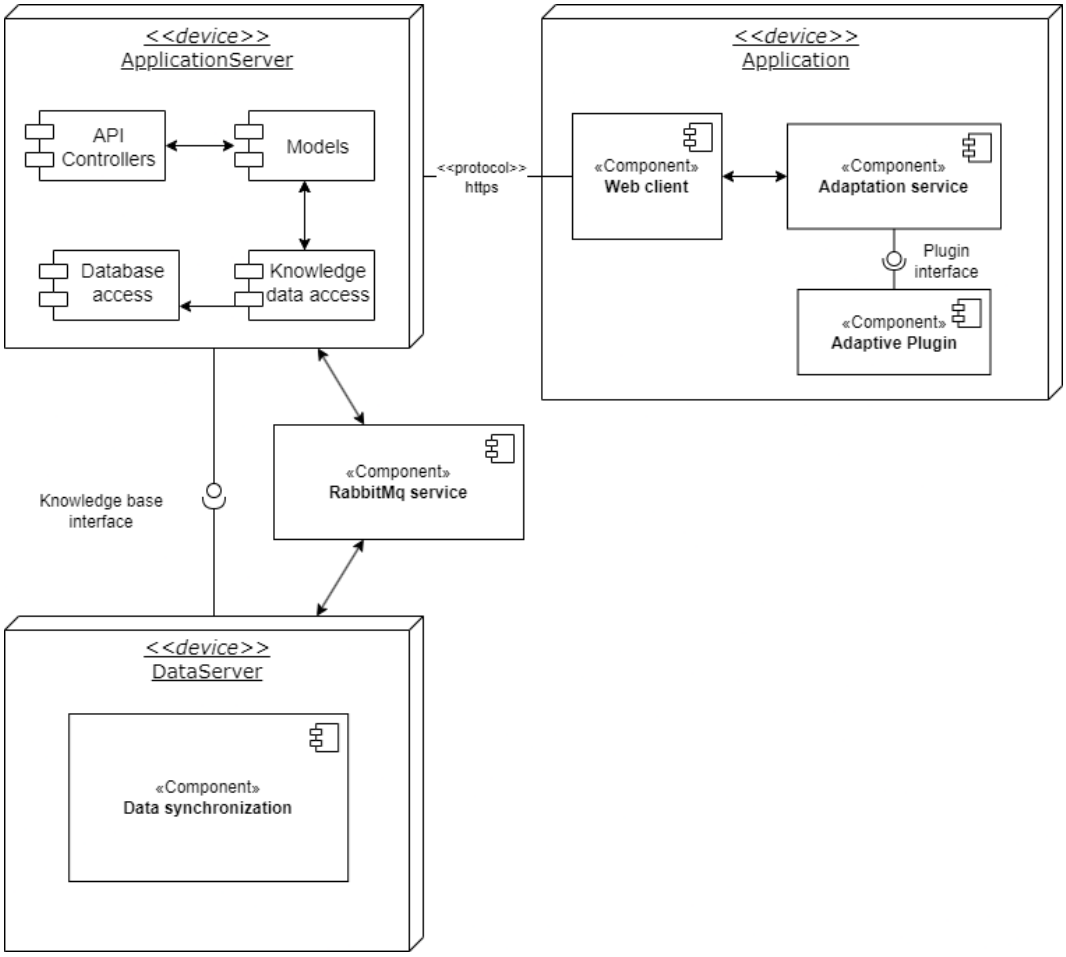

Рис. 3.3. Діаграма розгортання адаптивної програмної системи

Презентаційний рівень програмної системи представлений у вигляді мобільного, веб чи настільного застосунків, що спрямовані на безпосередню взаємодію з користувачем. Для забезпечення його розширюваності доцільно використати попередньо визначені властивості компонентно-орієнтованої архітектури [101]. Відповідно до вказаного підходу у системі виділяється два види компонент:

- основна – містить основну функціональність та є фактично ядром програмного засобу, до якої буде динамічно доєднуватися додаткова функціональність у процесі роботи програмного засобу.

- додаткова (plugin) – містить додаткову функціональність, що доповнює або розширює базову компоненту. Плагін можна додавати та видаляти в будь який момент, при цьому це не вплине на роботу інших плагінів.

Використання визначеної структури адаптивної системи дозволить забезпечити можливість повторного використання, розширюваності та взаємозамінності, оскільки кожен модуль буде реалізовано незалежно один від одного, а комунікація між модулями відбуватиметься безпосередньо через основну (батьківську) компоненту.

Рівень бізнес-логіки реалізується за допомогою прикладного програмного інтерфейсу та призначений для обробки даних користувача та запитів на адаптацію. Інформація про вимоги користувача та тип адаптивної системи, що надходить від застосунку опрацьовується та надсилається на рівень даних, де відбувається синхронізація бази даних та онтологічної бази знань, а також здійснюється конфігурація застосунку. На основі інформації отриманої з рівня даних формуються персоналізовані налаштування графічного інтерфейсу та список необхідних функціональних модулів.

Відповідно до вказаних принципів спроєктовано три рівні програмної системи:

- кросплатформовий застосунок, що відповідає за представлення інформації кінцевому користувачу та забезпечує можливість динамічної зміни графічного інтерфейсу та функціональності програми;
- вебсервер (прикладний програмний інтерфейс), що опрацьовує HTTPS-запити отримані з презентаційного рівня та виконує синхронізацію інформації між базою даних та онтологічною базою знань, використовуючи прикладний програмний інтерфейс pyodbc;
- вебсервіс керування базою знань, що містить в собі базу даних, в якій зберігається інформація про користувачів та їх завдання, а також онтологічну модель, що використовується для формування персоналізованих налаштувань.

Зв’язок між прикладним програмним інтерфейсом (API) та вебсервісом керування базою знань відбувається у два способи:

- використовуючи інтерфейс для зв’язку з базою даних – застосовується у разі виконання операцій читання та запису у базу даних або базу знань;
- використовуючи брокер повідомлень та протокол AMPQ – застосовується для формування запиту на адаптацію програмного забезпечення.

Поєднання таких двох підходів до зв’язування API та вебсервісу бази знань дозволяє покращити продуктивність та пришвидшити час опрацювання запитів різного типу. Зокрема, якщо користувачу не потрібно оновлювати конфігурацію програмного забезпечення, то зміни будуть асинхронно внесені в базу даних та онтологічну модель. Проте, якщо необхідно здійснити адаптацію програмного забезпечення, активується зв’язок з вебсервісом через брокер повідомлень. Це дасть змогу розподілити процес адаптації за типом програмного застосунку.

Окрім цього, оскільки процес опрацювання онтологічних правил є синхронним та однопотоковим, то такий поділ забезпечить можливість одночасного формування модифікованих налаштувань для різних типів програмних застосунків. Наприклад, якщо вебсервіс здійснює конфігурацію мобільного застосунку, то сервіси для конфігурування настільного та вебзастосунків будуть працювати в попередньому режимі без блокування основного потоку.

3.2. Метод використання брокерів повідомлень у процесі адаптації програмного забезпечення та механізми їх комунікації

Конкурентоздатність сучасного програмного забезпечення на широкому ринку інформаційних продуктів у великій мірі залежить наскільки гнучкою та адаптивною може бути програмна система. Ця вимога залежить від архітектури програмного забезпечення, її модульності та можливості динамічного

налаштування програмних компонентів. Виконання поставлених задач в залежності від отриманої інформації про систему та потреби користувача забезпечують адаптивні програмні системи. Проте, незважаючи на високу ефективність таких рішень, однією із проблем є забезпечення обміну повідомленнями у системі та обмін інформацією про адаптацію окремих компонент.

В реальних системах кількість користувачів перевищує сотні тисяч, що викликає необхідність правильного опрацювання запитів. В таких випадках доцільним є використання технології message-broker, які дозволяють формувати чергу повідомлень на основі пріоритетності та типу інформації [78]. Така черга повідомлень дозволяє формувати гарантоване опрацювання повідомлень та розподіляти потік даних між різними сервісами, без необхідності формування додаткового сервера-посередника [98, 116].

Механізм брокера повідомлень за своєю семантикою дозволяє підлаштовувати дані під потреби кожного сервісу або користувача. Окрім того, черги повідомлень можуть бути налаштовані таким чином, що гарантуватимуть доставку повідомлення, розподіл потоків даних, підписку на потрібні типи повідомлень [66, 86, 67].

Стандарти брокерів повідомлень мають певні відмінності, проте кожен реалізовує таку структуру компонентів обміну повідомлень:

- *Exchange* – частина брокера (наприклад сервер чи відповідний сервіс), яка отримує повідомлення та направляє їх у черги;
- *Queue* – тимчасові сховища, до яких надходять повідомлення та звідки споживачі їх отримують;
- *Bindings* – правила розподілу повідомлень з компоненти Exchange у компонент Queue

Таким чином при налаштуванні обміну повідомленнями формується певна сутність “*Exchange*”, до якої прив’язується черга за допомогою певних визначених правил розподілу.

Використання зазначеної технології для адаптивних програмних систем

забезпечить розділення процесу визначення конфігурації а також адаптації графічних компонент та функціональних модулів. Такий розподіл гарантуватиме коректне опрацювання даних та допоможе уникнути виняткових ситуацій, якщо адаптація компонентів є невдалою.

Відповідно до реалізації брокерів повідомлень, кожен об’єкт сутності “*Exchange*” також відрізняється за типом. Загалом “*Exchange*” поділяється на такі основні групи [49, 58]:

- *Direct* – доставляє повідомлення в черги на основі ключа маршрутизації повідомлень (рис. 3.4). Черга прив’язується до Exchange за допомогою ключа маршрутизації K.

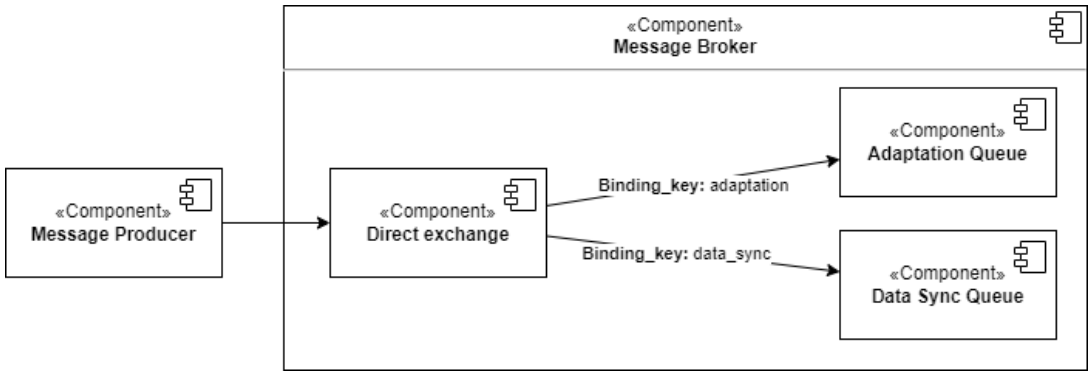

Рис. 3.4. Приклад структури типу обміну «Direct Exchange»

- *Topic* – доставляє повідомлення до однієї або кількох черг на основі відповідності між ключем маршрутизації повідомлень і шаблоном, який використовувався для прив’язки черги до обміну (рис. 3.5). Такий тип обміну повідомленнями часто використовується для реалізації різних варіацій шаблону публікації/підписки:

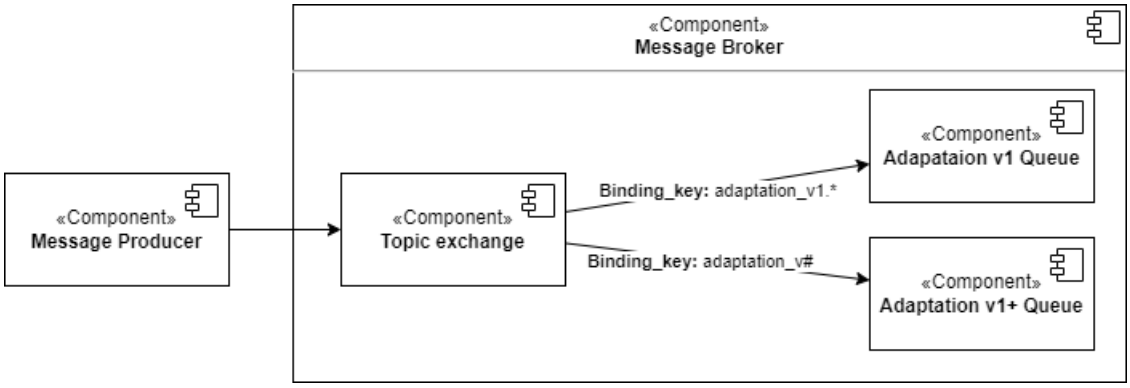

Рис. 3.5. Приклад структури типу обміну «Topic Exchange»

- *Fan-out* – направляє повідомлення до всіх прив'язаних до нього черг, а ключ маршрутизації ігнорується (рис. 3.6). Якщо N черг прив'язано до fan-out, коли нове повідомлення публікується на цьому обміні, копія повідомлення доставляється до всіх N черг:

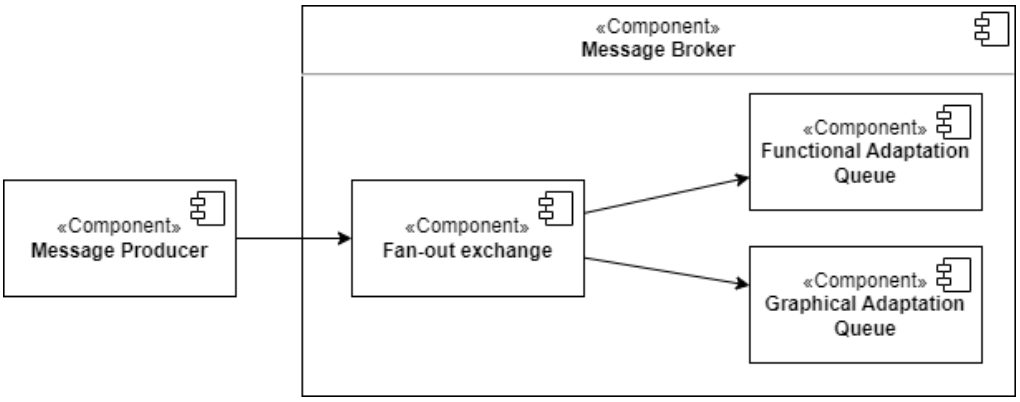

Рис. 3.6. Приклад структури типу обміну «Fan-out Exchange»

- *Header* – тип обміну, що використовує значення заголовків повідомлень та/або додаткові параметри під час маршрутизації повідомлення (рис. 3.7). Окрім додаткових параметрів для типу *header exchange* необхідно також визначити параметр «*x-match*», що приймає два значення:
    - «*all*» – усі користувацькі параметри та умови повинні бути співставленими;
    - «*any*» – хоча б один параметр повинен бути співставлений під час опрацювання повідомлення.

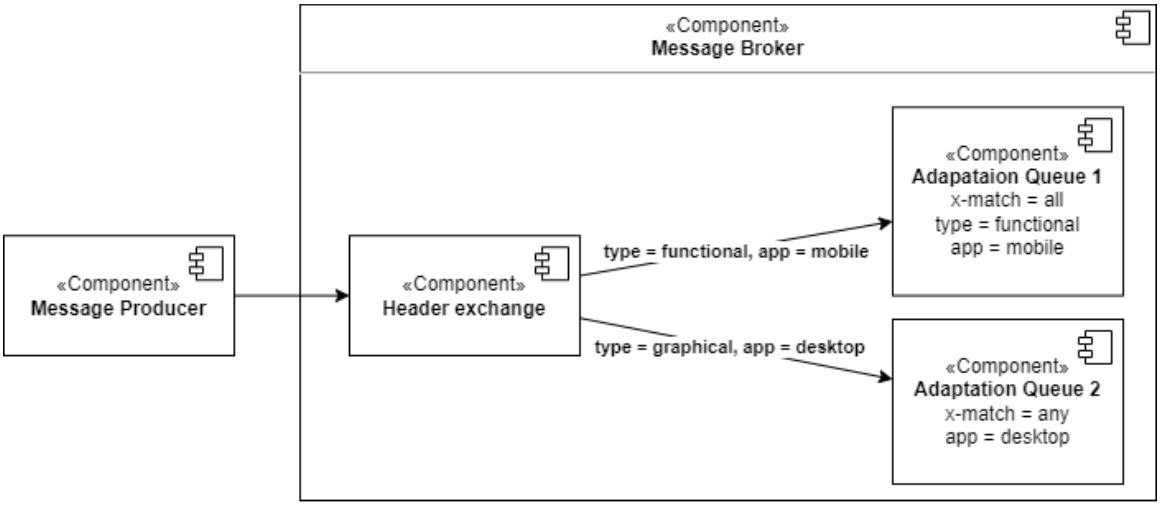

Рис. 3.7. Приклад структури типу обміну «Header Exchange»

- *Default* – стандартизований варіант *direct exchange*. Дозволяє здійснювати обмін шляхом доставлення повідомлення в чергу з іменем, що дорівнює ключу маршрутизації повідомлення. Кожна черга автоматично прив'язується до *default exchange* за допомогою ключа маршрутизації, який збігається з назвою черги.
- *Dead letter* – удосконалений варіант *topic exchange*. Дозволяє зберегти та надіслати повідомлення або дані до окремої черги, якщо під час їх опрацювання виникла помилка (рис. 3.8).

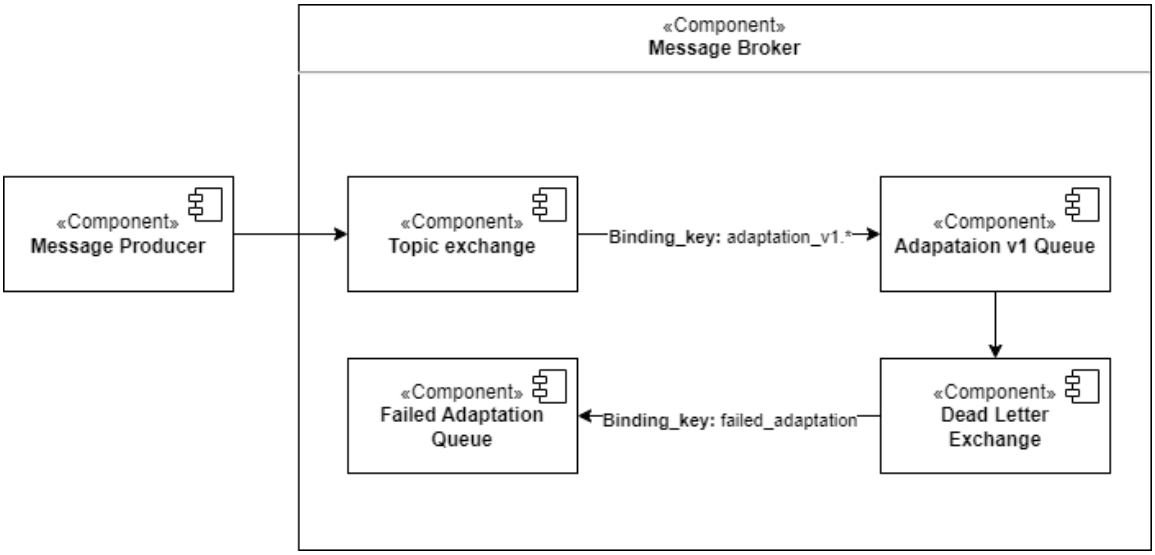

Рис. 3.8. Приклад структури типу обміну «Dead Letter Exchange»

У випадку динамічного формування конфігурації для адаптивних програмних систем можливе поєднання всіх основних типів обмінів повідомлень:

- *direct exchange* – доцільно використовувати для розподілу запитів на синхронізацію та адаптацію програмного застосунку. Іншим варіантом використання цього обміну є розподіл запиту на модифікацію конфігурації у залежності від типу застосунку;
- *topic exchange* – доцільно використовувати для формування запитів в залежності від версії адаптованого програмного забезпечення.

Оскільки під час застосування змін щодо адаптації можливе виникнення виняткових ситуацій, тому для цього процесу найбільш доцільно буде

використовувати тип обміну *dead letter exchange* (рис. 3.9). Такий обмін дає змогу не лише робити розподіл між типами програмних застосунків, але і забезпечить можливість кращого реагування на помилки в адаптації. Крім того, використання *dead letter echange* дозволяє зберегти запит на повторну конфігурацію системи у разі виникнення проблем з мережею або коли вебсервіс, що виконує адаптацію наразі не доступний.

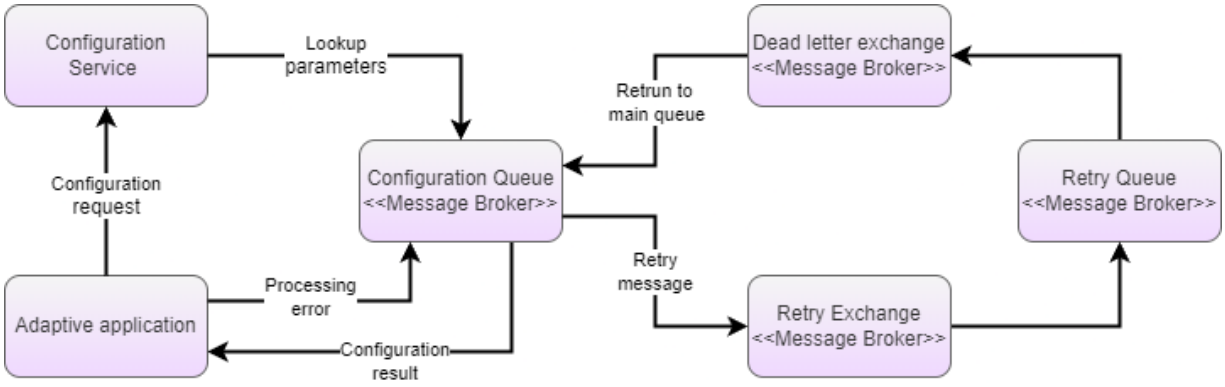

Рис. 3.9. Схема реалізації типу обміну «*dead letter exchange*» для адаптивних програмних систем

Відповідно до спроєктованої моделі обміну *dead letter exchange*, процес формування запиту на адаптацію програмного забезпечення виконується у послідовності [78]:

1. Програмне забезпечення надсилає інформацію про вимоги користувача на вебсервіс. При цьому здійснюється валідація отриманих даних та формується загальна структура повідомлення для його подальшого опрацювання у черзі брокера повідомлень;
2. Після розміщення повідомлення у черзі, сервіс конфігурації зчитує його та здійснює пошук та адаптацію поточної конфігурації програмного забезпечення для користувача, використовуючи наявну у повідомленні інформацію;
3. В залежності від результату адаптації сервіс конфігурації може виконати одну з двох дій:
    a. *у разі успішного визначення конфігурації* – користувачу

надсилаються адаптовані параметри та конфігурація системи. При цьому процес адаптації може бути продовжено на пристрої користувача;

b. *у разі виникнення помилки* – повідомлення поміщається назад у чергу. Процес адаптації буде відновлено з кроку визначення оптимальних параметрів та конфігурації програмного забезпечення.

Отже, така реалізація методу використання брокерів повідомлень дозволяє не лише розподілити навантаження на програмні та апаратні ресурси прикладного програмного інтерфейсу та сервісу конфігурації, але й забезпечує механізм перезапуску адаптації у випадку виникнення помилки.

3.3. Використання принципів рефлексії для реєстрації компонент в адаптивній програмній системі

Спроєктовані архітектура та зв’язки між компонентами адаптивної програмної системи надають можливість проведення динамічної адаптації функціональних та графічних елементів системи. Іншим завданням адаптації є коректна реєстрація метаданих про завантажені підмодулі. Саме метадані використовуються для визначення типу та точки входу для запуску модуля. Цю реєстрацію компонент можна здійснювати за допомогою двох підходів:

- Попереднє встановлення компоненти – завантажена компонента встановлюється на операційну систему, файл з метаданими програми зберігається поруч. Це забезпечує можливість запуску завантаженої компоненти без необхідності запуску основного програмного забезпечення;
- Використання рефлексії – завантажена компонента зберігається в директорії основного програмного забезпечення. Доступ до модулів та метаданих здійснюється за допомогою рефлексії.

Одним з недоліків використання попереднього встановлення допоміжної компоненти напряму на операційну систему це доступ та керування нею з

основного застосунку. У більшості операційних систем застосунки не мають прав доступу для встановлення та зміни інших програм. Це унеможливлює контроль за діяльністю допоміжної компоненти, оскільки її видалення чи модифікація не буде можливою всередині адаптивного програмного забезпечення.

Використання рефлексії дає змогу вирішити цю проблему, оскільки компоненту не потрібно встановлювати, а доступ до даних чи ресурсів надається безпосередньо з батьківської компоненти. Проте, даний підхід передбачає використання лише попередньо створеної бібліотеки шаблонів та класів, які визначатимуть програмні інтерфейси, які будуть реалізовуватися допоміжними модулями для забезпечення взаємодії [34].

За своїм змістом рефлексія забезпечує механізм для динамічного доступу до властивостей об’єктів та класів під час роботи програмного забезпечення. Зазвичай даний механізм використовується для динамічного створення та використання інформації про тип динамічного об’єкта та значення його атрибутів. Наприклад, за допомогою рефлексії можна отримати список усіх типів, що містяться в отриманій збірці, бібліотеці чи виконуваному файлі *.dll або *.exe, включаючи методи, поля, властивості та події. Крім того, рефлексія забезпечує можливість завантаження метаданих у прийнятному об’єктно-орієнтованому форматі [111].

Іншою перевагою використання механізмів рефлексії є повторне використання архітектури програмного забезпечення та його складових [84, 121]. Рефлексія архітектури виконується системою щодо її власної архітектури та наразі обмежена часом виконання та динамічним розвитком програмних систем. Проте, незважаючи на обмежену область застосування, такий механізм рефлексії дозволяє ефективно виконувати завдання проєктування архітектури на етапі розробки програмного забезпечення. Крім того, це забезпечує можливість повторно використовуючи архітектуру програмного забезпечення та її складові [40].

У випадку адаптивного програмного забезпечення механізм рефлексії

доцільно використовувати під час завантаження та застосування оновленої конфігурації. У процесі адаптації сформована конфігурація програмного забезпечення передається на менеджер конфігурації. Через менеджер конфігурації за допомогою рефлексії з метаданих видобувається інформація в залежності від типу компоненти:

- Графічна компонента – реєструється набір стилів та графічних компонент у менеджері стилів. При цьому додаткові правила конфігурації дозволяють уникати ситуації коли декілька компонент намагаються змінити один стиль;
- Функціональна компонента – для даного типу в системі реєструються метадані для завантаженої бібліотеки, ім’я точки входу (методу, структури або класу) для запуску компоненти, шлях до завантаженого файлу. Збереження цих параметрів дозволить доступитися до основних елементів адаптивної компоненти під час роботи програмного забезпечення.

3.4. Алгоритм динамічної адаптації інтерфейсу користувача та функціональності програмної системи з використанням онтологічної моделі

Поєднання трирівневої клієнт серверної архітектури і механізму брокера повідомлень дозволяють забезпечувати можливість динамічного визначення налаштувань та характеристик ПЗ та при цьому збільшити продуктивність у разі збільшення навантаження на сервер. Проте, такий механізм адаптації буде викликатися кожного разу при створенні запиту на модифікацію конфігурації. Вирішенням цієї проблеми є використання процесу кешування у разі успішного виконання запиту на адаптацію. В такому разі, якщо інформація в онтологічній моделі не змінилася, то кожен наступний запит на конфігурацію буде використовувати попередньо записані дані в кеші, без необхідності синхронізації з вебсервісом.

Процес визначення налаштувань та адаптації функціональних та

графічних модулів з використанням брокера повідомлень та механізму кешування наведено на рис. 3.10.

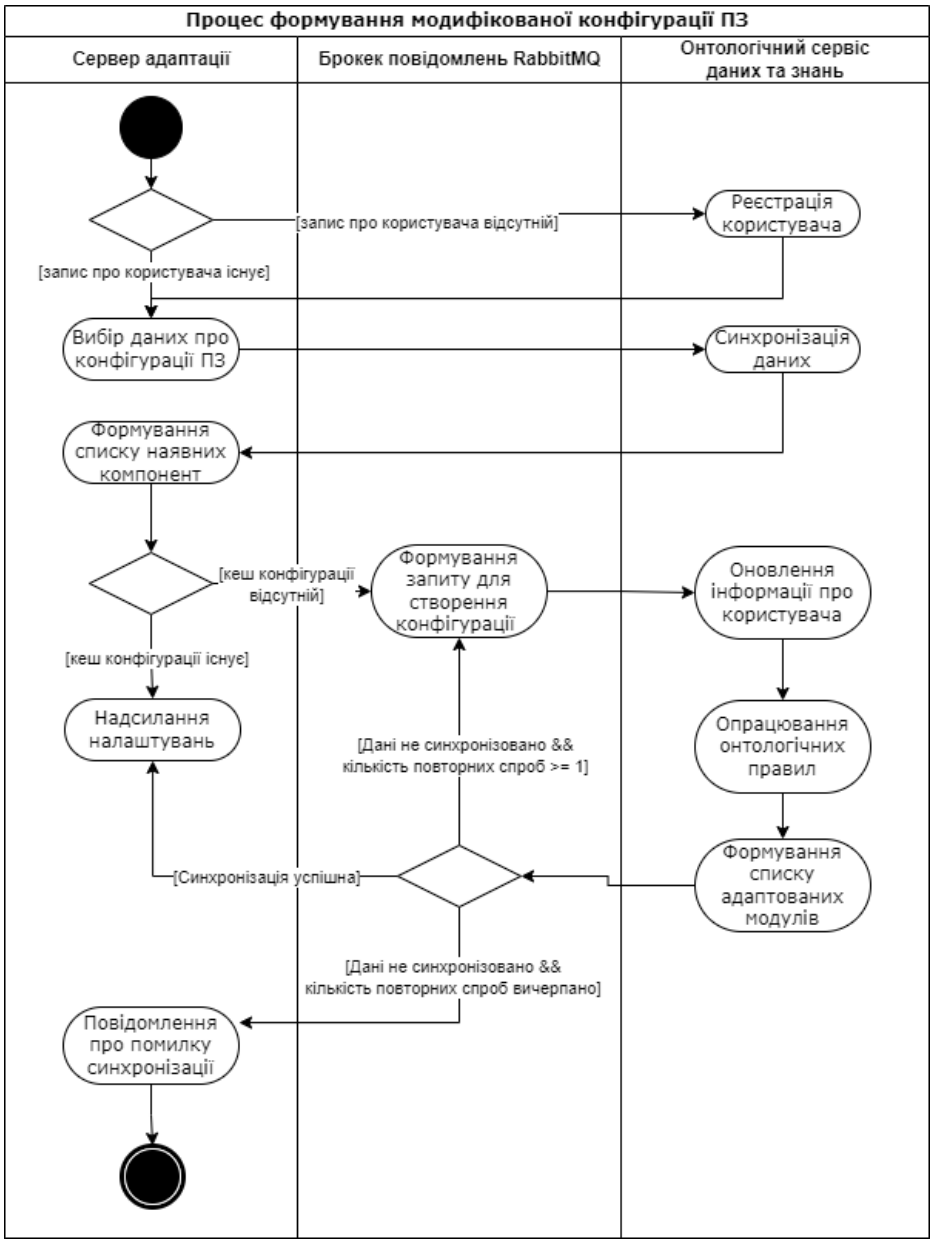

Рис. 3.10. Схема процесу динамічного визначення налаштувань програмного забезпечення

Адаптація програмного забезпечення розпочинається з авторизації користувача та пошуку його даних в онтологічній базі знань. Авторизація в даному процесі дозволяє не лише визначити роль та рівень доступу користувача в системі, але і забезпечує можливість збереження попередньо створених конфігурацій. Така реалізація забезпечує можливість зміни пристрою без необхідності повторного запиту до адаптації, у разі, якщо вимоги

користувача до системи не змінилися. Якщо запис користувача в базі знань не знайдено, відбувається запис інформації в онтологічну модель та синхронізація з базою даних.

Після успішної авторизації в системі вебсервер надсилає на застосунок користувача набір налаштувань та описи вимог для подальшого налаштування модулів програмного забезпечення. Отримані вимоги компонуються в форму для кращого відображення зв’язку між конкретною вимогою та модулем-рішенням.

Вибрані користувачем вимоги формують запит на адаптацію. У запиті надсилається інформація про зареєстровані в системі залежності між вимогами та наявними компонентами. У процесі опрацювання запиту сервер здійснює перевірку на наявність попередніх конфігурацій у кеші. Якщо запис у кеші існує, тоді формується список налаштувань та змін, які необхідно застосувати у застосунку користувача.

У випадку, коли попередніх конфігурацій не знайдено, створюється запит на синхронізацію з базою знань у брокері повідомлень. Онтологічний сервіс бази знань та даних асинхронно зчитує інформацію та, при потребі, оновлює інформацію про користувача. Після цього запускається семантичний механізм опрацювання онтологічних правил та визначаються модифіковані налаштування програмного забезпечення для користувача.

У процесі опрацювання онтологічних правил можливі кілька результатів:

- **синхронізація пройшла успішно** – семантичний механізм опрацювання онтологічних правил сформував нову конфігурацію та налаштування програмного забезпечення на основі вимог користувача. Нові налаштування системи надіслано на застосунок користувача;
- **дані не синхронізовано кількість спроб на оновлення інформації не вичерпано** – початковий запит на оновлення конфігурації поміщається назад у брокер повідомлень. Після очікування у спеціалізованій черзі запит повторно передається на обробку

онтологічному сервісу даних та знань;

- **дані не синхронізовано кількість спроб на оновлення інформації вичерпано** – запит на оновлення інформації видаляється з брокеру повідомлень, вебсервер надсилає повідомлення про помилку синхронізації та здійснює запис в журнал помилок для можливості подальшого їх аналізу.

Після успішної синхронізації сформовані персоналізовані налаштування опрацьовуються у застосунку користувача та запускається алгоритм адаптації програмного забезпечення, процес якого наведено на рис. 3.11.

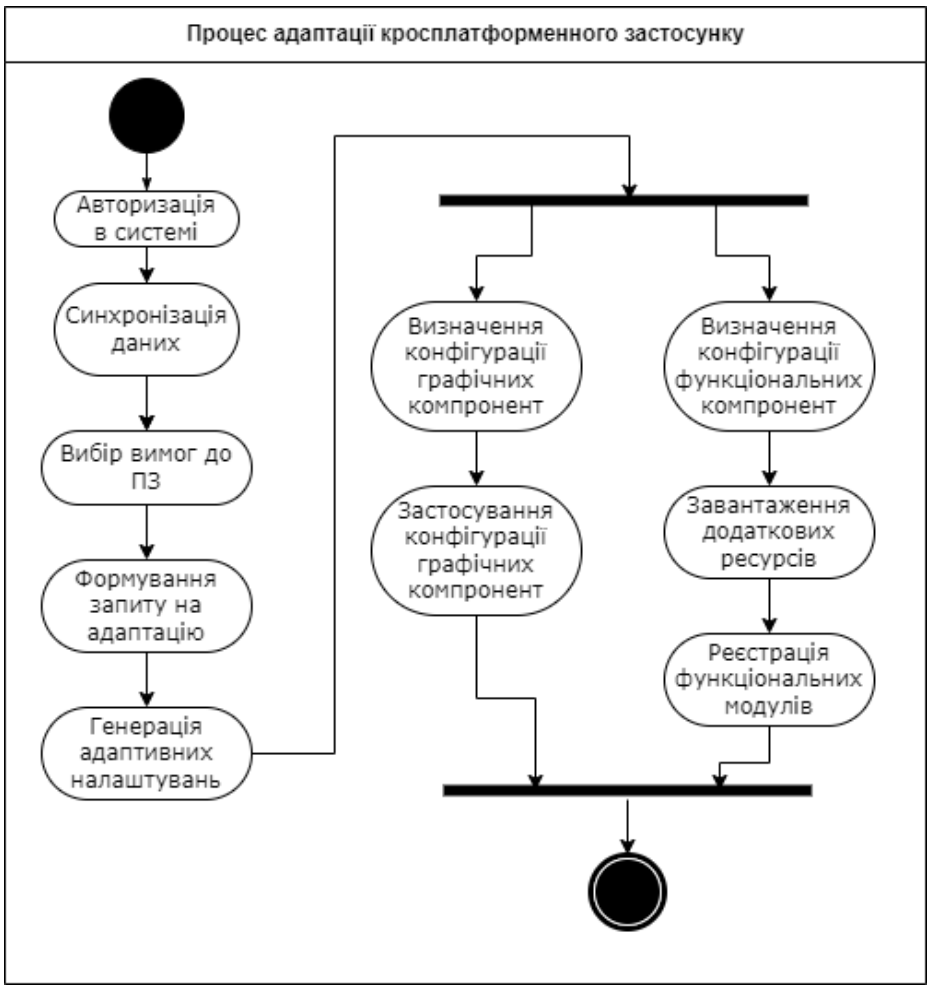

Рис. 3.11. Схема процесу адаптації програмного забезпечення на основі визначеної конфігурації

Таким чином, спочатку відбувається десеріалізація повідомлення з метою визначення інформації про налаштування графічних та функціональних

модулів. Після цього відбувається розгалуження процесу адаптації, що дозволяє паралельно застосувати налаштування для отриманих модулів. Детальний процес визначення та застосування модулів наведено в алгоритмі:

Алгоритм 1: Адаптація програмного забезпечення на основі визначеної конфігурації

Вхідні дані: набір налаштувань json_settings у форматі JSON

```
1   settings ← Deserialize-Settings(json_settings)
2   configResults ← empty list of configuration results
3   for setting in settings do
4       module_type ← Get-Module-Type(setting)
5       module_data ← Get-Module-Data(module_type, setting)
6       if module_data.HasAdditionalResources = true then
7           module_data ← Download-Resource(module_data)
8       end if
9       result ← Apply-Config(module_type, module_data)
10      configResults.Add(result)
11  end
12  Save-Configuration(configResults)
```

Відповідно до алгоритму адаптації, застосунок користувача після десеріалізації розпочинає процес застосування персоналізованих налаштувань. Застосування налаштувань для всіх модулів відбувається паралельно для швидшого завершення процесу модифікації конфігурації системи. Модифікація відбувається у такій послідовності:

1. Визначення типу модуля: графічний, функціональний.
2. На основі інформації про тип модуля здійснюється визначення характеристик та налаштувань які необхідно змінити:
    а) для графічного модуля – визначаються зміни які необхідно застосувати для мультимедійних елементів чи інтерфейсу користувача;

b) для функціонального модуля – визначається тип застосування та метод, який забезпечує доступ до іншої функціональності отриманого модуля.

3. Якщо отриманий модуль потребує додаткових ресурсів – здійснюється їх завантаження та оновлюється інформація про модуль.
4. Відбувається модифікація застосунку користувача відповідно до даних та налаштувань визначеного модуля.
5. Після завершення модифікації застосунку результати та налаштування зберігаються для забезпечення можливості використання створеної конфігурації без необхідності повторного запиту до сервера.

Використання такого алгоритму адаптації програмного забезпечення дозволяє забезпечити можливість динамічної модифікації налаштувань та наповнення застосунку користувача без необхідності статичного створення всіх можливих конфігурацій. Крім того, використання трирівневої архітектури з можливістю кешування результатів адаптації у поєднанні з брокером повідомлень дозволяє знизити навантаження на ресурси сервера. Також це дозволятиме зменшити час очікування на оновлення та синхронізацію даних у разі, якщо попередня конфігурація уже існує у кеші, а користувач не змінював попередньо збережену інформацію.

3.5. Висновки до розділу

У розділі охарактеризовано основні архітектурні рівні та алгоритмічну модель процесу адаптації програмного забезпечення. Встановлено, що зміни до структури програмної системи сприятимуть збільшенню кількості зв’язків між компонентами програмного забезпечення, що сприятиме зростанню тривалості опрацювання запитів адаптації. Вирішенням даної проблеми є використання брокерів повідомлень, що дозволяють розподілити навантаження на окремі сервіси програмного забезпечення.

Спроєктовано архітектуру адаптивної програмної системи, що дозволяє виконувати динамічну конфігурацію програмного забезпечення. У поєднанні з

онтологічною моделлю та брокером повідомлень таке рішення дозволить виконувати адаптацію, враховуючи вимоги користувача, тип необхідної адаптації та програмну платформу, на якій відбувається адаптація.

Спроєктовано модель обміну повідомлень для адаптивних програмних систем, що дозволяє використовувати технології брокерів повідомлень у адаптивних програмних системах реального часу для коректного опрацювання даних та уникнути виняткових ситуацій у випадку, якщо адаптація компонентів є невдалою. Принцип *dead letter exchange* забезпечує можливість перенаправляти повідомлення назад у чергу без втрати вже опрацьованих даних, що в подальшому зменшує навантаження на сервіс адаптації.

Визначено основні принципи використання рефлексії для адаптації компонентів програмного забезпечення. Встановлено, що використання механізму рефлексії під час модифікації програмного забезпечення дозволяє уникнути попереднього встановлення програмного модуля та повторної конфігурації програмної системи.

Запропоновано алгоритм адаптації програмного забезпечення, що дозволяє забезпечити можливість динамічної модифікації налаштувань та наповнення застосунку користувача без необхідності статичного створення всіх можливих конфігурацій. Крім того, це дозволятиме зменшити час очікування на оновлення та синхронізацію даних у разі, якщо попередня конфігурація уже існує у кеші, а користувач не змінював попередньо збережену інформацію.

РОЗДІЛ 4. РЕАЛІЗАЦІЯ АДАПТИВНОЇ ПРОГРАМНОЇ СИСТЕМИ НА ОСНОВІ ОНТОЛОГІЧНОГО ПІДХОДУ ТА ДОСЛІДЖЕННЯ ЇЇ ЕФЕКТИВНОСТІ

4.1. Опис використаних технологій проєктування та розробки прототипу адаптивної програмної системи

На основі спроєктованого методу адаптації програмних систем розроблено прототип програмної системи для допомоги людям з когнітивними порушеннями, що спроєктована за принципами трирівневої та компонентно-орієнтованої архітектури та складається з crossplatform-застосунку, вебсервера, сервісу онтологічної бази знань та брокера повідомлень RabbitMQ.

Для створення адаптивної програмної системи були використані такі засоби розробки:

- IDE Visual Studio, мова програмування C#, технологія .NET MAUI для створення crossplatform-додатку та технологія ASP.NET Core Web API – для реалізації прикладного програмного інтерфейсу (API);
- Visual Studio Code, мова програмування Python і вебфреймворк Flask для створення сервісу онтологічної бази знань та синхронізації даних між онтологією, базою даних та прикладним програмним інтерфейсом;

Crossplatform застосунок реалізовано для платформ Android та Windows з використанням бібліотеки MAUI .NET та мови програмування C#. Створено бібліотеку класів, що дозволяє визначати абстракції та механізми модифікації елементів інтерфейсу користувача та функціональних компонент.

Метод визначення, встановлення та адаптації функціональності програмної системи та інтерфейсу користувача реалізовано за допомогою рефлексії. Дане рішення дозволяє динамічно в процесі роботи програмного забезпечення вирішувати наступні завдання:

- аналізувати отриману інформацію про адаптацію конкретного компоненту, у процесі чого визначається тип модуля (функціональний або графічний) та область його застосування;

- реєструвати отримані компоненти та зберігати сформовану конфігурацію, що забезпечує можливість користування оновленим функціоналом без необхідності повторного запиту до серверу в разі завершення роботи програми.

Рівень бізнес-логіки, який відповідає за обробку інформації, що надходить на онтологічний сервіс, а також забезпечує проєктування відповідей, реалізовано за допомогою мови програмування Python і вебфреймворку Flask відповідно до вимог і принципів REST (Representational State) архітектурного стилю.

Для надання доступу створеного вебресурсу до онтологічної моделі предметної області використано модуль Owlready2, який забезпечує можливість створення онтологічно орієнтованої програмної системи. Інформація, отримана від користувача, записується у внутрішнє сховище за допомогою модуля SQLAlchemy, який дозволяє взаємодіяти з реляційними базами даних. Синхронізація даних, отриманих з мобільного додатку, та інформації з онтологічної бази знань здійснюється за допомогою системи опрацювання онтологічних правил Pellet.

4.1.1. Структура та функціональність бібліотеки для адаптації програмного забезпечення

Основою для успішної адаптації програмного забезпечення є бібліотека, що містить необхідні рівні абстракції та механізми модифікації існуючої функціональності та графічного інтерфейсу. З метою подальшої можливості паралельного опрацювання або модифікації лише функціональних чи графічних елементів визначено окремі інтерфейси та абстракції відповідних компонент:

- *IPlugin* – базовий абстрактний клас, що визначає основні елементи та методи адаптації для функціональних компонент та модулів;
- *IUserInterfaceElement* – базовий абстрактний клас, що визначає основні елементи та методи адаптації для графічних елементів та

ресурсів програмного забезпечення.

Окрім визначення абстрактних класів для кращої організації процесу адаптування програмного забезпечення створено відповідні перелічувані типи, що дозволяють розрізнити підтипи адаптованих компонент:

- *PluginType* – визначає підтип функціональної компоненти: dll-бібліотека чи програмний застосунок. Такий поділ дозволяє виділити компоненти, які відразу готові до використання після завантаження та яким не потрібне додаткове встановлення;
- *UIElementType* – визначає підтип графічної компоненти: сторінка, стиль, набір значень графічних ресурсів, шрифт, зображення. Поділ графічних компонент забезпечує можливість компонування ресурсів по групах з подальшим використанням або для всього програмного забезпечення або ж лише для його частини.

Визначені абстракції та перелічувані типи дозволяють однозначно ідентифікувати адаптивні елементи програмного забезпечення та здійснити їх модифікацію відповідно до встановлених правил. Відображення залежностей між основним програмним забезпеченням, бібліотекою адаптації та адаптованими компонентами наведено на рис. 4.1.

Відповідно до поданої схеми бібліотека адаптації не містить залежностей та зв’язків з основним та додатковими компонентами програмної системи. Це забезпечує можливість новим адаптивним модулям бути повністю незалежними від основної компоненти. Крім того, така модульність також не передбачає жорсткої залежності та попередньої реєстрації всіх додаткових модулів у адаптивній програмній системі. Кожна реалізація нової компоненти використовує структури, які визначені у бібліотеці адаптації, що в подальшому зробить їх доступними у процесі модифікації програмного забезпечення.

Використання слабкозв’язної архітектури забезпечує можливість зміни функціональності без необхідності повторної конфігурації системи. Проте, з’являється проблема отримання та аналізу інформації про додатковий модуль із основного програмного забезпечення. Оскільки немає прямої реєстрації та

зв’язування компонент, в такому разі доцільно використовувати рефлексію задля визначення типу компонента та його точки входу / запуску.

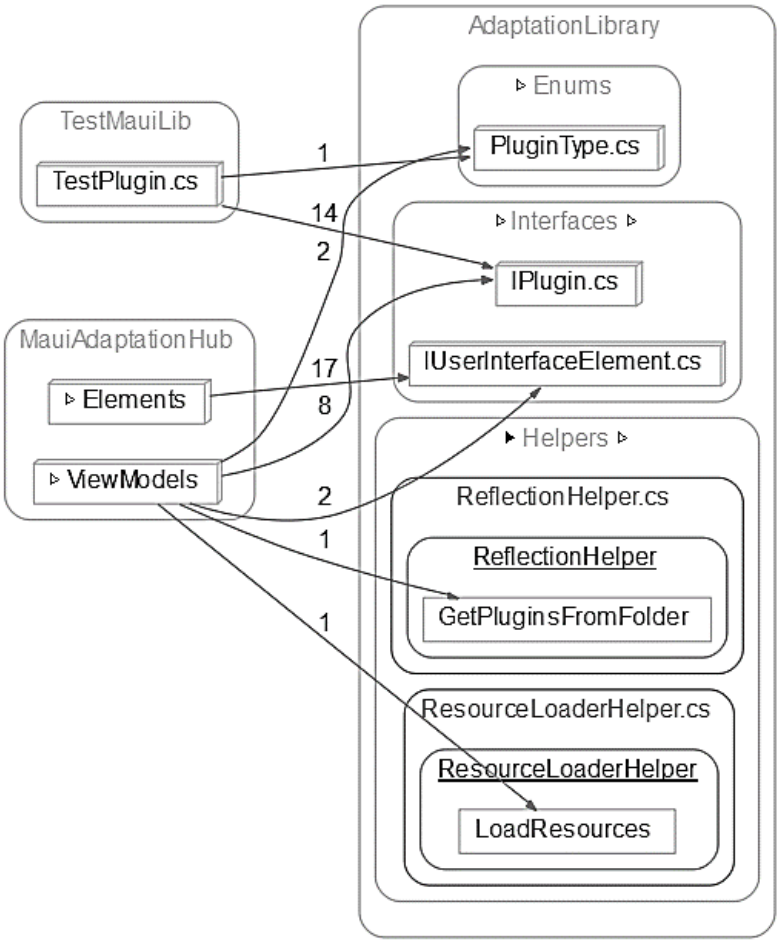

Рис. 4.1. Візуалізація залежностей між компонентами адаптивної програмної системи

Платформа .NET пропонує кілька методів для використання елементів рефлексії з метою дослідження завантаженого модуля в процесі виконання програмного забезпечення. У процесі адаптації програмного забезпечення доцільно використовувати:

- Assembly.LoadFile – забезпечує можливість динамічного завантаження .dll файлів в процесі роботи програмного забезпечення. Об’єкт отриманий в результаті виконання цієї функції використовується в подальшому дослідженні та опрацюванні завантаженої бібліотеки;

- ExportedTypes – повертає всі відкриті типи даних, межа видимості яких дозволена поза межами сформованої збірки;
- Activator.CreateInstance – забезпечує можливість динамічного створення об’єкта визначеного типу, класу або структури. У адаптації програмного забезпечення цей метод доцільно використовувати для створення об’єктів адаптивних функціональних компонент.

Ці три методи дозволяють під час роботи програмного забезпечення динамічно змінювати та додавати функціональність без необхідності проводити реконфігурацію. У поєднанні з визначеними абстрактними структурами даних це дозволяє зменшити зв’язність між батьківською та допоміжними компонентами. Реалізація алгоритму визначення та реєстрації допоміжних функціональних компонент наведено у фрагменті коду:

```
public static IEnumerable<IPlugin> GetPluginsFromFile(string
fileName)
{
    List<IPlugin> plugins = new List<IPlugin>();
    var DLL = Assembly.LoadFile(fileName);
    foreach (Type type in DLL.ExportedTypes.Where(t => t != null &&
t.BaseType ==
                                                    typeof(IPlugin)))
    {
        var plugin = Activator.CreateInstance(type) as IPlugin;

        if (plugin is not null)
        {
            plugins.Add(plugin);
        }
    }

    return plugins;
}
```

Відповідно до реалізації, отриманий на вхід шлях до файлу опрацьовується та використовується для завантаження збірки за допомогою функції Assembly.LoadFile. Після успішного завантаження відбувається пошук відкритих типів, класів та структур, які успадковують абстрактний клас IPlugin. Як результат пошуку повертається список елементів, які відповідають вказаній умові та здійснюється їх подальша перевірка:

- якщо у вказаній збірці немає елементів, що успадковують абстрактний клас IPlugin, то метод повертає порожній список та продовжується адаптація інших компонент;
- якщо у збірці існує хоча б один елемент, що успадковує абстрактний клас IPlugin, тоді відбувається ітерація по цих елементах. Кожен елемент активується за допомогою функції Activator**.**CreateInstance та зберігається у списку завантажених елементів. Після опрацювання останнього елементу функція GetPluginsFromFile повертає список доступних модулів та інформацію про їх тип.

Окрім можливості динамічного визначення елементів адаптації створена функція не має жорсткої прив’язки до типу або інформації про середовище виконання конкретного модуля. Це дозволяє створювати об’єкти на основі уніфікованого інтерфейсу та його методів, а реалізація доступу та запуску покладається на саму компоненту.

На відміну від методу адаптації функціональності програмної системи, визначення та завантаження графічних елементів є простішим процесом. Проте, під час адаптації графічних елементів програмного забезпечення використання принципів та елементів рефлексії не є доцільним. Більшість графічних елементів або сторінок у бібліотеці .NET MAUI подано у XML або XAML форматах, які можна доєднати без додаткових залежностей. Винятком є сторінки з виконуваним кодом, проте вони класифікуються у адаптивній програмній системі як функціональні компоненти.

Реалізація алгоритму визначення та реєстрації допоміжних графічних компонент наведено у фрагменті коду:

```
public static async Task
LoadResources(IEnumerable<IUserInterfaceElement>
userInterfaceElements)
{
    foreach (var element in userInterfaceElements)
    {
        await element.LoadElementWithDefaultResources();
    }
}
```

Таким чином під час адаптування графічних елементів використовується лише метод LoadElementWithDefaultResources, що є методом абстрактного класу IUserInterfaceElement. Цей метод так само як і для функціональних компонент дозволяє відділити від батьківської компоненти логіку застосування графічних елементів, що в свою чергу зменшує залежність між модулями програмної системи.

Реалізація бібліотеки адаптації програмної системи наведена в Додатку А.

4.1.2. Налаштування брокера повідомлень RabbitMQ

У процесі адаптації програмного забезпечення для зниження та розподілу навантаження на елементи програмної системи використано архітектурний шаблон брокерів повідомлень. Окрім того, використання спеціалізованих типів обміну дозволяє не лише краще опрацьовувати помилки адаптації, але і забезпечує можливість збереження запитів та повторної адаптації у разі виникнення виняткових ситуацій.

Налаштування обміну повідомлень між прикладним програмним інтерфейсом та сервісом бази даних та знань здійснювалося за допомогою платформи RabbitMQ. На відміну від інших платформ, RabbitMQ забезпечує доступ до налаштувань обміну повідомлень низького рівня, що в свою чергу дозволяє налаштовувати обмін більш гнучко.

Першим етапом до формування обміну повідомлень для адаптивної програмної системи є визначення основного обміну та черги, в яку спрямовуватимуться повідомлення. Приклад оголошення та ініціалізації елементів брокера повідомлень наведено у фрагменті коду:

```
_channel.ExchangeDeclare(RabbitMQConstants.WorkExchangeName,
                 RabbitMQConstants.DirectExchangeType);
_channel.QueueDeclare(
      queue: _rabbitMQOptions.QueueName,
      durable: true,
        exclusive: false,
        autoDelete: false,
        arguments: null);
_channel.QueueBind(_rabbitMQOptions.QueueName,
RabbitMQConstants.WorkExchangeName, string.Empty, null);
```

Відповідно до реалізації, спершу оголошується обмін за типом *direct*, який буде напряму спрямовувати запити в чергу. Після цього оголошується черга з параметрам *durable = true* та *autoDelete = false*. Такі налаштування черги забезпечуватимуть можливість збереження повідомлень у черзі в разі, якщо зникне з'єднання з брокером повідомлень, а також необхідність підтвердження успішного отримання та опрацювання відповідного повідомлення про адаптацію. Останнім кроком є зв'язування створеного обміну та черги повідомлень, що дозволить автоматично переспрямовувати будь яке повідомлення з обміну в чергу, без необхідності використання шаблону ключа маршрутизації.

Первинна конфігурація дозволяє зменшити навантаження на ресурси програмної системи шляхом почергового опрацювання процесу визначення конфігурації програмного забезпечення. У випадку здійснення поділу цього процесу за типом середовища виконання програмного застосунку необхідно вказати цей тип у додаткових налаштуваннях черги (arguments). Така конфігурація в свою чергу забезпечить можливість фільтрування повідомлень та їх перенаправлення до відповідного сервісу.

Для забезпечення можливості збереження повідомлень у разі виникнення виняткових ситуацій необхідно визначити тип обміну *dead-letter*. Приклад оголошення та ініціалізації обміну *dead-letter* наведено у фрагменті коду:

```
var queueArgs = new Dictionary<string, object>
{
      { RabbitMQConstants.DeadLetterExchangeHeader,
      RabbitMQConstants.WorkExchangeName },
        { RabbitMQConstants.TTLHeader, 60_000 },
};

_channel.ExchangeDeclare(RabbitMQConstants.RetryExchangeName,
                              RabbitMQConstants.DirectExchangeType);
_channel.QueueDeclare(RabbitMQConstants.RetryQueueName, true,
false, false, queueArgs);
_channel.QueueBind(RabbitMQConstants.RetryQueueName,
                      RabbitMQConstants.RetryExchangeName,
                      string.Empty, null);
```

Початковим етапом створення обміну *dead-letter* є визначення додаткових аргументів черги. Основними елементами є спеціальний заголовок – *x-dead-letter-exchange*, що визнає з якого обміну повідомлення будуть потрапляти у додаткову чергу. Також доцільно використовувати заголовок – *x-message-ttl*, що дозволяє встановлювати максимальний час перебування повідомлення у додатковій черзі перед його перенаправленням назад в основний обмін. Після встановлення додаткових налаштувань для реалізації підходу *dead-letter* необхідно також створити додаткову чергу, в якій будуть міститися повідомлення після помилок, а також додатковий обмін, який власне спрямовуватиме повідомлення у вказану чергу.

Проте, таке налаштування обміну *dead-letter* має значний недолік – у разі постійного виникнення помилок повідомлення опиниться в циклі. Це може спричинити зростання кількості запитів до сервісу адаптації, а також опрацювання застарілої інформації, у разі якщо користувач надішле новий запит на модифікацію конфігурації системи. В такому разі доцільно встановити максимальну кількість повторень та перевіряти цей параметр під час опрацювання повідомлення. Приклад реалізації цієї функціональності наведено у фрагменті коду:

```
if (result.IsFailure)
{
    _logger.LogError($"Cant process the message: {result.Error}");
    long count = 0;
    if (message.BasicProperties.Headers is not null &&
                        message.BasicProperties.Headers.ContainsKey("
                    x-death"))
    {
      var deathProperties =
    (List<object>)message.BasicProperties.Headers["x-death"];
      var lastRetry = (Dictionary<string,
    object>)deathProperties[0];
      count = (long)lastRetry["count"];
    }

    if (count < MaxNumberOfRetries)
    {
      _channel.BasicPublish("RETRY_EXCHANGE", string.Empty,
                                    ea.BasicProperties, body);
    }
}
```

Реалізована відповідно до вказаних принципів структура брокера повідомлень дозволить розподілити навантаження на сервіс бази даних та знань. Крім того, використання додаткових черг з максимальною кількістю повторень адаптації дозволить встановити причини виникнення виняткових ситуацій. Це в свою чергу сприятиме кращому виявленню помилок у опрацюванні даних або формуванні нових конфігурацій програмного забезпечення.

Реалізація налаштувань та взаємодії брокера повідомлень, з використанням технології RabbitMQ, наведена в Додатку Б.

4.1.3. Використання технології SignalR для побудови адаптивних застосунків реального часу

Використання принципів рефлексії та брокерів повідомлень у процесі роботи адаптивних та самоадаптивних програмних систем забезпечує можливість динамічної модифікації елементів програмного застосунку. Проте, незважаючи на ефективність брокера повідомлень та розподілення навантаження на сервіси, залишається проблема оповіщення та надсилання сформованих адаптивних налаштувань користувачеві. Вирішенням цієї проблеми є використання технологій асинхронних сповіщень, зокрема SignalR [105].

Бібліотека SignalR забезпечує можливість встановлення постійного з’єднання між клієнтською та серверною частинами програмного забезпечення [64]. У своїй реалізації бібліотека використовує вебсокети як один з основних методів обміну та транспортування повідомлень. Ці особливості дозволяють використовувати SignalR у різних випадках: для реалізації комунікації між користувачами; відстеження статистичних даних та інформаційних панелей у режимі реального часу; створення push-сповіщень; забезпечення можливості швидких оновлень.

Зазначені функціональні можливості технологій асинхронних сповіщень обумовили вибір SignalR для розробки адаптивних програмних систем. При

цьому повідомлення можуть бути розподіленими між користувачами кількома методами:

- *повідомлення надсилається на всі клієнти та для всіх користувачів*, що встановили постійне з'єднання з сервером;
- *повідомлення надсилається лише одному окремому користувачу***.** У цьому випадку враховується унікальний ідентифікатор користувача у системі;
- *повідомлення надсилається у спеціалізовані кімнати***.** У цьому випадку дані отримують всі користувачі, що приналежні до конкретної групи.

Зазначені методів доцільно використовувати у адаптивних програмних системах, у залежності від проблематики та встановлених вимог. Наприклад, для систем у яких є один тип користувачів достатньо використати найпростіший варіант розподілення повідомлень. Оскільки загальні вимоги залишаться однаковими, тоді при такому впровадженні кожен користувач отримає однаковий набір налаштувань. Приклад такого надсилання повідомлень наведено у фрагменті коду:

```
public async Task SendMessageToAllUser(SystemConfigurationDto
content)
{
    await Clients.All.SendAsync("ReceiveMessage", message);
}
```

Проте, як на нашу думку, даний метод не доцільно використовувати у випадках коли необхідно надіслати унікальні налаштування системи, оскільки буде необхідно виконувати додаткову перевірку на пристрої користувача чи повідомлення з його конфігурацією. Таке рішення зменшить зв'язність між компонентами, а також ускладнить процес модифікації програмного забезпечення. У такому разі доцільно використовувати метод розподілення повідомлень за унікальним ідентифікатором користувача. Це сприятиме зменшенню навантаження на користувацький пристрій та дозволить вносити зміни у логіку надсилання налаштувань лише на сервері:

```
public record User(string UUID, string Room);
public record Message(string User, SystemConfigurationDto
SystemConfiguration);

public async Task SendMessageToUser(string userId,
SystemConfigurationDto content)
{
    var message = new Message(userId, content);
    await Clients.User(userId).SendAsync("ReceiveMessage",
message);
}
```

Подальшим вдосконаленням методу надсилання конфігурації є створення окремих кімнат / груп для комунікації. Оскільки для більшості систем користувачі належать до певних класів (наприклад, у системі для допомоги людям з когнітивними порушеннями наявні класи: адміністратор, особа з когнітивними порушеннями та доглядальник), тому не доцільним є надсилання однакової конфігурації кожному користувачу окремо. Під час групового надсилання повідомлення всі користувачі отримають нові налаштування, тим самим зменшуючи кількість запитів на адаптацію, оскільки у разі зміни бази знань онтологічної моделі динамічно оновляться всі адаптивні налаштування для користувачів. Приклад налаштування методу групового надсилання повідомлень за допомогою технології SignalR наведено у фрагменті коду:

```
public record User(string UUID, string Room);
public record Message(string User, SystemConfigurationDto
SystemConfiguration);

public class AdaptiveSettingsSignalRHub: Hub
{
    private static ConcurrentDictionary<string, User> _users =
new();
    public override async Task OnDisconnectedAsync(Exception?
exception)
    {
        if (_users.TryGetValue(Context.ConnectionId, out var user))
        {
            await Groups.RemoveFromGroupAsync(Context.ConnectionId,
user.Room);
            await Clients.Group(user.Room).SendAsync("UserLeft",
user.UUID);
        }
    }
```

```
    public async Task JoinRoom(string userName, string roomName)
    {
        _users.TryAdd(Context.ConnectionId, new User(userName,
roomName));
        await Groups.AddToGroupAsync(Context.ConnectionId,
roomName);
        await Clients.Group(roomName).SendAsync("UserJoined",
userName);
    }

    public async Task SendMessageToRoom(string roomName,
SystemConfigurationDto content)
    {
        var message = new
Message(_users[Context.ConnectionId].UUID, content);
        await Clients.Group(roomName).SendAsync("ReceiveMessage",
message);
    }
}
```

Реалізація такої комунікації між користувачем і сервером забезпечила можливість оновлення інформації про нові налаштування системи у режимі реального часу. Поєднання технології SignalR та брокера повідомлень RabbitMQ дасть змогу зменшити кількість запитів щодо оновлення адаптивних налаштувань. Крім цього, використання групового сповіщення про результати адаптації дозволяє розподіляти та надсилати модифіковану конфігурацію програмного забезпечення для користувачів одного класу або користувачів з однаковими вимогами.

4.1.4. Опрацювання онтологічних правил з використанням технології Owlready2

Принципи рефлексії, механізм брокера повідомлень та технологій сповіщень програмних систем у реальному часі забезпечують можливість реалізації остаточного етапу адаптації програмного забезпечення – застосування отриманих модифікованих налаштувань програмного забезпечення. У випадку синхронізації та визначення формалізованих налаштувань системи використовується онтологічна модель предметної області. Оскільки онтологія представляється у кількох основних форматах

(наприклад, OWL або RDF) виникає необхідність уніфікованого опрацювання онтологічного графу та базу знань [11].

У випадку розробки адаптивної програмної системи ключовою особливістю є коректне опрацювання онтологічних правил. Для цієї задачі використовуються семантичні механізми прийняття рішень: ELK, HermiT та Pellet. У більшості правил адаптації використовуються вбудовані логічні правила SWRLB [61], що зазвичай не підтримуються семантичними рушіями. Тому у такому випадку доцільно використовувати бібліотеку Owlready2 [72, 91], оскільки дане рішення дозволяє не лише використовувати повний потенціал онтологічних правил SWRL, але й дозволяє здійснювати модифікацію як наповнення так і структури бази знань у процесі роботи програмного забезпечення.

Бібліотека Owlready2 підтримує два основних семантичних механізми прийняття рішень: HermiT та Pellet, для їх застосування потрібно використати один з двох методів:

- *sync_reasoner* – використовує стандартний механізм HertmiT;
- *sync_reasoner_pellet* – використовує модифікований та вдосконалений механізм Pellet, з можливістю застосування вбудованих логічних операторів: наприклад, *swrlb:greaterThan, swrlb:lessThanOrEquals* і ін.

За замовчуванням механізми опрацювання правил передбачають лише тимчасову зміну зв’язків та не забезпечують можливість збереження значень атрибутів. Для забезпечення можливості зміни інформації про відповідну сутність або екземпляр необхідно використати використано відповідні булеві параметри (з допустимими значеннями True або False) методів sync_reasoner:

- *infer_property_values* – забезпечує можливість модифікації та виведення об’єктних властивостей та зв’язків між концептами онтології;
- *infer_data_property_values* – забезпечує можливість модифікації та виведення властивостей даних, властивостей та атрибутів концептів онтології;

Застосування обох зазначених параметрів забезпечує успішне збереження та застосування усіх правил адаптації. Це дозволяє не тільки визначити необхідну конфігурацію для користувача але й встановлювати зв’язок між відповідною сутністю програмної системи та користувачем, що забезпечує можливість використання існуючої конфігурації на новому пристрої.

Перед запуском механізму опрацювання онтологічних правил спершу потрібно пересвідчитися, що усі дані про користувача є актуальними та не потребують модифікації. Після перевірки інформації здійснюється генерація налаштувань та формування результуючої відповіді у форматах JSON або XML. Таким чином, процес визначення нової конфігурації для користувача реалізовується наступною послідовністю кроків:

1. **Визначення та опрацювання інформації користувача** – на цьому етапі здійснюється пошук інформації про користувача та наявні конфігурації. Якщо користувач здійснює конфігурацію програмного забезпечення відповідно до нових вимог, тоді у базі знань замінюються старі зв’язки між компонентами Requirements відповідно до оновлених даних. Результат оновлення зберігається у онтологічній моделі:

```
ontology_user = next(i for i in ontology['User'].instances() if 
user_uuid in i.UUID)

if requirements_uuid:
    if ontology_user.__getattr__('need_configuration'):
        ontology_user.__setattr__("need_configuration", None)
    if ontology_user.__getattr__('has_requirement'):
        ontology_user.__setattr__("has_requirement", [])
         requirements = [requirement for requirement in 
ontology['Requirement'].instances() if requirement.Req_UUID in 
requirements_uuid]
    for requirement in requirements:
                create_relationship(ontology_user, requirement, 
'has_requirement')
    save_ontology(ontology, filepath)
```

2. **Здійснення синхронізації та опрацювання онтологічних правил –** даний процес виконується для встановлення нових зв’язків між

компонентами програмного забезпечення та сутністю користувача у базі знань. Це здійснюється з метою збереження конфігурації у разі якщо користувач буде змінювати пристрої у межах однієї платформи. Наприклад, перенесення даних з Android пристрою на інший пристрій платформи Android:

```
with ontology:
    sync_reasoner_pellet(debug=True, infer_property_values = True,
                                 infer_data_property_values = True)
```

3. **Формування результуючої конфігурації** – для відповідного користувача отримується інформація про необхідні модулі та графічні компоненти відповідно до встановленої на попередньому кроці конфігурації. Результатом є набір формалізований налаштувань, який готовий для застосування у відповідній програмній реалізації:

```
assembly_config = {}
assembly = ontology_user.__getattr__('need_configuration')
for prop in list(assembly.get_properties()):
    if is_data_property(prop):
        assembly_config[prop.get_python_name()] =
assembly.__getattr__(prop.get_python_name())[0]
configs['Assembly'] = assembly_config

modules = {}
graphics = {}
for use in
ontology_user.__getattr__('need_configuration').contains:
    module_data = {}
    for prop in list(use.get_properties()):
        module_data[prop.get_python_name()] =
use.__getattr__(prop.get_python_name())[0]
    if isinstance(use, ontology.Functional_Component) or
issubclass(type(use), ontology.Functional_Component):
        modules[use.__class__.__name__] = module_data
    if isinstance(use, ontology.Graphical_Component) or
issubclass(type(use), ontology.Graphical_Component):
        graphics[use.__class__.__name__] = module_data

configs['Modules'] = modules
configs['Graphics'] = graphics
```

Така реалізація процесу визначення адаптивних налаштувань програмного забезпечення надає гнучкість обробки згенерованої конфігурації.

Оскільки дані залишаються збереженими як у онтологічній базі знань так і у подальшому можуть бути синхронізованими з базою даних системи, то у разі відмови одного із сервісів завжди є доступ до актуальної інформації користувача. Крім того, використання даного процесу у поєднанні з брокером повідомлень дозволить розподілити навантаження на сервіс та забезпечить можливість здійснити горизонтальне масштабування онтологічної моделі за типом програмної платформи.

Реалізація взаємодії з онтологічною моделлю та опрацювання її правил наведена в Додатку В.

4.2. Порівняння абстрактного та класичного підходів проєктування онтологічних моделей відповідно до метрик якості

Процес оцінки якості онтології є комплексною задачею, що полягає у визначенні ключових властивостей онтологічного графу та взаємодії між його вершинами. Окрім аналізу структурних особливостей онтологічної моделі необхідно також здійснити оцінку метрик, пов’язаних з наповненням екземплярами сутностей для онтологічної бази знань [77].

Таким чином, для здійснення оцінки якості спроєктованих онтологічних моделей доцільно використовувати три основні групи метрик (структурні, функціональні та метрики профілю зручності використання), які дозволяють проаналізувати онтологію відповідно до таких атрибутів як: синтаксична якість; відповідність правилам; насиченість і семантична якість; інтерпретабельність; послідовність; ясність і прагматична якість; вичерпність, точність та актуальність.

Використовуючи описані в розділі 1.4 метрики якості онтологічних моделей, можна визначити основні структурно ієрархічні властивості, залежності та наповненість онтологічних моделей, та здійснити порівняння ергономічності та коректності класичного та абстрактного підходів до проєктування онтологічної моделі предметної області.

На рис. 4.1 – 4.2 наведено онтологічні графи, відповідно, для класичного

та абстрактного підходів побудови онтологічних моделей, що є базою для проведення обчислень значень структурно-ієрархічних метрик.

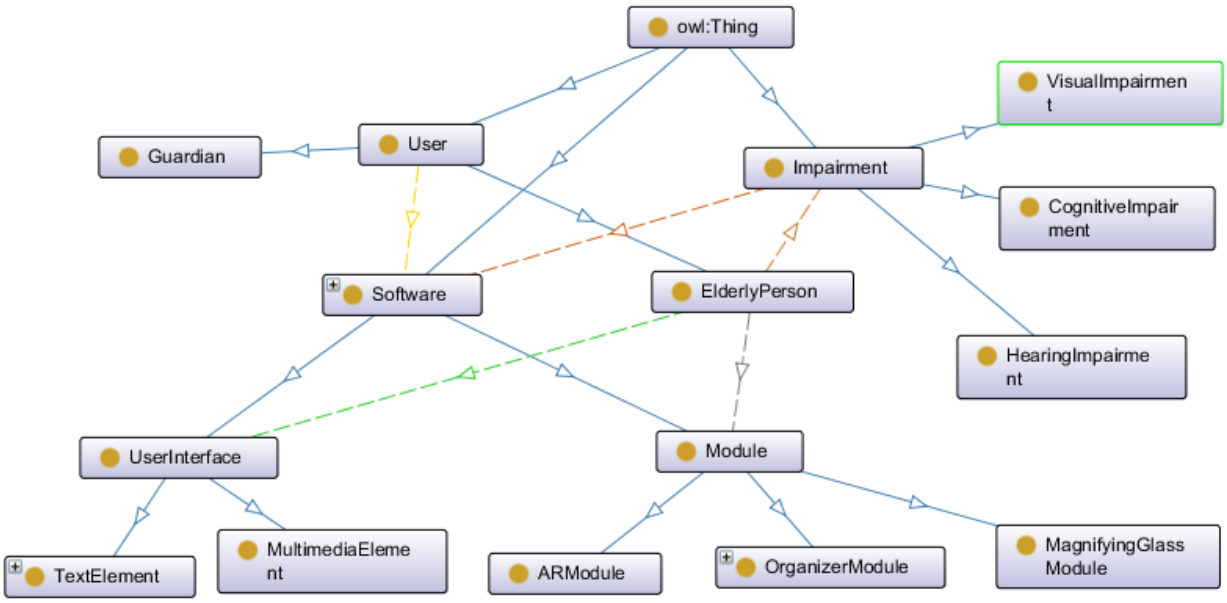

Рис. 4.2. Онтологічний граф моделі, яка спроєктована на основі класичного підходу

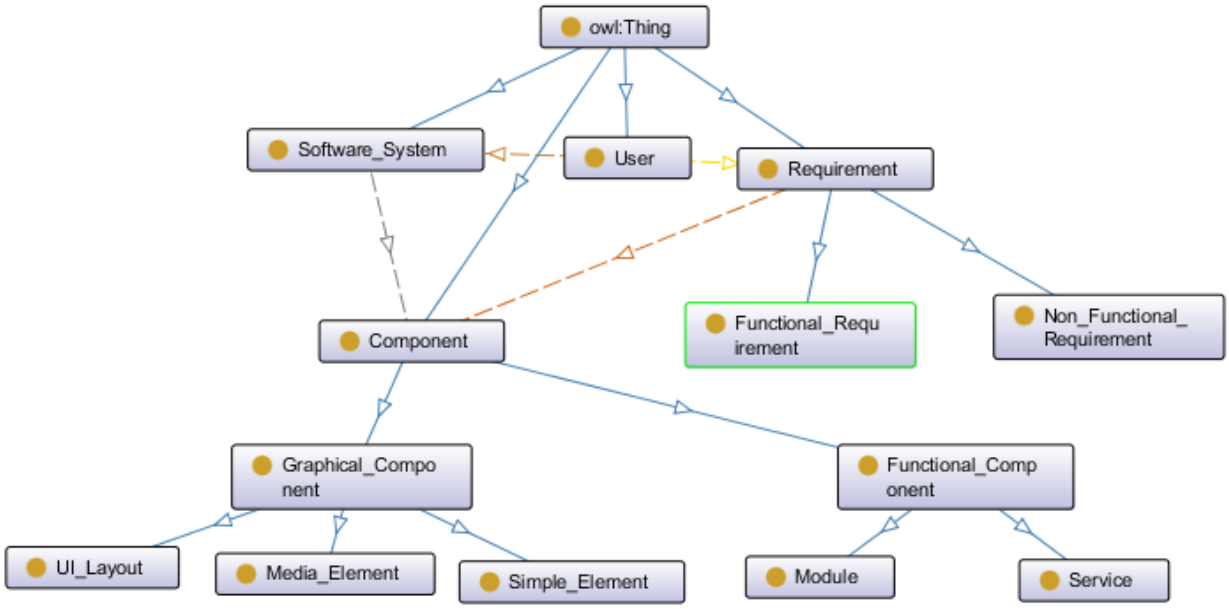

Рис. 4.3. Онтологічний граф моделі, яка спроєктована на основі абстрактного підходу

Функціональні метрики та метрики профілю зручності використання визначено на основі комбінованої інформації, яка отримана завдяки аналізу

онтологічного графу та метаданих бази знань, значення яких наведено у табл. 4.1 – 4.2.

Табл. 4.1.

Базові характеристики онтологічної моделі спроєктованої на основі класичного підходу

Назва характеристики	**Значення**
Кількість класів	15
Кількість зв’язків	5
Кількість зв’язків успадкування	12
Кількість властивостей	20
Кількість класів, що мають більше ніж один батьківський клас	0
Кількість класів, що входять в циклічний зв’язок	0
Кількість класів, що містять властивості	8
Кількість класів, що містять хоча б один екземпляр	10
Кількість екземплярів сутностей	1650

Табл. 4.2.

Базові характеристики онтологічної моделі спроєктованої на основі абстрактного підходу

Назва характеристики	**Значення**
Кількість класів	13
Кількість зв’язків	4
Кількість зв’язків успадкування	10
Кількість властивостей	18
Кількість класів, що мають більше ніж один батьківський клас	0
Кількість класів, що входять в циклічний зв’язок	0
Кількість класів, що містять властивості	6
Кількість класів, що містять хоча б один екземпляр	9
Кількість екземплярів сутностей	1650

Результати визначених базових структурно-ієрархічних метрик, метрик схеми та метрик наповнення бази знань, що обчислено відповідно до (1.1) – (1.6), зведено у табл. 4.3 та табл. 4.4.

Табл. 4.3.

Значення базових структурно-ієрархічних метрик для підходів проєктування онтологічних моделей

Метрика	Назва підходу	Значення метрики
Метрики глибини		
Абсолютна глибина (A_{depth})	Класичний підхід	32
	Абстрактний підхід	27
Середня глибина (Avg_{depth})	Класичний підхід	2,14
	Абстрактний підхід	2,07
Метрики ширини		
Абсолютна ширина ($A_{breadth}$)	Класичний підхід	15
	Абстрактний підхід	13
Середня ширина ($Avg_{breadth}$)	Класичний підхід	2,5
	Абстрактний підхід	2,6
Метрики заплутаності та циклів		
Показник заплутаності онтології (Tang)	Класичний підхід	0,0
	Абстрактний підхід	0,0
Показник циклічності онтології (A_{cycle})	Класичний підхід	0,0
	Абстрактний підхід	0,0

Оцінка отриманих метрик доводить, що показники структурно-ієрархічних метрик для абстрактного підходу у середньому на 10% є нижчими в порівнянні з класичним. Це вказує на покращення ергономіки, розуміння структури та зв’язків онтологічної моделі, що в подальшому забезпечує швидше та більш коректне опрацювання SWRL правил у процесі адаптації.

Окрім того, класичний підхід передбачає статичну структуру моделі, що в свою чергу означатиме необхідність зміни структури у разі впровадження нових функціональних або графічних компонент адаптивної програмної системи. Абстрактний підхід вирішує цю проблему шляхом визначення екземплярів існуючих сутностей та їх подальше розширення за допомогою додаткових властивостей.

Табл. 4.4.

Значення метрик схеми та метрик наповнення бази знань для підходів проєктування онтологічних моделей

Метрика	Назва підходу	Значення метрики
Метрики схеми		
Насиченість атрибутів (AR)	Класичний підхід	1,33
	Абстрактний підхід	1,38
Насиченість зв’язків (RR)	Класичний підхід	0,294
	Абстрактний підхід	0,285
Співвідношення класів-атрибутів (ACR)	Класичний підхід	0,53
	Абстрактний підхід	0,46
Метрики наповнення бази знань		
Середнє наповнення класів (AP)	Класичний підхід	110
	Абстрактний підхід	127
Насиченість класів (CR)	Класичний підхід	0,67
	Абстрактний підхід	0,69

Порівняння підходів до проєктування онтології на основі метрик схеми та наповнення бази знань показує що для абстрактного підходу збільшуються показники насиченості та середнє наповнення класів. Такі значення вказують на більшу інформативність нової онтологічної моделі і можливість її кращого розширення. Незначна зміна метрик схеми пояснюються створенням абстракцій для класів та зв’язків, що в подальшому забезпечують можливість динамічної модифікації екземплярів сутностей у залежності від конкретної реалізації адаптивної програмної системи.

4.3. Аналіз тривалості адаптації програмної системи залежно від її конфігурації

Процес адаптації згідно розробленого нами методу відбувається після перевірки даних користувача та їх реєстрації в онтологічній базі знань. Для формування персоналізованих налаштувань системи створюються екземпляри відповідних концептів онтологічної моделі, а також встановлюються зв’язки між створеними концептами предметної області. На основі семантичних правил приймаються рішення про необхідність призначення конкретних параметрів адаптації. Результатом цього підходу є набір параметрів, які потім використовуються для персоналізації системи. При модифікації мобільного додатку паралельно відбуваються два окремі процеси: персоналізація інтерфейсу користувача та модифікація функціональності програмної системи.

Згідно класичного підходу, формування моделі та налаштувань програмної системи полягає у визначенні конкретних сутностей предметної області. Незважаючи на високу швидкодію опрацювання, такий підхід втрачає ефективність, якщо структура онтологічної моделі є змінною. На прикладі реалізації онтологічної моделі відповідно до класичного підходу [48] можна помітити, що час визначення оптимальних характеристик системи лінійно залежить від загальної кількості екземплярів сутностей онтологічної моделі. Проте, швидкодія втрачається, якщо необхідно розширити програмну систему додатковими модулями, оскільки необхідно змінювати структуру онтологічної моделі, що в свою чергу створює додаткові зв’язки між елементами.

Розроблений абстрактний підхід усуває вказану проблему. Вдосконалена онтологічна модель визначає основні абстракції серед понять та сутностей предметної області. Таке рішення дозволяє, в свою чергу, залишити структуру онтологічної моделі незмінною, а її розширення забезпечується завдяки створенню нових екземплярів існуючих сутностей.

Аналіз процесу динамічної адаптації функціональності та інтерфейсу користувача здійснено на основі створеного прототипу адаптивної програмної

системи [77]. Для аналізу швидкодії процесу визначення оптимальних характеристик для класичного підходу та розробленого абстрактного підходу проєктування онтологічних моделей визначено тривалість опрацювання онтологічних правил. У процесі визначення тривалості адаптації було використано 5 пристроїв, що дозволило виконувати генерацію конфігурації для різних середовищ виконання. Тривалість адаптації визначалася з моменту надсилання запиту на вебсервіс до остаточного застосування отриманих параметрів. Тестування системи проводилося на кожному пристрої 5 разів, щоб визначити найгірший час адаптації в залежності від кількості сутностей онтологічної моделі. Результати визначення тривалості налаштувань програмної системи наведено у табл. 4.5.

Табл. 4.5.

Тривалість (*t*, *c*) визначення налаштувань програмної системи при класичному та абстрактному підходах

Кількість об'єктів, n		**100**	**150**	**250**	**450**	**850**	**1650**	**3300**
Класичний підхід	t_1, *c*	2,55	3,08	3,46	3,97	5,16	6,78	11,5
	t_2, *c*	2,65	2,84	3,14	4,17	4,69	6,63	11,4
	t_3, *c*	2,61	2,92	3,12	3,68	4,46	6,67	11,8
	t_4, *c*	2,74	2,84	3,13	3,58	4,58	6,65	11,2
	t_5, *c*	2,52	2,86	3,17	3,91	4,85	6,02	11,8
	Середній час, t_{avg}, *c*	2,61	2,91	3,20	3,86	4,75	6,55	11,54
Абстрактний підхід	t_1, *c*	2,30	2,73	2,66	3,24	3,23	3,99	6,55
	t_2, *c*	2,20	2,21	2,82	3,05	3,47	4,39	6,96
	t_3, *c*	2,52	2,23	2,61	2,80	3,64	4,40	6,66
	t_4, *c*	2,55	2,85	2,75	2,94	3,59	4,32	6,59
	t_5, *c*	2,65	2,41	2,68	2,96	3,34	4,46	7,47
	Середній час, t_{avg}, *c*	2,44	2,49	2,70	3,00	3,45	4,31	6,85

На рис. 4.4. наведено гістограму для порівняння тривалості визначення налаштувань програмної системи при використанні класичного та абстрактного підходів.

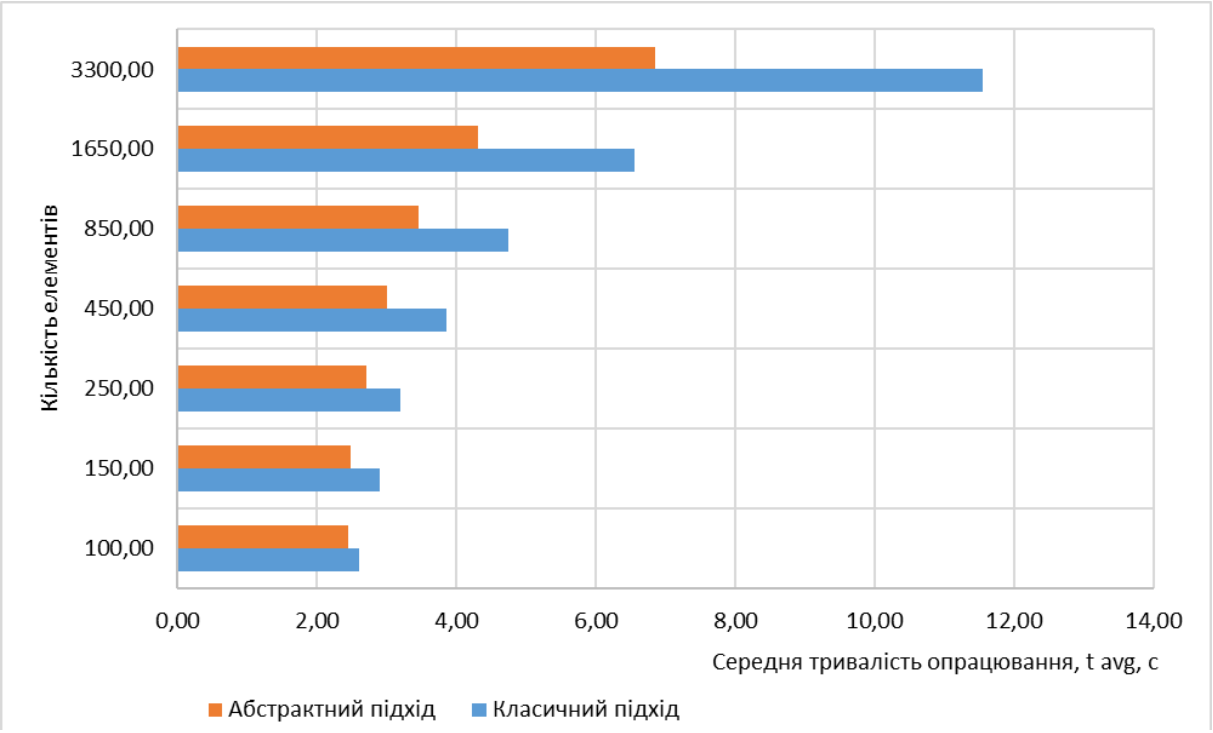

Рис. 4.4. Порівняння тривалості визначення налаштувань програмної системи при класичному та абстрактному підходах

Аналіз результатів свідчить, що абстрактний підхід у всіх тестових випадках забезпечує вищу швидкодію адаптації у порівнянні з класичним підходом: тривалість визначення налаштувань програмної системи при абстрактному підході, в середньому на 23% нижча ніж при класичному. Крім того, зі збільшенням кількості елементів та концептів онтологічної моделі, абстрактний підхід демонструє кращі результати.

Отримані значення вказують на незначне збільшення тривалості персоналізації параметрів зі збільшенням кількості екземплярів від 100 до 450. На подальше зростання тривалості адаптації впливає збільшення кількості зв’язків і властивостей для екземплярів сутностей онтологічної моделі, оскільки при обробці онтологічних правил, враховується можливість присвоєння необхідних характеристик екземплярам сутностей.

Здійснено порівняльний аналіз результатів значень швидкодії процесу визначення налаштувань програмної системи на основі онтологічних правил та зв'язків (табл. 4.6.). Встановлено, що для обох підходів характерною є тенденція зростання швидкодії визначення налаштування програмної системи зі збільшенням кількості елементів та концептів онтологічної моделі (рис. 4.5).

Табл. 4.6.

Швидкодія (*екз/с*) визначення налаштувань програмної системи при класичному та абстрактному підходах

К-сть об'єктів моделі, *n*	**Швидкодія адаптації *(екз/с)***	
	Класичний	**Абстрактний**
100	38,31	40,98
150	51,55	60,24
250	78,13	92,59
450	116,58	150,00
850	178,95	246,38
1650	251,91	382,83
3300	285,96	481,75

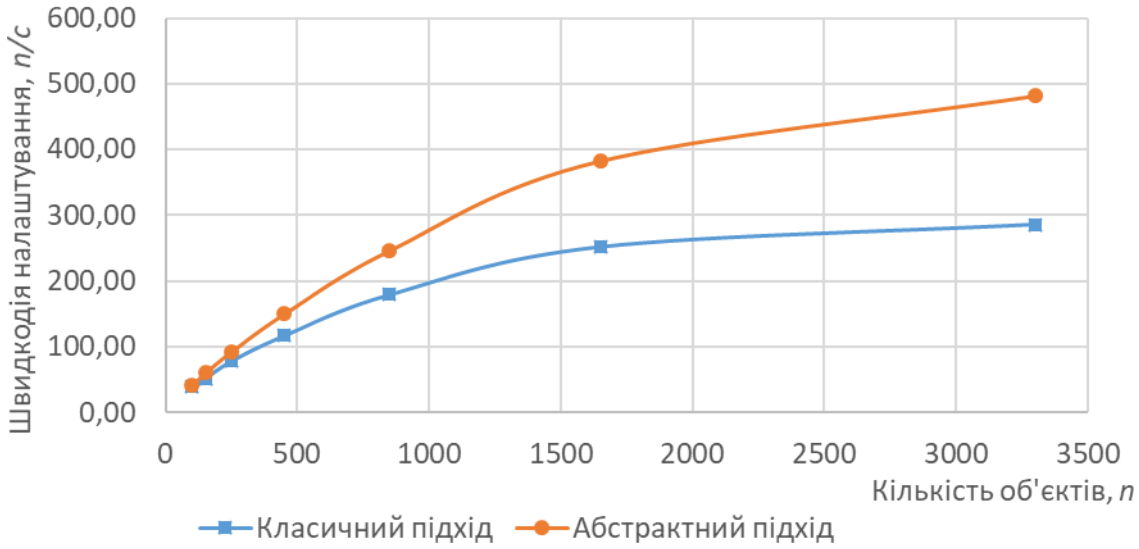

Рис. 4.5. Швидкодія визначення налаштувань програмної системи

Аналіз результатів свідчить, що при опрацюванні більше 3000 елементів онтології, абстрактний підхід демонструє майже вдвічі кращі показники швидкодії, що підтверджує його вищу продуктивність.

Таким чином, абстрактний підхід до проєктування онтологічної моделі дозволив пришвидшити процес адаптації та опрацювання онтологічних правил. Проте, як зазначалося раніше процес опрацювання правил є синхронним і однопотоковим. Це означатиме, що незважаючи на доволі високу швидкість адаптації, головний потік процесу адаптації буде блокуватися кожного разу при отриманні запиту на оновлення конфігурації. Вирішення даної проблеми можливе шляхом використання вертикального та горизонтального масштабування для онтологічних моделей.

Вертикальне масштабування передбачає збільшення та покращення ресурсів системи, що зазвичай потребує перевантаження сервісу. Проте, це не є оптимальним рішенням, оскільки таке масштабування буде необхідно проводити щоразу при збільшенні кількості сутностей в онтологічній моделі. Альтернативним рішенням є горизонтальне масштабування та поділ однієї онтологічної моделі на кілька підмоделей, що відповідають певному критерію. Для адаптивних програмних систем такий розподіл доцільно проводити на основі типу адаптованої системи. В такому разі здійснюється оптимізація не лише швидкості адаптації, але і створюється можливість розподілення процесу опрацювання онтологічних правил на кілька сервісів.

Отже, згідно принципів горизонтального масштабування проведено поділ онтологічної моделі відповідно до типу системи: веб, мобільний та настільний застосунок. Оскільки у початковій версії онтологічна модель є єдиною та неподільною, розподіл сутностей можна здійснювати двома методами: вручну та автоматизовано. Автоматизований підхід дозволяє не лише швидше витягнути необхідні екземпляри сутностей та властивості предметної області, але й зменшує складність даного процесу, оскільки не потребує залучення додаткових фахівців для опрацювання онтології.

Автоматизований підхід до розподілу екземплярів сутностей та

властивостей онтологічної моделі, схема якого представлена на рис. 4.6, реалізовано згідно наступної послідовності кроків:

1. Визначення критерію поділу – у випадку адаптивного програмного забезпечення за критерій поділу доцільно вибрати тип програмної системи;
2. Створення підмоделей онтології предметної області – опрацювання даних відбувається у паралелі для зменшення часу опрацювання онтологічних записів:
   2.1. Визначення записів, що підлягають критерію поділу для конкретного типу програмної системи;
   2.2. Вилучення елементів, що не увійшли в результат пошуку;
   2.3. Збереження отриманих результатів у вигляді підмоделі для конкретного типу програмної системи.

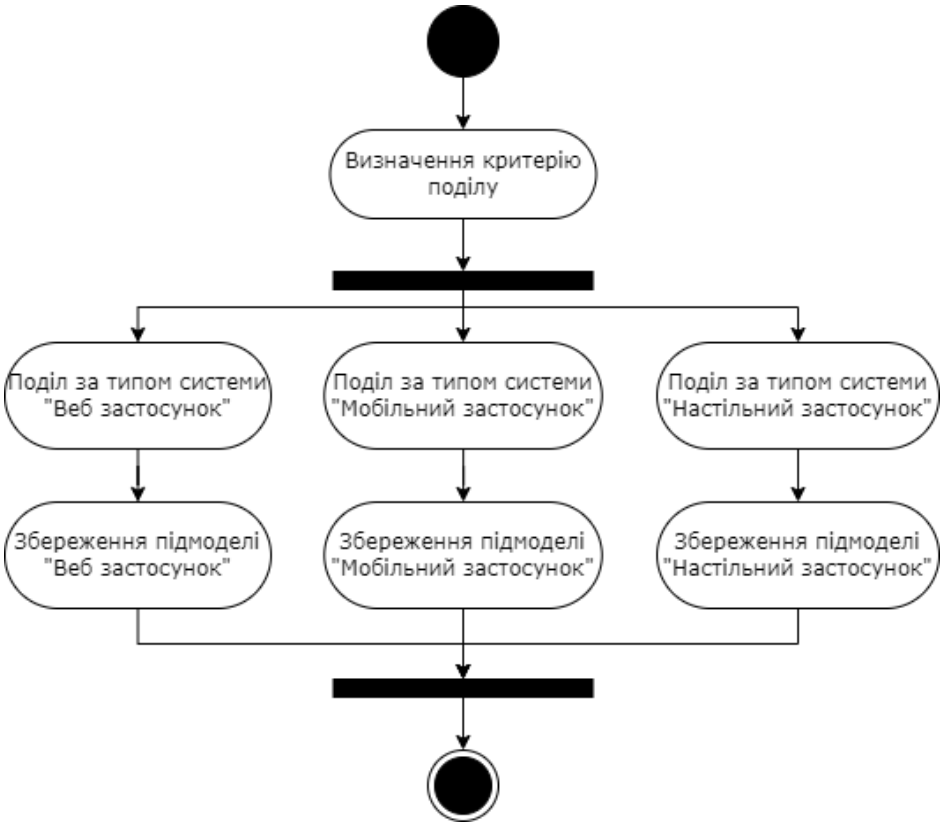

Рис. 4.6. Схема методу автоматизованого поділу онтологічної моделі за критерієм – “тип програмної системи”

Таким чином, використання автоматизованого поділу онтологічної

моделі забезпечує можливість здійснювати автоматизований поділ онтологічної моделі у процесі початкового розгортання або власне роботи сервісу бази даних та знань. Відповідно якщо у основній моделі відбудуться зміни у структурі чи наповненні ми матимемо можливість переформувати підмоделі відповідно до критерію поділу у режимі реального часу без необхідності зупинки роботи сервісу. Крім цього, таке рішення дозволить ефективніше використовувати елементи брокера повідомлень, оскільки ми матимемо змогу налаштовувати черги адаптації відповідно до критеріїв поділу.

Приклад автоматизованого поділу онтологічної моделі за допомогою інструментів мови програмування Python та технології Owlready2 наведено у фрагменті коду:

```
from owlready2 import *
def ontology_horizontal_scaling(ontology_path: str, scaling_type:
str):
    onto = get_ontology(ontology_path).load()
    for indiv in onto["Software_System"].instances():
        if scaling_type not in indiv.AssemblyVersion:
            destroy_entity(indiv)
    onto.save("./Adaptive_system_v2_web.owl")
    return onto

onto_web = ontology_horizontal_scaling("./Adaptive_system_v2.owl",
"web")
onto_mobile =
ontology_horizontal_scaling("./Adaptive_system_v2.owl", "mobile")
onto_desktop =
ontology_horizontal_scaling("./Adaptive_system_v2.owl", "desktop")
```

Для здійснення аналізу розробленого методу визначення налаштувань програмної системи на основі абстрактного підходу з використанням горизонтального масштабування проведено серія експериментальних налаштувань на базі розробленого прототипу програмної системи. Поділ моделі здійснювався на основі таких типів програмного забезпечення: вебзастосунок, мобільний застосунок, настільний застосунок. В результаті такого поділу у підмоделях сформовано конфігурації, компоненти та вимоги, що стосуються лише попередньо визначеного типу програмного забезпечення. Результати тривалості опрацювання онтологічних правил наведено у табл. 4.7.

Табл. 4.7.

Тривалість (t, c) визначення налаштувань програмної системи при абстрактному підході з використанням горизонтального масштабування

Кількість об'єктів, n	**100**	**150**	**250**	**450**	**850**	**1650**	**3300**
t_1, c	2,12	2,82	2,61	2,50	2,80	2,90	4,37
t_2, c	2,45	2,18	2,19	2,60	2,65	2,85	4,05
t_3, c	2,16	2,25	2,25	2,45	2,64	2,82	3,90
t_4, c	2,20	2,36	2,26	2,45	2,70	2,85	4,31
t_5, c	2,35	2,16	2,40	2,55	2,65	3,10	4,21
Середній час, t_{avg}, c	2,26	2,35	2,34	2,51	2,69	2,90	4,37

Здійснено порівняльний аналіз результатів тривалості виконання методу визначення налаштувань програмної системи (рис. 4.7.) для класичного та абстрактного підходів (табл. 4.5), а також для абстрактного підходу з використанням горизонтального масштабування онтологічної моделі (табл. 4.7).

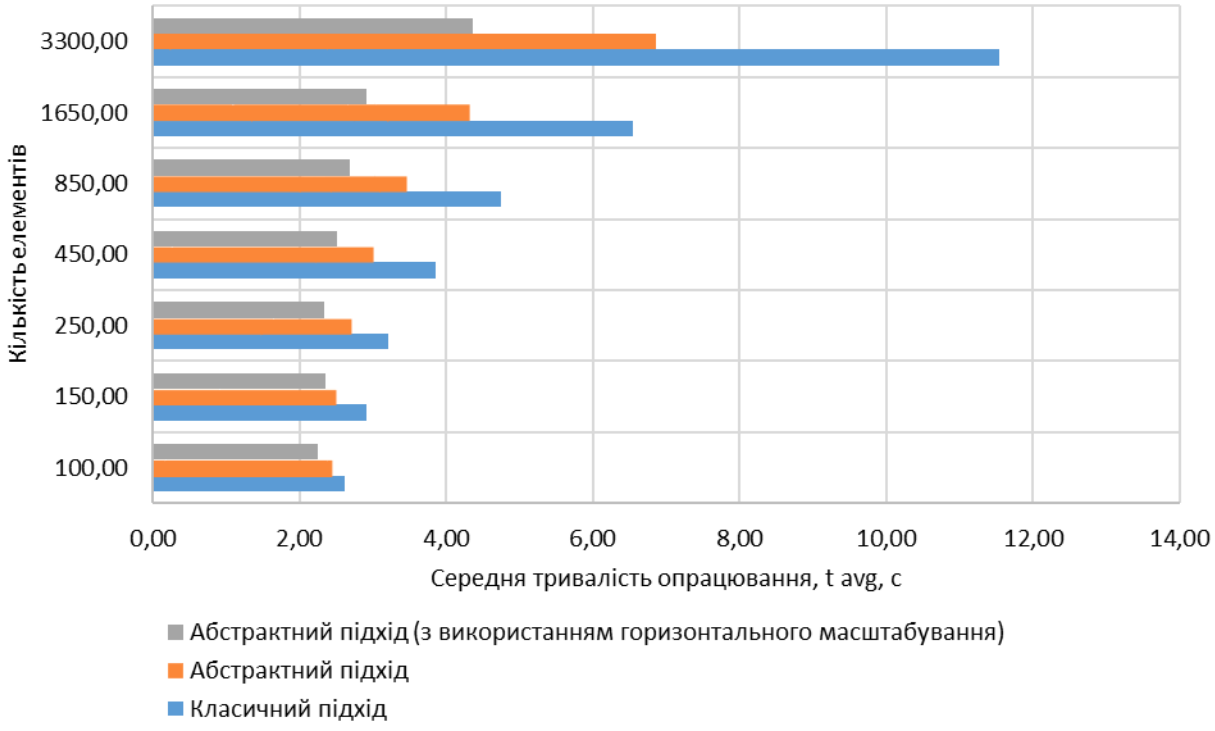

Рис. 4.7. Порівняння тривалості визначення налаштувань прототипу програмної системи з використанням розроблених підходів

Результати аналізу доводять, що використання принципів горизонтального масштабування для онтологічних моделей при абстрактному підході дає змогу зменшити тривалість визначення налаштувань в середньому на 19 %, за рахунок розподілу запитів між різними підмоделями

Для виявлення причинно-наслідкового зв'язку між кількістю елементів онтології (екземплярів сутності) та тривалістю адаптації використано метод регресійного аналізу, що дозволяє виявити вплив факторної ознаки (кількості елементів онтології) на результативну ознаку (тривалість адаптації). За основу взято експериментальні дані за результатами визначення тривалості адаптації програмної системи для різної кількості екземплярів сутностей, що наведено в табл. 4.5 та 4.7. Графічне представлення регресійних залежностей подано на рис. 4.8.

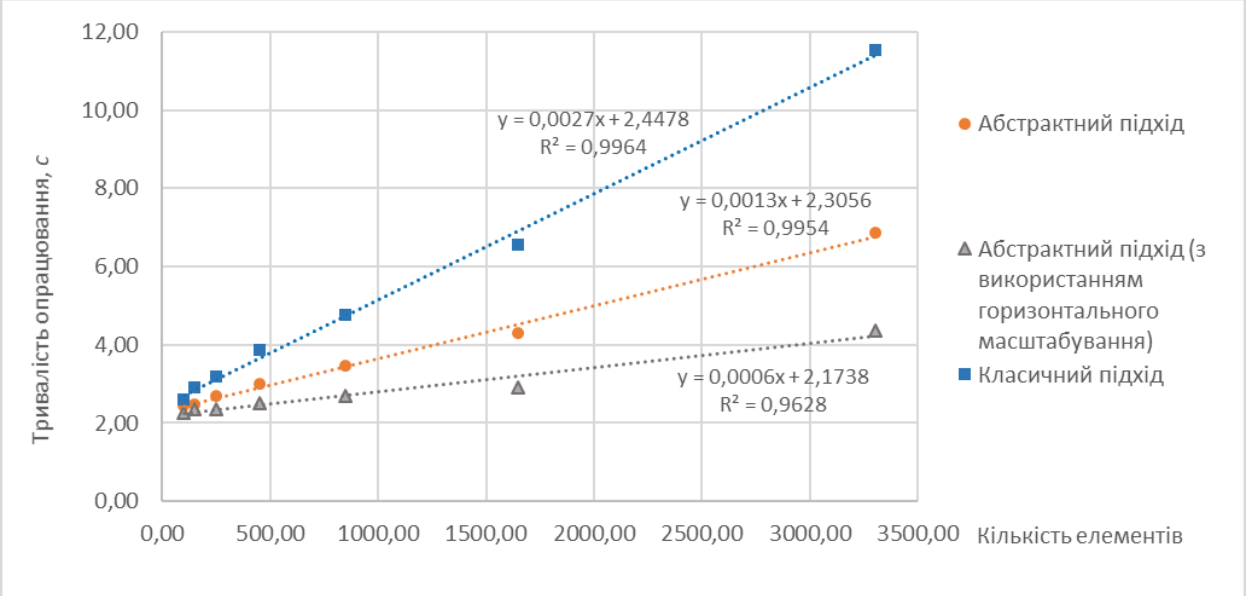

Рис. 4.8. Результати регресійного аналізу взаємозв'язку тривалості опрацювання запитів налаштувань від кількості елементів онтологічної моделі

Проаналізувавши за допомогою регресійного аналізу дані щодо тривалості адаптації з п'яти серій випробувань для однакового набору кількості екземплярів сутності згідно класичного і абстрактного підходу а також абстрактного підходу з використанням горизонтального масштабування

виявлено, що в усіх випадках з достовірністю апроксимації $R^2 > 0.9$ отримуємо лінійні залежності (4.1) – (4.3).

$$t_{cl\_apr}(x) = 0.0027N + 2.45,\ R^2 = 0.9964 \quad (4.1)$$

$$t_{abst\_apr}(x) = 0.0013N + 2.31,\ R^2 = 0.9954 \quad (4.2)$$

$$t_{abst\_scale\_apr}(x) = 0.0006N + 2.2,\ R^2 = 0.9628 \quad (4.3)$$

де: N – кількість екземплярів;

t_{cl_apr} – тривалість адаптації згідно класичного підходу;

t_{abst_apr} – тривалість адаптації згідно абстрактного підходу;

$t_{abst\_scale\_apr}$ – тривалість адаптації згідно абстрактного підходу з використанням горизонтального масштабування онтологічної моделі.

Регресійний коефіцієнт, який визначає кут нахилу лінійної залежності та указує на інтенсивність зростання тривалості адаптації в залежності від кількості елементів, який у випадку абстрактного підходу становить 0,0013, що вдвічі менше ніж у випадку класичного підходу (0,0027). Отже, абстрактний метод є менш чутливим до зростання кількості елементів, що дозволяє його використовувати у системах, які характеризуються великою кількістю зв’язків. Окрім того, даний регресійний коефіцієнт у випадку застосування абстрактного підходу з використанням горизонтального масштабування дорівнює 0,0006, що вдвічі менше ніж при абстрактному підході (0,0013). Це пояснюється тим, що відбувається поділ однієї великої бази знань на декілька, що спричиняє швидше опрацювання онтологічних правил.

Таким чином, наведені результати експериментального дослідження підтверджують доцільність використання розробленого абстрактного підходу. Поєднання даного підходу з використанням горизонтального масштабування дозволяє підвищити швидкодію налаштування адаптивних програмних систем, базова онтологічна модель якої характеризується великою кількістю зв’язків.

4.4. Висновки до розділу

Розроблено прототип адаптивної програмної системи на основі спроєктованих архітектури та онтологічної моделі предметної області. Спроєктовано та розроблено бібліотеку адаптації програмного забезпечення на основі використання принципів рефлексії, що забезпечує низький рівень зв’язності між основним програмним забезпеченням та допоміжними адаптивними компонентами. Запропоновано використання комбінованого підходу до реалізації формування повідомлень на основі типу обміну dead-letter та максимальної кількості повторень, що дозволяє знизити та розподілити навантаження на елементи програмної системи.

Проведено порівняльний аналіз абстрактного підходу та класичного підходу до проєктування онтологічних моделей адаптивних програмних систем. Встановлено, що використання абстрактного підходу зменшує показники структурно-ієрархічних метрик, що покращують розуміння структури та зв’язків онтології, а також забезпечує швидше та коректніше опрацювання онтологічних SWRL правил у процесі адаптації. Такі результати вказують на більш якісне розширення онтологічної моделі у процесі динамічної адаптації програмного забезпечення.

Проведено експериментальне дослідження швидкодії процесу визначення оптимальних характеристик системи на основі онтологічних моделей, використовуючи запропонований абстрактний підхід до проєктування онтологічної моделі та реалізації модифікації конфігурації адаптивного програмного забезпечення. Отримані результати, демонструють на 20% кращі показники швидкості генерації налаштувань програмного забезпечення, використовуючи онтологічну модель на основі абстрактного підходу з використанням принципів горизонтального масштабування для онтологічних моделей, в порівнянні з класичними підходами

ВИСНОВКИ

У дисертації розв’язано актуальну наукову задачу удосконалення процесів проєктування та розроблення адаптивних програмних систем на основі використання уніфікованої онтології, компонентно-орієнтованої архітектури, а також методу динамічної адаптації функціональності програмної системи та інтерфейсу користувача.

Основні наукові та практичні результати дисертаційної роботи:

1. Проведений аналіз методів та засобів проєктування та розроблення адаптивних програмних систем показав, що серед наявних підходів характерним є використання концептуальних моделей, які дають змогу відображати основні зв’язки та залежності предметної області. Встановлено, що використання онтологічних моделей дає змогу уніфікувати інформацію про предметну область, а також забезпечить можливість динамічної модифікації програмної системи без необхідності повторного виконання процесів реконфігурації та розгортання.

2. Розроблений метод побудови моделі адаптивної програмної на основі онтологічного підходу системи визначає основні абстракції серед компонент та зв’язків предметної області та забезпечує можливість динамічного наповнення бази знань у разі зміни системних характеристик. Використання абстракцій об’єктів предметної області дає змогу уникнути статичності структури та необхідності повторного залучення експертів предметної області у разі створення нової конфігурації програмної системи.

3. Розроблений метод визначення налаштувань програмної системи дозволяє опрацьовувати семантичні правила для окремих понять онтологічної моделі, використовуючи спроєктовану онтологічну модель та інформацію про зміни вимог та потреб користувача. У поєднанні з абстрактним підходом до проєктування онтологічної моделі це забезпечує можливість динамічного визначення налаштувань та проведення подальшої адаптації програмної системи для різних середовищ виконання.

4. Удосконалений метод динамічної адаптації функціональності програмної системи та інтерфейсу користувача у поєднанні з онтологічною моделлю дає змогу здійснювати адаптацію програмного забезпечення у режимі реального часу, а також визначати необхідні характеристики його компонент на основі даних про активний пристрій та інформації про нові або змінені вимоги користувача. Використання принципів рефлексії забезпечує можливість завантаження нових або модифікованих модулів, а також проведення динамічної адаптації без необхідності повторної реконфігурації та розгортання програмного забезпечення.

5. Удосконалений метод використання брокерів повідомлень у процесі адаптації програмного забезпечення дає змогу зменшити навантаження на сервіс адаптації за рахунок розподілу запитів до онтологічної бази знань. Встановлено, що поєднання принципів брокеру повідомлень та горизонтального масштабування дозволяють підвищити працездатність сервісу адаптації та забезпечити його безвідмовну роботи.

6. Удосконалена технологія створення адаптивних програмних систем на основі онтологічного підходу усуває статичність структури розробленої моделі та дає змогу уніфікувати процес адаптації програмних систем для різних предметних областей. Використання принципів рефлексії у поєднанні з компонентно-орієнтованою архітектурою програмної системи дає змогу забезпечити динамічність адаптації функціональності та інтерфейсу користувача без необхідності її реконфігурації та повторного розгортання.

7. Проведено експериментальне дослідження запропонованих методів створення онтологічних моделей адаптивних програмних систем. Встановлено, що використання абстрактного підходу до проєктування онтологічних моделей предметної області зменшує показники структурно-ієрархічних метрик у середньому на 10%, що покращує ергономіку, розуміння структури та зв’язків онтологічної моделі.

8. Дослідження розроблених методів, що проводились на основі створеного прототипу програмної системи доводять, що використання моделі,

побудованої на абстракціях об’єктів, покращує показники тривалості адаптації у середньому на 23 % порівняно з класичним підходом до проєктування онтологічної моделі. Використання при абстрактному підході принципів горизонтального масштабування для онтологічних моделей дає змогу, завдяки розподілу запитів між різними підмоделями, зменшити тривалість визначення налаштувань в середньому на 19 %, що забезпечує більшу швидкодію опрацювання запитів на адаптацію програмної системи та формування її конфігурації.

СПИСОК ЛІТЕРАТУРИ

1. Бугай А. А. Концептуальна модель веб-інтерфейсу користувача з використанням інтелектуальних технологій. Адаптивні системи автоматичного управління, 2018. Вип. 32, с. 15–22.

2. Буров Є., Пасічник В. Програмні системи на базі онтологічних моделей задач. Вісник Національного університету Львівська політехніка. Серія: Інформаційні системи та мережі, 2015, № 829, с. 36–57.

3. Литвин В. В. Автоматизація процесу розвитку базової онтології на основі аналізу текстових ресурсів. Вісник Національного університету «Львівська політехніка». Серія: Інформаційні системи та мережі, 2010, № 673, с. 319–325.

4. Литвин В. В., Демчук А. Б., Войчишен М. М. Метод побудови інтелектуального агента на основі онтології предметної області. Вісник Національного університету «Львівська політехніка», 2011. Вип. 715, с. 215–224.

5. Луцик І., Луцик І. Використання онтологій для програмних модулів адаптивних систем керування на базі нечіткої логіки. Матеріали VIII міжнародної науково-технічної конференції КМОСС-2023. (м. Дніпро, 1-3 листопада 2023 р). 2023, с. 115–116. URL: https://udhtu.edu.ua/wp-content/uploads/2023/11/zbirnyk-tez-kmoss-2023.pdf

6. Луцик І., Федасюк Д. Онтологічна модель адаптивної програмної системи для людей з когнітивними порушеннями. Матеріали конференцій Молодіжної наукової ліги «Теоретичне та практичне застосування результатів сучасної науки». (м. Запоріжжя, 27 листопада 2020 р) Запоріжжя, 2020, с. 49–52. https://doi.org/10.36074/27.11.2020.v2.04

7. Луцик І., Федасюк Д. Програмна система для допомоги людям старшого віку з когнітивними порушеннями. Матеріали конференції Інформаційні технології – 2020: VII Всеукраїнська науково-практична

конференція молодих науковців. (м. Київ, 21 травня 2020 р.) Київ, 2020, с. 123–125. URL: https://informationtechn2020.blogspot.com/2020/05/blog-post_19.html

8. Федасюк Д. В., Луцик І. І. Адаптивна програмна система на основі онтологічного підходу для людей з когнітивними порушеннями. Вісник Національного університету «Львівська політехніка». Серія Інформаційні системи та мережі. 2021, № 9, с. 61–74. https://doi.org/10.23939/sisn2021.09.061

9. Шаров С. В., Лубко Д. В., Осадчий В. В. Вибір моделі представлення знань у системі ІСІКС. Системи обробки інформації, 2015. Вип. 11, с. 108–111.

10. Abdelhafiz B. M., Elhadef M. Sharding Database for Fault Tolerance and Scalability of Data. 2021 2nd International Conference on Computation, Automation and Knowledge Management (ICCAKM). (Dubai, United Arab Emirates, 19.01.2021). Dubai, United Arab Emirates: IEEE, 2021, p. 17–24. https://doi.org/10.1109/ICCAKM50778.2021.9357711

11. Alaoui K., Bahaj M. Semantic Oriented Data Modeling Based on RDF, RDFS and OWL. Advanced Intelligent Systems for Sustainable Development (AI2SD'2019). Ed. Mostafa Ezziyyani. Cham: Springer International Publishing, 2020, p. 411–421. https://doi.org/10.1007/978-3-030-36674-2_42

12. Ameller D., Collell O., Franch X. The Three-Layer architectural pattern applied to plug-in-based architectures: the Eclipse case. Software: Practice and Experience, 2012. Vol. 43, No. 4, p. 391–402. https://doi.org/10.1002/spe.2142

13. Angelopoulos K., Papadopoulos A. V., Souza V. E. S., et al. Engineering Self-Adaptive Software Systems: From Requirements to Model Predictive Control. ACM Transactions on Autonomous and Adaptive Systems, 2018. Vol. 13, No 1, p. 1–27. https://doi.org/10.1145/3105748

14. Anna Victoria Oikawa C. R., Freitas V., Castro M., et al. Adaptive Load Balancing based on Machine Learning for Iterative Parallel Applications. 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). (Västerås, Sweden, 03.2020). Västerås, Sweden: IEEE, 2020, p. 94–101. https://doi.org/10.1109/PDP50117.2020.00021

15. Arndt N., Naumann P., Radtke N., et al. Decentralized Collaborative Knowledge Management Using Git. Journal of Web Semantics, 2019. Vol. 54, p. 29–47. https://doi.org/10.1016/j.websem.2018.08.002

16. Ashiwal V., Zoitl A., Konnerth M. A Service Bus Concept for Modular and Adaptable PLC-Software. 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). (Vienna, Austria, 09.2020). Vienna: IEEE. 2020, p. 22–29. https://doi.org/10.1109/ETFA46521.2020.9211908

17. Balla D., Simon C., Maliosz M. Adaptive scaling of Kubernetes pods. NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium 2020, p. 1–5. https://doi.org/10.1109/NOMS47738.2020.9110428

18. Barrette R., Grewal R. Adaptive User Interfaces and the Use of Inference Methods. Computational Science and Techniques. 2021. Vol. 8, p. 616–624. https://doi.org/10.15181/csat.v8i0.2186

19. Bhatia M. P. S., Kumar A., Beniwal R. Ontologies for Software Engineering: Past, Present and Future. Indian Journal of Science and Technology, 2016. Vol. 9, Issue 9. https://doi.org/10.17485/ijst/2016/v9i9/71384

20. Bhatia M. P. S., Kumar A., Beniwal R. Ontology based framework for automatic software's documentation. 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom) 2015, p. 421–424. URL: https://ieeexplore.ieee.org/document/7100285

21. Blinowski G., Ojdowska A., Przybyłek A. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. IEEE Access, 2022, Vol. 10, p. 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803

22. Bolock A. E., Mohamed I., Herbert C., et al. A Visual Rule Generation Tool for SWRL. RuleML+RR, 2020. URL: https://api.semanticscholar.org/CorpusID:221014752

23. Bourgey M., Dali R., Eveleigh R. та ін. GenPipes: an open-source framework for distributed and scalable genomic analyses. GigaScience, 2019. Vol. 8, No 6, article ID giz037. https://doi.org/10.1093/gigascience/giz037

24. Bravo Contreras M. C., Hoyos Reyes L. F., Reyes Ortiz J. A. Methodology for ontology design and construction. Contaduría y Administración, 2019. Vol. 64, Issue 4: 134, p. 1–24. https://doi.org/10.22201/fca.24488410e.2020.2368

25. Bruijn J. de, Lara R., Polleres A., et al. OWL DL vs. OWL flight: conceptual modeling and reasoning for the semantic Web. The Web Conference. 2005. URL: https://api.semanticscholar.org/CorpusID:2773948

26. Burov Y., Mykich K., Karpov I. Building a Versatile Knowledge-Based System Based on Reasoning Services and Ontology Representation Transformations. IEEE 15th International Conference on Computer Sciences and Information Technologies (CSIT). (Zbarazh, Ukraine, 23.09.2020). Zbarazh, Ukraine: IEEE, 2020, p. 255–260. https://doi.org/10.1109/CSIT49958.2020.9321943

27. Chaudhary N., Kumar S., Gupta S. A Novel Ontology Design and Comparative Analysis of Various Retrieval Schemes on Education Domain in Protégé. ICT Analysis and Applications. Ed. Simon Fong, Nilanjan Dey, Amit Joshi. Singapore: Springer Singapore, 2021, p. 487–495. https://doi.org/10.1007/978-981-15-8354-4_48

28. Chauhan A., Vijayakumar V., Ragala R. Towards a Multi-level Upper Ontology/ foundation Ontology Framework as Background Knowledge for Ontology Matching Problem. Procedia Computer Science, 2015. Vol. 50, p. 631–634. https://doi.org/10.1016/j.procs.2015.04.096

29. Cheng Y., Gupta A., Butt A. R. An in-memory object caching framework with adaptive load balancing. Proceedings of the Tenth European Conference on Computer SystemsEuroSys ’15: Tenth EuroSys Conference 2015. (Bordeaux France, 17.04.2015). Bordeaux France: ACM, 2015, p. 1–16. https://doi.org/10.1145/2741948.2741967

30. Chindanonda P., Podolskiy V., Gerndt M. Metrics for Self-Adaptive Queuing in Middleware for Internet of Things. IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W). (Umea, Sweden,

06.2019). Umea, Sweden: IEEE, 2019, p. 130–133. https://doi.org/10.1109/FAS-W.2019.00042

31. Chindanonda P., Podolskiy V., Gerndt M. Self-Adaptive Data Processing to Improve SLOs for Dynamic IoT Workloads. Computers. Vol. 9, No 1, p. 12. https://doi.org/10.3390/computers9010012

32. Costa N., Knorr M., Leite J. Next Step for NoHR: OWL 2 QL. The Semantic Web – ISWC 2015. Marcelo Arenas (Ed.) et al. Cham: Springer International Publishing, 2015, p. 569–586. https://doi.org/10.1007/978-3-319-25007-6_33

33. Coullon H., Henrio L., Loulergue F., et al. Component-based Distributed Software Reconfiguration:A Verification-oriented Survey. ACM Comput. Surv, 2023. Vol. 56, Issue 1, p. 1–37. https://doi.org/10.1145/3595376

34. Cuesta C. E., Romay M. P., De La Fuente P., et al. Reflection-Based, Aspect-Oriented Software Architecture. Software Architecture. Ed. F. Oquendo., B. C. Warboys., R. Morrison., et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, p. 43–56. https://doi.org/10.1007/978-3-540-24769-2_4

35. Dadeau F., Gros J.-P., Kouchnarenko O. Automated Generation of Initial Configurations for Testing Component Systems. Formal Aspects of Component Software. Ed. Gwen Salaün, Anton Wijs. Cham: Springer International Publishing, 2021, p. 134–152. https://doi.org/10.1007/978-3-030-90636-8_8

36. Debruyne C., Tran T.-K., Meersman R. Grounding Ontologies with Social Processes and Natural Language. Journal on Data Semantics, 2021. issue 2, No 2–3, p. 89–118. https://doi.org/10.1007/s13740-013-0023-3

37. Devi R., Mehrotra D., Zghal H. B., et al. SWRL reasoning on ontology-based clinical dengue knowledge base. International Journal of Metadata, Semantics and Ontologies, 2013. Issue 14, No 1, p. 39. https://doi.org/10.1504/IJMSO.2020.107795

38. Dimitrova V., Denaux R., Hart G., et al. Involving Domain Experts in Authoring OWL Ontologies. The Semantic Web – ISWC 2008. ed. A. Sheth., S.

Staab., M. Dean., et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 1–16. https://doi.org/10.1007/978-3-540-88564-1_1

39. Ding Z., Zhou Y., Zhou M. Modeling Self-Adaptive Software Systems by Fuzzy Rules and Petri Nets. IEEE Transactions on Fuzzy Systems, 2017. Vol. 26, Issue 2, p. 967–984. https://doi.org/10.1109/TFUZZ.2017.2700286

40. Dowling J., Cahill V. The K-Component Architecture Meta-Model for Self-Adaptive Software. Metalevel Architectures and Separation of Crosscutting Concerns. ed. A. Yonezawa., S. Matsuoka., G. Goos., J. Hartmanis., J. Van Leeuwen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, p. 81–88. https://doi.org/10.1007/3-540-45429-2_6

41. Duque-Ramos A., Boeker M., Jansen L., et al. Evaluating the Good Ontology Design Guideline (GoodOD) with the Ontology Quality Requirements and Evaluation Method and Metrics (OQuaRE). PLoS ONE. Ed. Georgios V. Gkoutos. Issue 9, No 8, article ID e104463. https://doi.org/10.1371/journal.pone.0104463

42. Dyvak M., Kovbasistyi A., Melnyk A., et al. Recognition of Relevance of Web Resource Content Based on Analysis of Semantic Components. 9th International Conference on Advanced Computer Information Technologies (ACIT-2019) (Ceske Budejovice, Czech Republic, 06.2019). Czech Republic: IEEE, 2019, p. 297–302. https://doi.org/10.1109/ACITT.2019.8779897

43. Dyvak M., Melnyk A., Rot A., Hernes M, Pukas A. Ontology of Mathematical Modeling Based on Interval Data / M. Dyvak et al. Complexity, 2022. Vol. 2022, p. 1–19. https://doi.org/10.1155/2022/8062969

44. Dyvak M., Pukas A., Melnyk A., Voytyuk I., Valchyshyn S. and Romanets I. Software Architecture for Modeling the Interval Static and Dynamic Objects 2021 11th International Conference on Advanced Computer Information Technologies (ACIT), Deggendorf, Germany, 2021, p. 572–575. https://doi.org/10.1109/ACIT52158.2021.9548577

45. Fedasyuk D., Lutsyk I. Approach to Implementation of Configuration Process for Adaptive Software Systems based on Ontologies. International Journal of

Computing. 2023. Vol. 22, Issue 3, p. 381–388. https://doi.org/10.47839/ijc.22.3.3234

46. Fedasyuk D., Lutsyk I. Method of modification of self-adaptive software systems based on ontology. 2022 IEEE 16th International Conference on Advanced Trends in Radioelectronics, Telecommunications and Computer Engineering (TCSET), (Lviv-Slavske, Ukraine, February 22–26, 2022). IEEE, 2022, p. 530-533. https://doi.org/10.1109/TCSET55632.2022.9766856

47. Fedasyuk D., Lutsyk I. The Use of Ontology in the Process of Designing Adaptive Software Systems. 2022 IEEE 17th International Conference on Computer Sciences and Information Technologies (CSIT), (Lviv, November 10–12, 2022). IEEE, 2022, p. 503–506. https://doi.org/10.1109/CSIT56902.2022.10000528

48. Fedasyuk D., Lutsyk I. Tools for adaptation of a mobile application to the needs of users with cognitive impairments. 2021 IEEE 16th International Conference on Computer Sciences and Information Technologies (CSIT), (Lviv, Ukraine, September 22–25, 2021). IEEE, 2021, p. 321-324. https://doi.org/10.1109/CSIT52700.2021.9648702

49. Fernandes J. L., Lopes I. C., Rodrigues J. J, P. C., et al. Performance evaluation of RESTful web services and AMQP protocol. 2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN). (DA NANG, Vietnam, 07.2013). DA NANG, Vietnam: IEEE, 2013, p. 810–815. https://doi.org/10.1109/ICUFN.2013.6614932

50. Ferreira D. R., Alves S., Thom L. H. Ontology-Based Discovery of Workflow Activity Patterns. Business Process Management Workshops. ed. Daniel F., Barkaoui K, Dustdar S. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, p. 314–325. https://doi.org/10.1007/978-3-642-28115-0_30

51. Filieri A., Maggio M., Angelopoulos K., et al. Control Strategies for Self-Adaptive Software Systems. ACM Transactions on Autonomous and Adaptive Systems, 2017. Issue 11, No 4, p. 1–31. https://doi.org/10.1145/3024188

52. Freitas A. A. C. de, Scalser M. B., Costa S. D., et al. Towards an ontology-based approach to develop software systems with adaptive user interface.

Proceedings of the 21st Brazilian Symposium on Human Factors in Computing Systems (New York, NY, USA, 2022). New York, NY, USA: Association for Computing Machinery, 2022. https://doi.org/10.1145/3554364.3559139

53. Gangemi A., Presutti V. Ontology Design Patterns. Handbook on Ontologies. ред. Steffen Staab, Rudi Studer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, p. 221–243. https://doi.org/10.1007/978-3-540-92673-3_10

54. Giunchiglia F., Dutta B., Maltese V., et al. A Facet-Based Methodology for the Construction of a Large-Scale Geospatial Ontology. Journal on Data Semantics, 2012. Issue 1, No 1, p. 57–73. https://doi.org/10.1007/s13740-012-0005-x

55. Guizzardi G. On Ontology, ontologies, Conceptualizations, Modeling Languages, and (Meta)Models. Proceedings of the 2007 Conference on Databases and Information Systems IV: Selected Papers from the Seventh International Baltic Conference DB&IS'2006 (NLD, 2007). NLD: IOS Press, 2007, p. 18–39.

56. Henderson-Sellers B. Bridging metamodels and ontologies in software engineering. Journal of Systems and Software, 2011. Issue 84, No 2, p. 301–313. https://doi.org/10.1016/j.jss.2010.10.025

57. Hogan A., Blomqvist E., Cochez M., et al. Knowledge Graphs. ACM Computing Surveys, 2021. Issue 54, No 4, p. 1–37. https://doi.org/10.1145/3447772

58. Hong X. J., Sik Yang H., Kim Y. H. Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application. 2018 International Conference on Information and Communication Technology Convergence (ICTC). (Jeju, 10.2018). Jeju: IEEE, 2018, p. 257–259. https://doi.org/10.1109/ICTC.2018.8539409

59. Horizontal and Vertical Scaling | System Design – GeeksforGeeks — geeksforgeeks.org. URL: https://www.geeksforgeeks.org/system-design-horizontal-and-vertical-scaling/

60. Horizontal Vs. Vertical Scaling: Which Should You Choose? (2024) URL: https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling/

61. Horrocks I., Patel-Schneider P. F., Boley H., et al. SWRL: A semantic web rule language combining OWL and RuleML. W3C Member submission. 2003. Vol. 21, Issue 79, p. 1–31.

62. Hussain J., Ul Hassan A., Muhammad Bilal H. S., et al. Model-based adaptive user interface based on context and user experience evaluation. Journal on Multimodal User Interfaces. 2018. Issue 12, No 1, p. 1–16. https://doi.org/10.1007/s12193-018-0258-2

63. Iftikhar M. U., Ramachandran G. S., Bollansee P., et al. DeltaIoT: A Self-Adaptive Internet of Things Exemplar. 2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). (Buenos Aires, Argentina, 05.2017). Buenos Aires, Argentina: IEEE, 2017, p. 76–82. https://doi.org/10.1109/SEAMS.2017.21

64. Ingebrigtsen E. SignalR: real-time application development : utilize real-time functionality in your .NET applications with ease. Birminham: Packt Pub., 2013, 222 p.

65. Jain R., Suman U. A Systematic Literature Review on Global Software Development Life Cycle. ACM SIGSOFT Software Engineering Notes, 2015. Issue 40, No 2, p. 1–14. https://doi.org/10.1145/2735399.2735408

66. Johansson L., Dossot D. RabbitMQ Essentials – Second Edition. 2nd edition. Packt Publishing, 2020, 154 p.

67. John V., Liu X. A Survey of Distributed Message Broker Queues, 2017. https://doi.org/10.48550/arXiv.1704.00411

68. Kaddoum E., Gleizes M.-P., Georgé J.-P., et al. Characterizing and Evaluating Problem Solving Self-* Systems. 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns (COMPUTATIONWORLD). (Athens, Greece, 11.2009). Athens, Greece: IEEE, 2009, p. 137–145. https://doi.org/10.1109/ComputationWorld.2009.100

69. Kaddoum E., Raibulet C., Georgé J.-P., et al. Criteria for the evaluation of self-* systems. Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing SystemsICSE ’10: 32nd International Conference on

Software Engineering. (Cape Town South Africa, 03.05.2010). Cape Town South Africa: ACM, 2010, p. 29–38. https://doi.org/10.1145/1808984.1808988

70. Kotis K. I., Vouros G. A., Spiliotopoulos D. Ontology engineering methodologies for the evolution of living and reused ontologies: status, trends, findings and recommendations. The Knowledge Engineering Review, 2020. Issue 35, article ID e4. https://doi.org/10.1017/S0269888920000065

71. Krupitzer C., Roth F. M., VanSyckel S., et al. A survey on engineering approaches for self-adaptive systems. Pervasive and Mobile Computing, 2015. Issue 17, p. 184–206. https://doi.org/10.1016/j.pmcj.2014.09.009

72. Lamy J.-B. Owlready: Ontology-oriented programming in Python with automatic classification and high level constructs for biomedical ontologies. Artificial Intelligence in Medicine, 2017. Issue 80,07.2017, p. 11–28. https://doi.org/10.1016/j.artmed.2017.07.002

73. Lemos R. de, Giese H., Müller H. A., et al. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. Software Engineering for Self-Adaptive Systems II. Rogério de Lemos (Ed.) et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 1–32. https://doi.org/10.1007/978-3-642-35813-5_1

74. Lilis G., Kayal M. A secure and distributed message oriented middleware for smart building applications. Automation in Construction, 2018. Issue 86, p. 163–175. https://doi.org/10.1016/j.autcon.2017.10.030

75. Lopatto I., Hovorushchenko T., Popov P. and Pavlova O., "Intelligent Multi-Agent System for Improving the Quality of Software by Taking into Account the Information of the Subject Area at All Stages of its Development," 2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), Cracow, Poland, 2021, p. 548-551. https://doi.org/10.1109/IDAACS53288.2021.9660866

76. Lourdusamy R., John A. A review on metrics for ontology evaluation. 2nd International Conference on Inventive Systems and Control (ICISC). (Coimbatore, 01.2018). Coimbatore: IEEE, 2018, p. 1415–1421. https://doi.org/10.1109/ICISC.2018.8399041

77. Lutsyk I., Fedasyuk D. Analysis of approaches to design ontological models of an adaptive software system. Computer systems and information technologies, 2024. No. 3, p. 13–20. https://doi.org/10.31891/csit-2024-3-2

78. Lutsyk I., Fedasyuk D. Usage of the message broker technology in the adaptive software systems. International Conference on Information and Communication Technologies in Education, Research, and Industrial Applications ICTERI-2024 (Lviv, Ukraine, September 23–27, 2024). ("Communications in Computer and Information Science" Series). URL: https://easychair.org/smart-program/ICTERI-2024/

79. Lytvyn V., Burov Y., Vysotska V., et al. Knowledge Novelty Assessment During the Automatic Development of Ontologies. 2020 IEEE Third International Conference on Data Stream Mining & Processing (DSMP). (Lviv, Ukraine, 08.2020). Lviv, Ukraine: IEEE, 2020, p. 372–377. https://doi.org/10.1109/DSMP47368.2020.9204124

80. Macías-Escrivá F. D., Haber R., Toro R. del et al. Self-adaptive systems: A survey of current approaches, research challenges and applications. Expert Systems with Applications, 2013. Issue 40, No 18, p. 7267–7279. https://doi.org/10.1016/j.eswa.2013.07.033

81. Magnanini F., Ferretti L., Colajanni M. Scalable, Confidential and Survivable Software Updates. IEEE Transactions on Parallel and Distributed Systems, 2022. Vol. 33, Issue 1, p. 176–191. https://doi.org/10.1109/TPDS.2021.3090330

82. Mikailov M., Petrick N., Azarbaijani Y., et al. Scaling and Parallelization of Big Data Analysis on HPC and Cloud Systems. 2019 International Conference on Advances in Computing and Communication Engineering (ICACCE) 2019, p. 1–8. https://doi.org/10.1109/TAI.2023.3246032

83. Mishra S. K., Sahoo B., Parida P. P. Load balancing in cloud computing: A big picture. Journal of King Saud University – Computer and Information Sciences, 2020. Issue 32, No 2, p. 149–158. https://doi.org/10.1016/j.jksuci.2018.01.003

84. Moreira R., Blair G., Carrapatoso E. FORMAware: Framework of Reflective Components for Managing Architecture Adaptation. Software Engineering and Middleware. ed. A. Coen-Porisini., A. Van Der Hoek., G. Goos., J. Hartmanis., J. Van Leeuwen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, p. 115–129. https://doi.org/10.1007/3-540-38093-0_8

85. Narula G. S., Yadav U., Duhan N., et al. Evolution of FOAF and SIOC in Semantic Web: A Survey. Big Data Analytics. ed. V. B. Aggarwal., V. Bhatnagar., D. K. Mishra. Singapore: Springer Singapore, 2018, p. 253–263. https://doi.org/10.1007/978-981-10-6620-7_25

86. Nguyen C. N., Lee J., Hwang S., et al. On the role of message broker middleware for many-task computing on a big-data platform. Cluster Computing, 2018. Issue 22, No S1, p. 2527–2540. https://doi.org/10.1007/s10586-018-2634-9

87. Ntafalias A., Tsakanikas S., Skarvelis-Kazakos S., et al. Design and Implementation of an Interoperable Architecture for Integrating Building Legacy Systems into Scalable Energy Management Systems. Smart Cities, 2022. Issue 5, No 4, p. 1421–1440. https://doi.org/10.3390/smartcities5040073

88. Pan T., Yu N., Jia C., et al. Sailfish: accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. Proceedings of the 2021 ACM SIGCOMM 2021 Conference(New York, NY, USA, 2021). New York, NY, USA: Association for Computing Machinery, 2021, p. 194–206. https://doi.org/10.1145/3452296.3472889

89. Peçanha C., Duarte B., Souza V. RASO: an Ontology on Requirements for the Development of Adaptive Systems. Anais do WER18 – Workshop em Engenharia de RequisitosWorkshop em Engenharia de Requisitos 2018. PUC-Rio, 2018. https://doi.org/10.17771/PUCRio.wer.inf2018-23

90. Perez-Palacin D., Mirandola R., Merseguer J. On the relationships between QoS and software adaptability at the architectural level. Journal of Systems and Software, 2014. Issue 87, p. 1–17. https://doi.org/10.1016/j.jss.2013.07.053

91. Preis S. J. Towards hybrid self-learning ontologies: A new Python module for closed-loop integration of decision trees and OWL. Journal of AI,

Robotics & Workplace Automation. 01.03.2024, article ID AIRWA-vol3-iss2-pg124. https://doi.org/10.69554/AIML1271

92. Priya N., Shanmuga Priya S. Load Balancing Algorithms in Cloud Computing Environment – An Effective Survey. Congress on Intelligent Systems. Mukesh Saraswat(Ed.) et al. Singapore: Springer Nature Singapore, 2022, p. 279–290. https://doi.org/10.1007/978-981-16-9416-5_20

93. Raibulet C., Arcelli Fontana F., Capilla R., et al. An Overview on Quality Evaluation of Self-Adaptive Systems. Managing Trade-Offs in Adaptable Software Architectures. Elsevier, 2017, p. 325–352. https://doi.org/10.1016/B978-0-12-802855-1.00013-7

94. Raibulet C., Masciadri L. Evaluation of dynamic adaptivity through metrics: an achievable target? 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture3rd European Conference on Software Architecture (ECSA). (Cambridge, United Kingdom, 09.2009). Cambridge, UK: IEEE, 2009, p. 341–344. https://doi.org/10.1109/WICSA.2009.5290667

95. Rastgoo V., Hosseini M., Kheirkhah E., et al. Semantic Web-Based Software Engineering by Automated Requirements Ontology Generation in SOA. International Journal of Web & Semantic Technology, 2014. Vol. 5, p. 1–11. https://doi.org/10.5121/ijwest.2014.5201

96. Reda R., Carbonaro A., De Boer V., et al. Supporting Smart Home Scenarios Using OWL and SWRL Rules. Sensors, 2022. Issue 22, No 11, p. 4131. https://doi.org/10.3390/s22114131

97. Reinecke P., Wolter K., Van Moorsel A. Evaluating the adaptivity of computing systems. Performance Evaluation, 2010. Issue 67, No 8, p. 676–693. https://doi.org/10.1016/j.peva.2009.12.001

98. Rosa N. S., Cavalcanti D. J. M. A Control-Theoretical Approach to Adapt Message Brokers. Advanced Information Networking and Applications. Ed. Leonard Barolli. Cham: Springer International Publishing, 2023, p. 261–273. https://doi.org/10.1007/978-3-031-29056-5_24

99. Ruiz F., Hilera J. R. Using Ontologies in Software Engineering and Technology. Ontologies for Software Engineering and Software Technology. ред. Coral Calero, Francisco Ruiz, Mario Piattini. Springer Berlin Heidelberg, 2006, p. 49–102. https://doi.org/10.1007/3-540-34518-3_2

100. Saadi A., Hammal Y., Oussalah M. C. Automata-Based Approach to Manage Self-Adaptive Component-Based Architectures: International Journal of Software Innovation. Issue 10, No 1, p. 1–22. https://doi.org/10.4018/IJSI.297623

101. Sabatucci L., Seidita V., Cossentino M. The Four Types of Self-adaptive Systems: A Metamodel. Intelligent Interactive Multimedia Systems and Services 2017. ed. G. De Pietro., L. Gallo., R. J. Howlett., et al. Cham: Springer International Publishing, 2018, p. 440–450. https://doi.org/10.1007/978-3-319-59480-4_44

102. Salatino A. A., Thanapalasingam T., Mannocci A., et al. The Computer Science Ontology: A Large-Scale Taxonomy of Research Areas. The Semantic Web – ISWC 2018. Denny Vrandečić (Ed.) et al. Cham: Springer International Publishing, 2018, p. 187–205. https://doi.org/10.1007/978-3-030-00668-6_12

103. Schölkopf L., Wolf M.-M., Hutmann V., et al. Conception, Development and First Evaluation of a Context-Adaptive User Interface for Commercial Vehicles. 13th International Conference on Automotive User Interfaces and Interactive Vehicular ApplicationsAutomotiveUI ’21 (Leeds United Kingdom, 09.09.2021). Leeds United Kingdom: ACM, 2021, p. 21–25. https://doi.org/10.1145/3473682.3480256

104. Shafiq D. A., Jhanjhi N. Z., Abdullah A. Load balancing techniques in cloud computing environment: A review. Journal of King Saud University – Computer and Information Sciences, 2021. Issue 34, No 7, p. 3910–3933. https://doi.org/10.1016/j.jksuci.2021.02.007

105. Sharma N., Agarwal R. HTTP, WebSocket, and SignalR: A Comparison of Real-Time Online Communication Protocols. Mining Intelligence and Knowledge Exploration. Ed. Seifedine Kadry, Rajendra Prasath. Cham: Springer Nature Switzerland, 2023, p. 128–135. https://doi.org/10.1007/978-3-031-44084-7_13

106. Szabo C., Sims B., Mcatee T., et al. Self-Adaptive Software Systems in Contested and Resource-Constrained Environments: Overview and Challenges. IEEE Access. Vol. 9, 2021, p. 10711–10728. https://doi.org/10.1109/ACCESS.2020.3043440

107. Tartir S., Arpinar I., Moore M., et al. OntoQA: Metric-Based Ontology Quality Analysis. Proceedings of the IEEE ICDM 2005 Workshop on Knowledge Acquisition from Distributed, Autonomous, Semantically Heterogeneous Data and Knowledge Sources. (Houston, Texas, 2005). Houston, Texas, 2005. URL: https://corescholar.libraries.wright.edu/knoesis/660

108. Thaddeus S., Raja S. V. K. Ontology-driven Model for Knowledge-Based Software Engineering. International Conference on Software Engineering and Knowledge Engineering (2006). URL: https://api.semanticscholar.org/CorpusID:27215197

109. Tommasini R., Sedira Y. A., Dell'Aglio D., et al. VoCaLS: Vocabulary and Catalog of Linked Streams. The Semantic Web – ISWC 2018. Ed. Denny Vrandečić, Kalina Bontcheva (Eds.) Cham: Springer International Publishing, 2018, p. 256–272. https://doi.org/10.1007/978-3-030-00668-6_16

110. Torres A., Galante R., Pimenta M. S., et al. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. Information and Software Technology, 2017. Issue 82, 02.2017, p. 1–18. https://doi.org/10.1016/j.infsof.2016.09.009

111. Troelsen A., Japikse P., Troelsen A., Japikse P. Type Reflection, Late Binding, Attribute, and Dynamic Types. Pro C# 10 with .NET 6. Berkeley, CA: Apress, 2022, p. 661–712. https://doi.org/10.1007/978-1-4842-7869-7_17

112. Tyshchenko A., Onishchenko K., Pysarenko K. The web-interface availability model for people with disabilities. Herald of Advanced Information Technology, 2019. Vol. 3, p. 206–2014. https://doi.org/10.15276/hait.03.2019.4

113. Villegas Machado N. M., Müller H. A., Tamura Morimitsu G. On Designing Self-Adaptive Software Systems. Sistemas y Telemática, 2011. Vol. 9, Issue 18, p. 29. https://doi.org/10.18046/syt.v9i18.1076

114. Vyakaranal S. B., Naragund J. G. Weighted Round-Robin Load Balancing Algorithm for Software-Defined Network. Emerging Research in Electronics, Computer Science and Technology. Eds. V. Sridhar, M.C. Padma, K.A. Radhakrishna Rao. Singapore: Springer Singapore, 2019, p. 375–387. https://doi.org/10.1007/978-981-13-5802-9_35

115. Wang L., Fang W., Du Y. Load Balancing Strategies in Heterogeneous Environments. Journal of Computer Technology and Applied Mathematics, 2024. Vol. 1, No. 2, p. 10-18. https://doi.org/10.5281/zenodo.12599358

116. Weyns D. An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective. 1. Wiley, 2021. https://doi.org/10.1002/9781119574910

117. Wilson R. S. I., Goonetillake J. S., Ginige A., et al. Ontology Quality Evaluation Methodology. In Computational Science and Its Applications – ICCSA 2022: 22nd International Conference, Malaga, Spain, July 4–7, 2022, Proceedings, Part I. Springer-Verlag, Berlin, Heidelberg, 2022, p. 509–528. https://doi.org/10.1007/978-3-031-10522-7_35

118. Wilson R. S. I., Goonetillake J. S., Indika W. A., et al. Analysis of Ontology Quality Dimensions, Criteria and Metrics. In Computational Science and Its Applications – ICCSA 2021: 21st International Conference, Cagliari, Italy, September 13–16, 2021, Proceedings, Part III. Springer-Verlag, Berlin, Heidelberg, 2021, p. 320–337. https://doi.org/10.1007/978-3-030-86970-0_23

119. Yang J., Yue Y., Vinayak R. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21) (04.2021). USENIX Association, 2021, p. 503–518. URL: https://www.usenix.org/system/files/nsdi21-yang.pdf

120. Yang Q., Tao X., Xie H., et al. FuAET: a tool for developing fuzzy self-adaptive software systems. Proceedings of the 6th Asia-Pacific Symposium on Internetware (Hong Kong China, 17.11.2014). Hong Kong China: ACM, 2014, p. 54–63. https://doi.org/10.1145/2677832.2677840

121. Ying S., Liang Z. Q., Wang J. L., et al. A Reflection Mechanism for Reusing Software Architecture. 2006 Sixth International Conference on Quality Software (QSIC'06). Proceedings of the 6th International Conference on Quality Software. (Beijing, 10.2006). Beijing: IEEE, 2006. https://doi.org/10.1109/QSIC.2006.5

122. Zaid L. A., Kleinermann F., De Troyer O. Applying semantic web technology to feature modeling. Proceedings of the 2009 ACM Symposium on Applied Computing (SAC) (Honolulu, Hawaii, USA, March 9-12 2009). Honolulu Hawaii: ACM, 2009, p. 1252–1256. https://doi.org/10.1145/1529282.1529563

123. Zhu H., Liu D., Bayley I., et al. Quality model and metrics of ontology for semantic descriptions of web services. Tsinghua Science and Technology, 2017. Vol. 22, Issue 3, p. 254–272. https://doi.org/10.23919/TST.2017.7914198

ДОДАТКИ

Додаток А. Програмний код модуля адаптації інтерфейсу користувача та функціональності програмної системи

Базовий клас для функціональної компоненти:

```
namespace AdaptationLibrary.Interfaces
{
    /// <summary>
    /// Standart inerface to get
information about adaptive plugins
    /// </summary>
    public abstract class IPlugin
    {
        #region Plugin Properties
        [Required]
        public string Name { get; set;
}

        [Required]
        public string Version { get;
set; }

        [Required]
        public string Location { get;
set; }

        public PluginType PluginType {
get; set; }

        public string Icon { get; set;
}
        #endregion

        #region Methods

        public IPlugin()
        {
            Name = "Test plugin";
            Version = "v1.0.0";
            Location = "NotInstalled";
            Icon = string.Empty;
            PluginType =
PluginType.DLL;
        }

        public IPlugin(string name,
string version, string location,
string icon, PluginType pluginType)
        {
            Name = name;
            Version = version;
            Location = location;
            Icon = icon;
            PluginType = pluginType;
        }

        /// <summary>
        /// Load additional
module/plugin based on context given
        /// </summary>
        /// <param
name="context">Parameter represent
main context to launch the plugin,
i.e. Page</param>
        public abstract Task
Load(object? context = null);

        /// <summary>
        /// Get information about
plugin icon based on given location
        /// </summary>
        /// <returns>FileInfo of given
plugin</returns>
        public abstract FileInfo
GetIcon();

        /// <summary>
        /// Load additional
module/plugin based on resource
location and optionaly given context
        /// </summary>
        /// <param
name="resourceLocation">Location of
additional resources of plugin</param>
        /// <param
name="context">Parameter represent
main context to launch the plugin,
i.e. Page</param>
        public abstract Task
LoadWithResources(string
resourceLocation, object? context =
null);

        /// <summary>
        /// Restore previously saved
state of plugin
        /// </summary>
        public abstract void
RestoreSavedState();

        /// <summary>
        /// Save current state of
plugin
        /// </summary>
        public abstract void
SaveState();
        #endregion

    }
}
```

Допоміжний клас з методами рефлексії, для завантаження та

реєстрації функціональних компонент:

```
namespace AdaptationLibrary.Helpers
{
    public static class
ReflectionHelper
    {
        public static
IEnumerable<IPlugin>
GetPluginsFromFile(string fileName)
        {
            List<IPlugin> plugins =
new List<IPlugin>();
            var DLL =
Assembly.LoadFile(fileName);
            foreach (Type type in
DLL.ExportedTypes.Where(t => t != null
&& t.BaseType == typeof(IPlugin)))
            {
                var plugin =
Activator.CreateInstance(type) as
IPlugin;

                if (plugin is not
null)
                {

plugins.Add(plugin);
                }
            }

            return plugins;
        }

        public static
IEnumerable<IPlugin>
GetPluginsFromFolder(string
folderPath)
        {
            List<IPlugin> plugins =
new List<IPlugin>();

            var DLLs =
Directory.GetFiles(folderPath,
"*.dll");

            foreach (string fileName
in DLLs)
            {

plugins.AddRange(GetPluginsFromFile(fi
leName));
            }

            return plugins;
        }
    }
}
```

Базовий клас для графічної компоненти:

```
namespace AdaptationLibrary.Interfaces
{
    public enum UIElementType
    {
        Page = 0,
        Style = 1,
        Strings = 2,
        Font = 3,
        SVG = 4,
        None = 5,
    }
    public abstract class
IUserInterfaceElement
    {
        #region Element Properties

        public string Name { get; set;
}

        public string
UIElementLocation { get; set; }

        public UIElementType
UIElementType { get; set; }
        #endregion
        #region Element Methods
        public IUserInterfaceElement()
        {
            Name = "Test Element";
            UIElementLocation =
string.Empty;
            UIElementType =
UIElementType.None;
        }

        public
IUserInterfaceElement(string name,
string uiElementLocation,
UIElementType uIElementType)
        {
            Name = name;
            UIElementLocation =
uiElementLocation;
            UIElementType =
uIElementType;
        }

        /// <summary>
        /// Load additional resource
and/or UI elements with default
resource location
        /// </summary>
        public abstract Task
LoadElementWithDefaultResources();

        /// <summary>
        ///  Load additional resource
and/or UI elements with custom
resource locations
        /// </summary>
```

```
        /// <param
name="elementResources">Custom
paths</param>
        public abstract Task
LoadElementWithResources(string[]
elementResources);
        #endregion
    }
}
```

Додаток Б. Програмний код модуля визначення динамічної конфігурації програмної системи

```
from flask import Blueprint, Response,
request, current_app, jsonify
import os
from
api.adaptive_ontology.cognitive_ontolo
gy import *
from api.rabbitmq.publisher import
OntologySyncSender
from api.rabbitmq.receiver import
OntologySyncReceiver
from threading import Thread

ontology_sync_publisher =
OntologySyncSender()
thread =
Thread(target=OntologySyncReceiver)
thread.start()

ontology_resource =
Blueprint('ontology', __name__)
filepath = os.path.join('./storage',
'Adaptive_system.owl')

if not os.path.exists(filepath):
    ontology = create_ontology()
    print("From ontology folder")
else:
    ontology =
create_ontology(filepath)
    print("From storage folder")

@ontology_resource.route('/ontology/ad
d_user', methods = ['POST'])
def add_user_to_ontology():
    current_user = request.get_json()
    properties_list = ['UUID',
'FullName', 'Role']
    data_list = [current_user["uuid"],
current_user["fullname"],
current_user["role"]]

    create_ontology_entity(ontology,
current_user["role"]+current_user["uui
d"], 'User', properties_list,
data_list)

    if not
os.path.exists(current_app.config['FIL
E_STORAGE']):

os.mkdir(current_app.config['FILE_STOR
AGE'])
    save_ontology(ontology, filepath)

    return jsonify({'message': 'User
added'})

@ontology_resource.route('/ontology/sy
nc_config/<user_uuid>', methods =
['PUT'])
def
get_user_prefered_config_sync(user_uui
d):
    data = request.get_json()
    requirements_uuid =
data['Requirements']
    configs = {}
    try:
        ontology_user = next(i for i
in ontology['User'].instances() if
user_uuid in i.UUID)

        if requirements_uuid:
            if
ontology_user.__getattr__('need_config
uration'):

ontology_user.__setattr__("need_config
uration", None)

            if
ontology_user.__getattr__('has_require
ment'):

ontology_user.__setattr__("has_require
ment", [])

            requirements =
[requirement for requirement in
ontology['Requirement'].instances() if
requirement.Req_UUID in
requirements_uuid]
            for requirement in
requirements:

create_relationship(ontology_user,
requirement, 'has_requirement')

            save_ontology(ontology,
filepath)

        with ontology:

sync_reasoner_pellet(debug=True,
infer_property_values = True,
infer_data_property_values = True)

        assembly_config = {}
        assembly =
ontology_user.__getattr__('need_config
uration')
        for prop in
```

```
list(assembly.get_properties()):
            if is_data_property(prop):

assembly_config[prop.get_python_name()
] =
assembly.__getattr__(prop.get_python_n
ame())[0]

        configs['Assembly'] =
assembly_config

        modules = {}
        graphics = {}

        for use in
ontology_user.__getattr__('need_config
uration').contains:
            module_data = {}
            for prop in
list(use.get_properties()):

module_data[prop.get_python_name()] =
use.__getattr__(prop.get_python_name()
)[0]

            if isinstance(use,
ontology.Functional_Component) or
issubclass(type(use),
ontology.Functional_Component):

modules[use.__class__.__name__] =
module_data

            if isinstance(use,
ontology.Graphical_Component) or
issubclass(type(use),
ontology.Graphical_Component):

graphics[use.__class__.__name__] =
module_data

        configs['Modules'] = modules
        configs['Graphics'] = graphics

    except:
        print("Unexpected error:",
sys.exc_info()[0])
        raise ValueError('Cannot
process user data and sync ontology')

    return jsonify(configs)

@ontology_resource.route('/ontology/br
oker_config/<user_uuid>', methods =
['PUT'])
def
get_user_preferred_config_broker(user_
uuid):
    data = request.get_json()
    message = {
        "uuid": user_uuid,
        "Requirements":
data['Requirements']
    }


ontology_sync_publisher.sendMessage(me
ssage)

    return Response(status=200)
```

Додаток В. Програмний код модуля налаштування брокера повідомлень для адаптивної програмної системи

Сервіс для запуску опрацювання даних отриманих з брокера повідомлень:

```
namespace AdaptiveAppAPI.Services
{
    public class AdaptiveSettingsRecieverService: BackgroundService
    {
        private readonly IMessageConsumer _messageConsumer;

        public AdaptiveSettingsRecieverService(IMessageConsumer messageConsumer)
        {
            _messageConsumer = messageConsumer;
        }

        protected override Task ExecuteAsync(CancellationToken stoppingToken)
        {
            _messageConsumer.ReadMessage();
            return Task.CompletedTask;
        }
    }
}
```

Клас для опрацювання повідомлень та подій про адаптацію програмного забезпечення:

```
namespace AdaptiveAppAPI.Helpers
{
    public class
AdaptationMessageConsumer:
IMessageConsumer
    {
        private const int
MaxNumberOfRetries = 5;
        private readonly
ILogger<AdaptationMessageConsumer>
_logger;
        private ConnectionFactory?
_connectionFactory;
        private IConnection? _connection;
        private IModel? _channel;
        private readonly
IHubContext<AdaptiveSettingsSignalRHub>
_notificationHub;

        private void CreateConnection()
        {
            try
            {
                _connectionFactory = new
ConnectionFactory() { Uri = new Uri("")
};
                _connection =
_connectionFactory.CreateConnection();
                _channel =
_connection.CreateModel();
            }
            catch (Exception ex)
            {
                _logger.LogError($"Cant
create connection {ex.Message}");
            }
        }
        public
AdaptationMessageConsumer(ILogger<Adaptat
ionMessageConsumer> logger)
        {
            _logger = logger;
            CreateConnection();
        }

        public void ReadMessage()
        {
            if (_connection is null)
            {
                return;
            }

_channel?.ExchangeDeclare(RabbitMQConstan
ts.WorkExchange,
RabbitMQConstants.ExchangeType);
            _channel?.QueueDeclare(
                queue:
RabbitMQConstants.WorkQueue,
                durable: true,
                exclusive: false,
```

```
                autoDelete: false,
                arguments: null);

_channel?.QueueBind(RabbitMQConstants.Wor
kQueue, RabbitMQConstants.WorkExchange,
string.Empty, null);

            var queueArgs = new
Dictionary<string, object>
            {
                {
RabbitMQConstants.DeadLetterExchangeHeade
r, RabbitMQConstants.WorkExchange },
                {
RabbitMQConstants.TTLHeader, 60_000 },
            };


_channel?.ExchangeDeclare(RabbitMQConstan
ts.RetryExchange,
RabbitMQConstants.ExchangeType);

_channel?.QueueDeclare(RabbitMQConstants.
RetryQueue, true, false, false,
queueArgs);

_channel?.QueueBind(RabbitMQConstants.Ret
ryQueue, RabbitMQConstants.RetryExchange,
string.Empty, null);
            var consumer = new
EventingBasicConsumer(_channel);
            consumer.Received += async
(model, ea) =>
            {
                byte[] body =
ea.Body.ToArray();
                string message =
Encoding.UTF8.GetString(body);

                var userInfo =
JsonConvert.DeserializeObject<SystemConfi
gurationDto>(message);

                bool isFailure = true;

                if
(ea.BasicProperties.Headers is not null
&&
ea.BasicProperties.Headers.ContainsKey("u
serId"))
                {

ea.BasicProperties.Headers.TryGetValue("u
serId", out object userId);

ea.BasicProperties.Headers.TryGetValue("r
oom", out object room);

                    if (userId != null &&
userInfo != null)
                    {
                        var content = new
Message(userId.ToString(), userInfo);
                        await
_notificationHub.Clients.Group(room !=
null ? room.ToString():
"all").SendAsync("ReceiveMessage",
message);
                    }
                }
                if (isFailure)
                {

_logger.LogError($"Cant send
configurations");
                    long count = 0;
                    if
(ea.BasicProperties.Headers is not null
&&
ea.BasicProperties.Headers.ContainsKey("x
-death"))
                    {
                        var
deathProperties =
(List<object>)ea.BasicProperties.Headers[
"x-death"];
                        var lastRetry =
(Dictionary<string,
object>)deathProperties[0];
                        count =
(long)lastRetry["count"];
                    }

                    if (count <
MaxNumberOfRetries)
                    {

_channel.BasicPublish(RabbitMQConstants.R
etryExchange, string.Empty,
ea.BasicProperties, body);
                    }
                }


_channel?.BasicAck(ea.DeliveryTag,
false);
            };

            _channel?.BasicConsume(
                queue:
RabbitMQConstants.WorkQueue,
                autoAck: false,
                consumer: consumer);
        }

    }
}
```

Додаток Г. Акти про впровадження та дослідне випробування результатів дисертації

ЗАТВЕРДЖУЮ
Директор ПП "ЛІНК АП СТУДІО"
Андрій САМБІР
"21" жовтня 2024 р.

АКТ
про дослідне випробування результатів
дисертаційного дослідження аспіранта кафедри програмного забезпечення
Національного університету «Львівська політехніка»
Луцика Іллі Ігоровича

Даний акт складений про те, що метод динамічного визначення адаптивних налаштувань програмного забезпечення з використанням технологій брокера повідомлень, розроблений під час виконання дисертаційної роботи Луцика Іллі Ігоровича на тему «Методи та засоби створення адаптивних програмних систем на основі онтологій», пройшов дослідне випробування на ПП "ЛІНК АП СТУДІО" (м. Львів).

Запропонований метод дозволяє визначати та динамічно модифікувати налаштування програмного забезпечення з урахуванням даних про активний пристрій та вимоги до програмного забезпечення. Використання принципів та технологій брокера повідомлень забезпечує можливість розподілення запитів до вебсервісів, що, в свою чергу, дозволяє в подальшому масштабувати програмне забезпечення у залежності від мережевого трафіку. Отримані результати демонструють на 20 % кращі показники швидкості генерації налаштувань програмного забезпечення для запропонованого методу у порівнянні з класичним підходом визначення конфігурації системи.

Використання запропонованого методу дозволить інтегрувати нові функціональні компоненти динамічно, без необхідності повторного налаштування та розгортання програмного забезпечення.

Даний акт не є підставою для взаємних фінансових розрахунків.

ЗАТВЕРДЖУЮ
Проректор з науково-педагогічної роботи
Національного університету
«Львівська політехніка»
_____ Олег ДАВИДЧАК
"23" жовтня 2024 р.

АКТ
про впровадження результатів дисертаційної роботи
Луцика Іллі Ігоровича
«Методи та засоби створення адаптивних програмних систем на основі онтологій»,
представленої на здобуття наукового ступеня доктора філософії
за спеціальністю 121 "Інженерія програмного забезпечення"
у навчальному процесі кафедри програмного забезпечення

Даний акт складений комісією у складі:
д.т.н., проф. Федасюк Д. В. – завідувач кафедри програмного забезпечення, лектор дисципліни «Наукові дослідження та семінари за їх тематикою»;
д.т.н., проф. Журавчак Л. М. – професор кафедри програмного забезпечення;
к.т.н., доц. Сенів М. М. – доцент кафедри програмного забезпечення.

Цим актом підтверджується використання результатів дисертаційної роботи Луцика Іллі Ігоровича на тему «Методи та засоби створення адаптивних програмних систем на основі онтологій» в навчальному процесі кафедри програмного забезпечення для студентів спеціальності 121 «Інженерія програмного забезпечення» в лекційному курсі та практичних заняттях дисципліни освітньо-кваліфікаційного рівня магістр «Наукові дослідження та семінари за їх тематикою».

Розроблені у дисертації методи та представлені підходи до створення онтологічних моделей предметної області можуть бути застосовані у процесі проєктування та розробки складних програмних комплексів. Вивчення методу представлення об'єктів програмної системи у вигляді онтологічної моделі дозволить уніфікувати знання про предметну область та забезпечує можливість продовження процесів розроблення програмного забезпечення без повторного залучення експертів предметної області у разі необхідності його повторного розгортання.

Члени комісії:
зав. каф. програмного забезпечення,
д.т.н., професор _____ Дмитро ФЕДАСЮК

проф. каф. програмного забезпечення,
д.т.н., професор _____ Любов ЖУРАВЧАК

доц. каф. програмного забезпечення,
к.т.н., доцент _____ Максим СЕНІВ