УДК 681.3 http://doi.org/10.15407/pp2025.02.003

П.І. Андон, С.В. Суботін, П.І. Перконос, Є.М. Твердохліб

СУЧАСНИЙ СТАН ТА ПРОБЛЕМИ РОЗВИТКУ СЕРВІС-ОРІЄНТОВАНОЇ ПАРАДИГМИ

На основі аналізу розвитку програмної інженерії визначені основні принципи та джерела сервіс-орієнтованої парадигми. Розглянуті архітектурні особливості та основні шаблони сервіс-орієнтованих застосувань та стандарти, на яких вони ґрунтуються. Надана характеристика програмної інженерії розробки сервіс-орієнтованих застосувань та сучасного стану засобів її підтримки. Виявлені основні проблеми побудови сервіс-орієнтованих застосувань та запропоновані шляхи їх подолання.

Ключові слова: автоматизація наукових досліджень, інтегрована інформаційна інфраструктура, корпоративні інформаційні системи, сервіс-орієнтована архітектура, Semantic Web, семантичний веб-сервіс, онтологія.

Ph.I. Andon, S.V. Subotin, P.I. Perkonos, Ie.M. Tverdokhlib

CURRENT STATE AND EXTENTION PROBLEMS OF THE SERVICE-ORIENTED PARADIGM

Based on the analysis of the development of software engineering, the main principles and sources of the service-oriented paradigm are identified. The architectural features and basic templates of service-oriented applications and the standards on which they are based are considered. The software engineering of service-oriented application development and the current state of its support tools are described. The main problems of building service-oriented applications are identified and ways to overcome them are proposed.

Keywords: automation of scientific research, integrated information infrastructure, corporate information systems, service-oriented architecture, Semantic Web, semantic web service, ontology.

Вступ

В умовах всеосяжної інформатизації суспільства та насиченості його обчислювальними ресурсами, поєднаними відомчими та глобальними мережами, домінуючою парадигмою побудови та використання програмних засобів стає сервіс-орієнтована парадигма. Тому розуміння основних принципів та джерел становлення цієї парадигми, архітектурних особливостей, стандартів, що лежать в її основі, сучасного стану засобів програмної інженерії, котрі забезпечують створення та використання розподілених сервіс-орієнтованих застосувань, є запорукою ефективного створення та використання прикладних програмних засобів.

Джерела та еволюція сервіс-орієнтованої парадигми

Сервіс-орієнована парадигма сформувалась в результаті еволюції програмної інженерії відповідно до розвитку обчислювальної інфраструктури й потреб суспільства та успадкувала принципи й методи, відпрацьовані в попередніх парадигмах, що домінували в різні часи (рис.1).

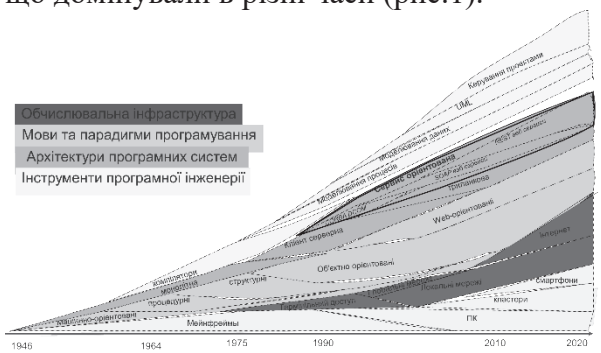

Рис. 1. Еволюція програмної інженерії

Один з основних принципів - принцип повторного використання коду набував різних форм у різних парадигмах по-

чинаючи з машинно-орієнтованого програмування. Поява універсальних мов програмування (Алгол, Фортран) дозволила повторно використовувати програми на комп'ютерах із різною системою команд, а також компонувати складні алгоритми з типових частин завдяки процедурній парадигмі програмування.

Складність процесів, що автоматизувались, швидко зростала, і логіка їх виконання, заснована на операторах прямих посилань, ставала неконтрольованою та важкою для супроводження. Для забезпечення прозорості алгоритмів були запропоновані типові принципи організації логіки виконання, засновані на прозорих конструкціях послідовного, умовного та циклічного виконання, відомі як структурне програмування . Мови програмування, засновані на цій парадигмі (Pl/1, Pascal, C), швидко витіснили застарілі Fortran та Algol. Структурна парадигма започаткувала модульний принцип побудови програм, згідно з якою складна система декомпозується на ряд модулів, які могли розроблятись незалежно різними виконавцями і потім збиратись у готову систему.

Але для цього потрібно було здійснювати попереднє проєктування системи, виходячи з вимог до її функціональності. Для забезпечення проєктування програмних систем виникла методологія моделювання різних аспектів систем за допомогою графічних нотацій SADT (structured analysis and design technique) [1].

Централізована обробка даних різними користувачами на одній «потужній» ЕОМ забезпечила взаємодію користувачів та використання напрацювань за рахунок їх розміщення в спільній файловій системі мейнфрейму та обміну даними й програмами на магнітних носіях.

Наступним кроком організації прикладних застосувань була розробка баз даних (БД) - пойменованих сукупностей даних, організованих за певними правилами, що передбачають загальні принципи опису, зберігання і маніпулювання даними, незалежно від прикладних програм. Перші БД, запропоновані для використання на мейнфреймах, використовували ієрархічну або мережеву (графову) модель взаємозв'язків між записами БД та відповідні мови опису та доступу до даних, швидко витіснених більш гнучкою реляційною моделлю даних, яка стала однією із складових методології SADT та відтворена в CASE середовищах. Зберігання даних у централізованому структурованому сховищі зі стандартизованими методами доступу до них спростило користувачам обмін даними та забезпечило поєднання програм, що автоматизують окремі операції технологічного ланцюга в єдину систему з автоматичною передачею даних між ними.

З появою мікропроцесорів обчислювальна інфраструктура підприємств почала стрімко змінюватись. Поряд із мейнфреймами з'явились міні та персональні комп'ютери, вартість придбання та утримання яких була значно меншою, а за продуктивністю вони практично не поступалися мейнфреймам. Тому підприємства швидко насичувались персональними комп'ютерами, і обробка даних почала виконуватись розподілено на багатьох робочих місцях, розосереджених по всьому підприємству. Однак водночас виникла потреба інформаційної взаємодії та спільного використання даних і програмних напрацювань, яка на мейнфреймах вирішувалася природним чином за рахунок централізованої обробки на одній ЕОМ і розміщення даних в її файловій системі та базах даних.

Ця проблема була швидко вирішена шляхом поєднання окремих розподілених на підприємстві комп'ютерів у мережу завдяки досвіду забезпечення доступу до мейнфреймів з терміналів, віддалених на багато сотень, а то й тисяч кілометрів через канали зв'язку. Локальні мережі забезпечили сумісне використання розподілених обчислювальних ресурсів підприємства і доступ до спільної файлової системи та баз даних. Монолітна архітектура прикладних систем, притаманна мейнфреймам, практично була витіснена дволанковою клієнт-серверною архітектурою, за якою функціонал структурованого зберігання та доступу до даних виконувався окремим застосуванням (СУБД) на окремому сервері. Їх обробка відбувалася на інших застосунках, які взаємодіяли з сервером за стандар-

тними мережевими протоколами та стандартною мовою запитів до даних.

Подальшим розвитком структурного програмування стала об'єктно-орієнтована парадигма (ООП), яка поєднала в одній конструкції (класі) структуру опису об'єкта автоматизації та його поведінку (процедуру обробки його даних) і розглядала прикладну систему як динамічну сукупність взаємодіючих об'єктів. Механізми інкапсуляції структури опису об'єктів та їхньої поведінки значно спростили логіку прикладної системи, а механізми наслідування та поліморфізму гнучкість повторного використання готових рішень. Тому в широко доступні структурні мови програмування додавалась підтримка об'єктно-орієнтованої парадигми (C->C++, Pascal->Object Pascal), створювались нові мови підтримки ООП (JAVA, Python).

Також, як і для методології структурного програмування, були запропоновані методології IDEF та відповідні CASE засоби, для моделювання систем в ООП була запропонована методологія UML [2].

Повторне використання реалізованих алгоритмів у новій прикладній системі в процедурній або в об'єктно-орієнтованій парадигмі потребувала їхньої компіляції та зв'язування із власним кодом у монолітну, виконувану в одному адресному просторі програму. Водночас розробник був обмежений у використанні готових рішень рамками платформи розробки своєї системи. Тому фірма Microsoft запропонувала технологію COM (Component object model) - виклику методів об'єктів, засновану на стандартизації опису інтерфейсу методів. Водночас код виконується паралельно на тій же ЕОМ в окремому адресному просторі як готовий компонент і не потребує компіляції та вбудовування в прикладну систему. За приклад використання цієї технології можна навести механізми OLE (Object linking and embeding) вбудовування в документи офісних систем частин, обробка яких здійснюється іншими застосуваннями. Наступним кроком розвитку COM технології стала специфікація DCOM (distributed COM), яка дозволяла здійснювати віддалений виклик процедур об'єктів, що створені та існують іншою програмою на іншій ЕОМ з передачею параметрів та отриманням результату по мережі. Нажаль, ці специфікації були реалізовані тільки в рамках операційної системи Microsoft Windows. Тому групою OMG була розроблена альтернативна специфікація CORBA цього механізму інтеграції ізольованих систем, розроблених різними мовами програмування, працюючих в різних вузлах мережі на різних платформах та взаємодіючих одна з одною так само просто, наче вони знаходились в адресному просторі одного процесу, що можна вважати початком сервіс-орієнтованої парадигми.

Перша глобальна мережа ARPANET створювалась як закрита мережа військового призначення під контролем міністерства оборони США. Розроблені в її рамках протоколи TCP/IP [3] адресації вузлів та передачі пакетів даних між ними були закритими та заборонені для використання у відкритих проєктах. Після закриття 1989 року проєкту ARPANET ця заборона була знята, що сприяло стрімкому нарощуванню поєднання вже існуючих локальних мереж в єдину глобальну мережу і використання служб передачі файлів та електронної пошти, створених в рамках проєкту ARPANET.

Для забезпечення сумісності апаратури та програмного забезпечення вузлів глобальної мережі 1994 року було створено консорціум W3C, покликаний розробляти єдині принципи та стандарти (рекомендації) Інтернету. На основі одного з перших стандартів – мови розмітки текстових документів із гіперпосиланнями HTML та протоколу прикладного рівня передачі даних у мережі інтернет http [4] були створені програми браузерів, що підтримують взаємодію користувача через мережу з програмою, що виконується на віддаленому сервері. Це здійснюється через відображення сформованого програмою документа мовою HTML та відправки підготовлених користувачем даних для виконання програми. Такі триланкові WEB застосування на сьогодні практично витіснили традиційні дволанкові клієнт-серверні застосування, оскільки не потребують розгортання на машині користувача,

доступні з будь-якого робочого місця, під'єднаного до Інтернету, в тому числі з мобільного смартфону, та можуть надаватись у користування як сервіс і не потребують від користувача підтримки та обслуговування.

Протокол HTTP не накладає ніяких обмежень на структуру та формат даних, що передаються, тому за його допомогою можна розповсюдити технологію CORBA та DCOM віддаленого виклику процедур та забезпечити взаємодію не тільки користувача з прикладними застосуваннями через браузер, а й між будь-якими застосуваннями, виконуваними на вузлах Internet мережі. Для цього консорціум W3C, враховуючи досвід використання HTML для кодування інтерфейсу користувача, адаптував його для представлення будь-яких структурованих даних XML (extended markup language) та на його основі визначив стандарт міжпрограмної взаємодії через інтернет SOAP (simple object access protocol) [5], який був значно спрощений порівнянно з CORBA. Підтримка SOAP була реалізована в різних платформах програмування та швидко витіснила CORBA для організації взаємодії прикладних застосувань як в рамках одного підприємства ESB (Enterprise Service Bus), так і між підприємствами (B2B).

Завдяки широкому розповсюдженню WEB-застосувань за технологією AJAX, що використовує взаємодією користувачів через браузер із вбудованим інтерпретатором мови JAVAscript, більш популярним для організації програмного інтерфейсу програм через інтернет став архітектурний стиль REST, запропонований Роєм Філдінгом 2000 року [6].

Цей підхід почав витісняти протокол SOAP завдяки тому, що він передбачає віддалений виклик процедур та отримання результату напряму за протоколом HTTP, підтримка якого вбудована в JAVAscript. А формат обміну не обмежується і може бути використаний прийнятий в JAVAscript формат JSON, більш компактний, ніж XML, і тому організація взаємодії здійснюється простіше та ефективніше.

Віддалений виклик процедур мережею на основі стандарту SOAP або REST технології для виконання певної обробки або отримання потрібних даних забезпечує використання готових програмних рішень без необхідності їхньої компіляції та зв'язування в єдиний модуль, скориставшись вже розгорнутим ПЗ на власній інфраструктурі або на інфраструктурі постачальника ПЗ як сервісу. Це в свою чергу дозволяє компонувати прикладні системи автоматизації бізнес-процесів на основі автономних програмних компонент – сервісів, кожний з яких виконує чітко визначену бізнес логіку у власному операційному оточенні, отримує запит на виконання та повертає результат, використовуючи стандартні протоколи та чітко визначений інтерфейс.

Така розподілена сервіс-орієнтована архітектура прикладних систем в умовах розвинутого Інтернету стала домінуючою та поступово витісняє традиційні локальні та клієнт-серверні застосування завдяки:

- скороченню витрат та термінів розробки потрібного програмного забезпечення за рахунок декомпозиції складної системи на більш керовані автономні компоненти (мікросервіси), використання готових сервісів, доступних в Інтернеті;
- забезпеченню використання програмних систем та окремих компонент без необхідності їхнього розміщення та обслуговування на власних технічних ресурсах;
- колективному використанню програмних систем та окремих її компонент;
- масштабуванню та модифікації системи відповідно до потреб шляхом додавання нових компонент або заміщення продуктивнішими.

Сервіс-орієнтовані системи суттєво відрізняються від традиційних десктоп та клієнт-серверних застосувань своєю розподіленою природою та асинхронним виконанням обробки даних на різних вузлах.

Тому технологія та засоби розробки програмного забезпечення на всіх етапах починають розвиватись з урахуванням цих особливостей та підтримувати подієорієнтовану парадигму програмування, яка

забезпечує синхронізацію одночасного виконання окремих процедур процесу.

Архітектурні особливості та шаблони сервіс-орієнтованої парадигми

Згідно із базовою моделлю [7] сервіс-орієнтована архітектура - це парадигма побудови прикладних систем на основі використання взаємодіючих розподілених програмних компонент, які виконуються в різних операційних оточеннях, що можуть належати і бути під контролем різних власників.

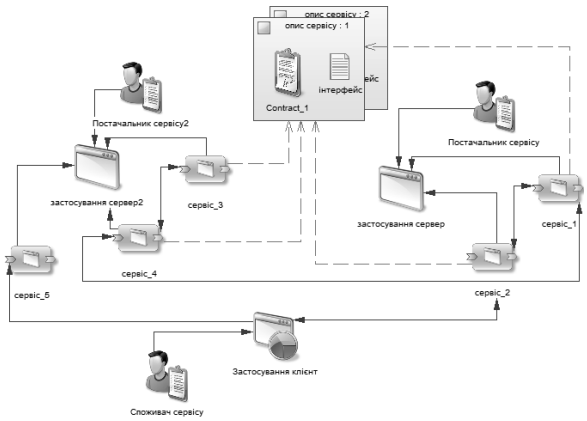

Рис. 2. СОА

У сервіс-орієнтованому застосуванні частина функціональності не реалізується, а її виконання делегується іншому застосуванню-серверу через надсилання повідомлення-запиту на виконання потрібної дії та отримання результату за узгодженим протоколом. Застосування-сервер забезпечує виконання чітко визначеної функціональності (сервісу) у відповідь на отримане повідомлення та відправку результату виконання в чітко визначеному форматі. Зазвичай, це отримання інформації, яку підтримує провайдер, або зміна стану об'єкта в зоні відповідальності провайдера. Водночас клієнт для виконання потрібної йому функціональності може використовувати різні сервіси від різних постачальників, організовуючи логіку свого процесу відповідно до отриманих результатів виконання. В свою чергу сервіс для виконання своєї функціональності також може використовувати інші сервіси, як свої, так і від інших постачальників.

Застосування-сервер розробляється, розгортається та підтримується постачальником сервісу (провайдером), який гарантує виконання сервісу за запитами потенційних користувачів за визначеним протоколом відповідно до умов постачання на ресурсах провайдера. Для того, щоб користувач зміг скористатися сервісом, постачальник має забезпечити наступні властивості та складові сервісу:

Виявлення сервісу потенційними користувачами повинно забезпечуватись постачальником через публікацію опису його сервісу на сайтах, загальнодоступних реєстрах, адресного інформування або іншим чином.

Зацікавленість передбачає, що сервіс виконує потрібну функціональність потенційним користувачам на прийнятних умовах, які викладаються в його описі.

Доступність передбачає, що сервіс забезпечує взаємодію через спільне з потенційними користувачами комунікаційне середовище (як правило, інтернет) за узгодженими протоколами (зазвичай, розповсюджені стандарти) а також адресою, які визначені в описі, та готовий виконувати запити на прийнятних умовах, які викладаються в описі (як правило, 24/7).

Інтерфейс визначає формат і структуру даних вхідного та вихідного повідомлень, на основі яких сервіс виконує визначену функціональність, а застосування-клієнт отримує результати виконання, на основі яких будує логіку своїх процесів. Для цього крім формату та структури даних, що передаються, обидві сторони повинні однаково інтерпретувати інформацію, якою обмінюються. Тобто використовувати однакову семантику (онтологію) даних, наданих у повідомленнях і яка має бути надана в описі сервісу.

Функціональність провайдер реалізує сервіс на власний розсуд, враховуючи потреби потенційних користувачів і надає відомості про результати виконання сервісу, не вдаючись у деталі «як саме він це робить». Користувач зі свого боку використовує сервіс як «чорну скриньку» для задоволення власних потреб, надаючи дані для виконання завдання та отримуючи потрібний результат через опублікований ін-

терфейс. Водночас сервіс може здійснювати додаткову обробку, яка не опубліковується і не є суттєвою для використання сервісу потенційним користувачем, але необхідна для підтримки стану об'єкта та роботи сервісу, якими він опікується. Для правильного використання сервісу крім структури та семантики повідомлень клієнт має бути обізнаний про дії, які виконує сервіс та їх вплив на стан спільних об'єктів споживача та постачальника.

Умови використання публікуються в описі сервісу та визначають підстави надання доступу (комерційна основа, виробничі взаємовідносини, вільний доступ).

QoS визначає якість надання послуг (швидкість відклику, обсяги, надійність, вартість тощо), які повинні додаватися до опису сервісу оптимального вибору потрібного користувачу сервісу серед наявних пропозицій.

Базова модель СОА визначає загальні принципи побудови сервіс-орієнтованих систем, конкретні реалізації котрих можуть відрізнятися стандартами взаємодії із сервісами, які використовуються, топологією взаємозв'язків, типами інтерфейсів тощо.

Зокрема, розрізняють дві топологічні архітектурні моделі композиції сервісів для реалізації бізнес-процесів:

Оркестровка використовує центральний керуючий сервіс, який, подібно до диригента в оркестрі, координує виконання окремих сервісів, що реалізують частину складного процесу, викликаючи їх у потрібній черзі. Центральний процес організує логіку виконання складного процесу, контролюючи результати виконання окремих кроків та формуючи послідовність викликів потрібних сервісів. Остання дія залежить від результатів виконання попередніх для забезпечення виконання композитного сервісу.

Хореографія не має централізованого контролю над процесом подібно до команди танцюристів, кожний з яких знає своє завдання та взаємодіє з партнерами на основі визначених правил.

Складність процесів, які реалізуються прикладним застосуванням в монолітній архітектурі обмежено можливостями комп'ютера, на якому воно виконується і, як правило, не виходить за межі процесів одного або декількох підрозділів підприємства. Тому для забезпечення складних процесів у межах всього підприємства потрібно інтегрувати окремі монолітні застосування, що автоматизують окремі ланки таких процесів, з виконанням їх у відповідності з встановленими бізнес-правилами. Також має забезпечуватися автоматична передача даних між ними на засадах розподіленої сервіс-орієнтованої архітектури.

Для інтеграції окремих компонент у межах однієї організації застосовуються різні методи (архітектурні шаблони) та відповідні інструменти. Одним із перших таких шаблонів можна визначити корпоративну сервісну шину (ESB), яка підтримує обмін даних між різнорідними застосуваннями в режимі реального часу. Організації можуть підключати застосування до централізованої шини, досягаючи мінімальних витрат на реалізацію інтерфейсу, використовуючи протоколи та компоненти платформи-застосування. ESB у свою чергу пропонує централізовану платформу, яка стандартизує та надає сервіси для перетворення різних форматів даних, протоколів передачі даних, маршрутизації повідомлень на основі онтологічного метаопису застосувань та їхньої взаємодії . Це спрощує організаціям обслуговування та масштабування своїх застосувань та керування ними.

З комерційної точки зору розрізняють різні типи сервісів:

Внутрішні сервіси - такі, що підтримуються та використовуються організацією для виконання власних процесів.

B2B (buisines to buisines) – сервіси, що підтримуються та надаються партнерами для забезпечення взаємодії в спільних процесах.

B2C (buisines to consumer) – сервіси, що підтримуються організацією та надаються клієнтам для взаємодії з бізнесом, в тому числі для надання послуг, зокрема і на комерційній основі.

G2B (government to buisines) – сервіси, що підтримуються державою та нада-

ються суб'єктам господарювання для надання адміністративних послуг.

На сьогодні використання корпоративних сервісних шин (ESB) обмежено в основному застарілими системами, що потребують взаємодії компонент за різними протоколами. Розвиток Інтернету та всеосяжне впровадження WEB-протоколів передачі даних практично витіснило інші протоколи. Тому складна архітектура з громіздкою централізованою шиною, через яку взаємодіють монолітні прикладні застосування, повсюдно замінюється архітектурами на основі WEB сервісів, а монолітні застосування декомпозуються на ряд слабко зв'язаних легких застосувань – мікросервісів, які взаємодіють один з одним через API (програмний інтерфейс). Клієнт-серверні десктоп-застосування, які поєднують бізнес логіку та користувацький інтерфейс витісняються триланковими застосуваннями з організацією взаємодії з користувачем через WEB інтерфейс за допомогою браузера. Це спрощує доступ до прикладних систем та дозволяє розповсюджувати та надавати програмне забезпечення як послугу (сервіс) не тільки через програмний інтерфейс, а й через інтерфейс користувача.

Завдяки такій архітектурі значно спрощується масштабування системи та її гнучкість. Автономні сервіси можна легко переміщувати з одного сервера на інший, поєднувати сервіси в єдину систему еволюційним шляхом у процесі експлуатації без необхідності перезбирати монолітне застосування під час зміни або додавання функціональності.

Впровадження не потребує повного переналаштування процесів та надає можливість використовувати звичні, добре зарекомендовані застосування. Достатньо лише додати відповідний інтерфейс для вже готової функціональності.

Мікросервісна архітектура дає підприємству можливість швидше адаптуватись до змін бізнесу, замінюючи реалізацію сервісу, залишаючи незмінним його інтерфейс. Використання стандартних протоколів взаємодії дозволяє поєднувати компоненти, розроблені на різних платформах та різними мовами.

Сервіс-орієнтована архітектура пропонує розробнику новий підхід до використання готових рішень без необхідності вбудови його коду в монолітне застосування. Замість цього надається композиція складних сервісів із готових розгорнутих сервісів, що реалізують частини процесу. Одночасно сервіси можуть бути розподілені в мережі і навіть належати різним компаніям та надаватись як послуги, не потребуючи розгортання та підтримки.

Розподілена сервісна архітектура прикладних застосувань суттєво відрізняється від монолітних застосувань, які виконуються в межах адресного простору одного процесора та взаємодіють через спільну оперативну пам'ять та базу даних.

Забезпечення взаємодії окремих компонент сервіс-орієнтованої системи (сервісів), що виконуються на різних процесорах та використовують власні бази даних для зберігання своїх даних, потребує інших архітектурних підходів, заснованих на обміні повідомленнями через мережу. Залежно від потреб організація обміну повідомленнями може здійснюватися синхронно або асинхронно.

Синхронний запит/відповідь передбачає відправку клієнтом запиту видавцю сервісу та очікування відповіді від видавця. Виконання клієнта блокується на час очікування та його виконання продовжується після отримання відповіді

Асинхронний запит/відповідь передбачає відправку клієнтом запиту без очікування відповіді. Клієнт не блокується на час очікування, а обробка відповіді здійснюється обробником події після надходження відповіді.

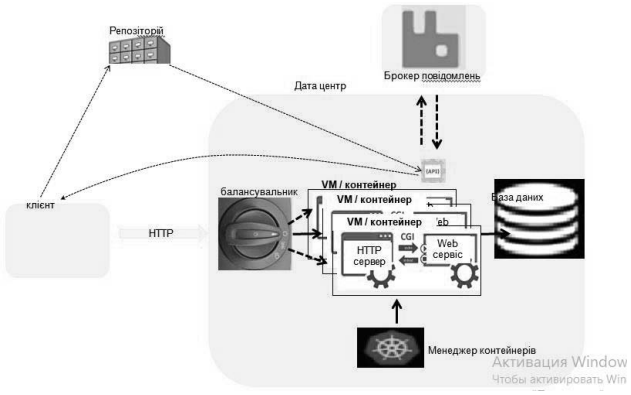

Рис. 3. Основні компоненти СОА

Взаємодія здійснюється через віддалений виклик (RPI- remoute procedure invocartion) сервісу через транспортний протокол мережі, переважно через посередництво HTTP сервера, який бере на себе багатопоточне прослуховування мережевого каналу приймання вхідного повідомлення та передачу його на виконання сервісу та відправлення результату виконання. Крім того, асинхронна взаємодія може здійснюватись через посередництво так званих брокерів.

Взаємодія через віддалений доступ обумовлює жорстку залежність від сервісу видавника та вимагає його постійної готовності, що не завжди можливо, оскільки завжди існує ризик часткової відмови на запит, який може бути обумовлений збоями обладнання та мережевого каналу або недоступністю сервісу у зв'язку з технічним обслуговуванням. Крім того, сервіс може бути перевантаженим та не відповідати на запити вчасно.

Тому СОА-система повинна передбачати відповідні заходи для запобігання таких ситуацій. Для цього можуть бути використані наступні шаблони:

Мережевий час очікування. Повторювання запиту у разі перевищеня певного часу очікування, що може запобігти впливу збоїв.

Обмеження кількості запитів. Після обумовленої кількості невдалих запитів, блокує ймовірно недоступний сервіс.

Запобіжник перевіряє доступність сервісу контрольним викликом через певний час та за потреби знімає блокування.

Виклик сервісу здійснюється за його мережевою адресою, яка може змінюватись у випадку фізичного переміщення сервісу, що призводить до відмови його клієнтів та потребує відповідного корегування адреси в коді клієнта, його перезбирання та перевстановлення. В разі використання сервісів на власних ресурсах ситуація може бути полегшена використанням конфігураційних файлів, які корегуються під час переміщення сервісів, що допомгає уникнути необхідності перезбирання клієнтів. Однак адреси екземплярів зовнішніх сервісів під контролем інших організацій а також сервісів, розміщених у хмарних середовищах, можуть мінятися динамічно. Більше того, набір цих екземплярів може постійно мінятися через постійне масштабування, відмови та оновлення, що потребує засобів динамічного виявлення актуальних адрес сервісів. Механізм динамічного виявлення сервісів заснований на використанні реєстру, здійснюється адміністратором або автоматично самим сервісом вбудованими засобами. Функції такого реєстру може виконувати, зокрема, звичайний DNS сервер.

Такий механізм виявлення дозволяє легко масштабувати систему у разі перевантаження певних сервісів та динамічно балансувати навантаження. Якщо кількість запитів до сервісу починає перевищувати його можливості, запускається ще один або кілька екземплярів, які реєструються в реєстрі, й запит спрямовується до якогось із них, використовуючи одну із стратегій балансування:

Round Robin - обслуговування по колу, за якого кожний наступний запит передається наступному зареєстрованому в реєстрі екземпляру;

Weighted Round Robin - враховує продуктивність екземпляру, визначену ваговим коефіцієнтом;

Least Connections - враховує кількість підключень до екземплярів на даний момент;

Destination Hash Scheduling і Source Hash Scheduling Sticky Sessions - враховує зони мережевого розташування клієнта на основі IP-адреси.

Механізм взаємодії сервісів через RPI потребує активності обох компонент, чого не завжди можна досягти. Цього недоліку можна уникнути в архітектурі на основі обміну повідомленнями через посередництво брокерів подібно архітектурі ESB. Згідно із цим механізмом сервіс - відправник записує повідомлення в канал, а отримувач зчитує його з цього каналу. Під час відправки повідомлення активність отримувача не обов'язкова, оскільки повідомлення зберігається в каналі доки отримувач його не прочитає та не опрацює.

Повідомлення складається із заголовку, який містить метадані щодо повідомлення, включаючи ідентифікатор пові-

домлення та зворотну адресу, що визначає канал, куди потрібно спрямувати повідомлення-відповідь, та тіло повідомлення, яке містить власне дані в узгодженому форматі (текстовому або двійковому).

За специфікою обробки повідомлень розглядають два типи каналів:

- «крапка — крапка» - використовуються для доставки повідомлення тільки одному отримувачу, який зчитує їх із цього каналу та обробляє його. Як правило, через такі канали передаються повідомлення типу «команда»;
- «видавець — підписник» доставляє кожне повідомлення всім підключеним споживачам. Зазвичай через такі канали передаються повідомлення типу «подія».

Взаємодія за допомогою повідомлень за своєю природою асинхронна, оскільки після відправлення повідомлення сервіс не очікує відповіді, а сам ініціює його отримання з відповідного каналу, використовуючи ідентифікатор повідомлення наданого в повідомленні запиту разом з каналом відправника. Крім того, на відміну від віддаленого виклику, відповіді можуть оброблятися будь - яким екземпляром сервісу, незалежно від того, який екземпляр здійснив запит.

API асинхронного сервісу, який використовує обмін повідомленнями крім операцій включає опис подій, які він публікує в каналах типу видавець-підписник.

Також, як і для віддаленого виклику сервісу взаємодія через повідомлення потребує знання дислокації «каналів» в мережі та протоколів доставки повідомлень. Тому повинен використовуватись механізм виявлення, але не для адреси сервісу, а для його каналів. Також, як і під час віддаленого виклику, потрібно гарантувати доступність «каналу», що може бути проблематичним у разі великого його завантаження. З цими проблемами можна впоратися успішніше, використовуючи не пряму відсилку повідомлень в канал, а через посередництво спеціального сервісу-брокера. Він не аналізує тіло запитів, а лише виконує приймання повідомлень та їх збереження в своїй базі даних і видачу повідомлень на запити споживачів.

Для виконання запиту клієнта достатньо взаємодіяти лише з брокером та відправляти повідомлення у відповідний канал. Клієнту не потрібно використовувати механізм виявлення сервісу та турбуватись про масштабування.

Брокер буферизує повідомлення доти, доки вони не зможуть бути оброблені, і взаємодія сервісів в системі буде здійснюватись навіть у разі тимчасової недоступності сервісу-виконавця.

Потенційно вузьке місце продуктивності централізованого обробника повідомлень долається завдяки відсутності прикладної обробки повідомлення, а лише його збереженням в БД а також через балансування навантаження сервісу-брокера.

Брокер може протоколювати процес обміну повідомленнями, авторизації під час доступу до каналів та виконувати інші функції, пов’язані з обміном.

Залежно від організації зберігання повідомлень, розрізняють два типи брокерів:

Брокер повідомлень - сервіс, який приймає повідомлення від клієнта та зберігає його в одній або декількох чергах видавництв сервісів доти, доки сервіс видавця не забере повідомлення зі своєї черги та не виконає відповідні дії, результатом яких може бути розміщення відповіді в черзі клієнтського сервісу.

Брокер подій - використовує парадигму видавця-підписника. При цьому в брокері повідомлень кожний сервіс не має власної черги. Замість цього клієнти розміщують повідомлення в базі брокера в чергах відповідно до типу повідомлення, на яке підписуються сервіси-обробники, які отримують ці повідомлення від брокера подій на основі адреси, наданої під час підписки або шляхом запитів до брокера.

В сервіс-орієнтованому застосуванні окремі сервіси мають «власну модель» БД, тому потребують підтримки розподілених транзакцій. Механізм двофазної фіксації (2PC), яку використовують в монолітних застосуваннях під час роботи з декількома базами даних передбачає жорстку взаємозалежність та доступність сервісів у процесі виконання розподіленої тран-

закції, що суперечить принципу слабкої зв'язаності сервісів в СОА.

Згідно з відомою CAP теоремою Брюєра [8] неможливо одночасно забезпечити: узгодженість даних в усіх вузлах (Consistency), доступність вузлів(Availability), стійкість до розділення (Partition tolerance). Оскільки в сервіс-орієнтованому застосуванні апріорі підтримується Partition tolerance, неможливо забезпечити Consistency або Availability, які потрібні для 2PC, тому для підтримки розподілених транзакцій в СОА застосовується інший архітектурний шаблон «оповідання» «SAGA». Цей шаблон забезпечує узгодженість даних, користуючись послідовністю прямих та компенсуючих транзакцій, які координуються асинхронними повідомленнями.

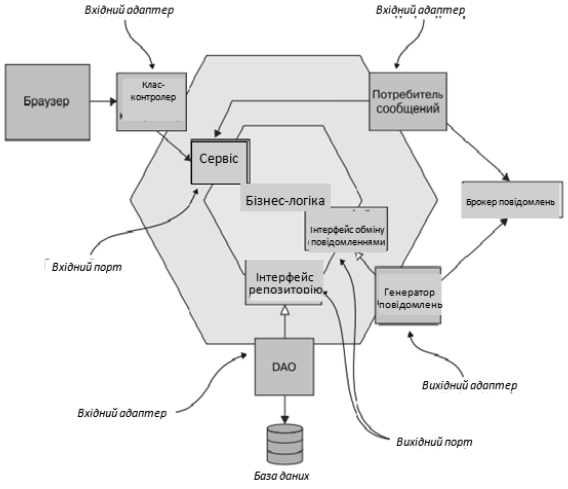

Рис. 4. Архітектура сервісу в СОА

Архітектура сервісів відрізняється від пошарової архітектури (MVC, MVP, MVPP) монолітних застосувань. Сервіси в СОА зазвичай реалізують бізнес логіку операцій пов'язаних з одним суб'єктом, відстежують його стан у своїй автономній БД та виконують свій функціонал за запитами через API. Тому архітектурно сервіс має не пошарову архітектуру, а так звану шестигранну [9] з ядром (рис 2), яке виконує бізнес-логіку, та адаптерами, які забезпечують взаємодію. Механізм віддаленого виклику сервісу заснований на використанні програмного інтерфейсу (API) сервісу, який визначає перелік операцій, що можуть викликатись, та подій, що можуть генеруватись під час виконання. Для кожної операції визначаються формат та семантика даних, необхідних для її виконання, а також формат та семантика результату виконання операції. Для події визначається її тип, формат та семантика пов'язаних даних, що публікуються в каналі взаємодії. Клієнтська бізнес-логіка для віддаленого виконання операції зовнішнім сервісом звертається до проксі-інтерфейсу – класу адаптера, який інкапсулює узгоджений API. RPI-проксі формує та відправляє через мережевий протокол (як правило HTTP) повідомлення-запит сервісу. Запит приймається класом-адаптером RPI-сервер, який витягує з повідомлення дані відповідно до API, та викликає бізнес-логіку сервісу . Після відпрацювання бізнес-логіка сервісу викликає метод RPI-серверу, який запаковує результат виконання відповідно до API, та відправляє відповідь через мережевий протокол, який приймається методом RPI-проксі. А той витягує дані результату та викликає обробку результату клієнтською бізнес-логікою. Водночас надсилання HTTP запиту клієнтом може здійснюватися синхронно так, що процес клієнта очікує відповідь сервісу, або асинхронно таким чином, що процес клієнта продовжується, а обробка результату здійснюється методом–обробником події приходу результату HTTP-запиту, який вказується у надсиланні запиту.

Завдяки цьому внутрішні зміни бізнес-логіки операцій не впливають на клієнтів, якщо не змінюється API.

Стандарти сервіс-орієнтованої парадигми

Сервіс-орієнтована архітектура прикладних систем покладається на взаємодію її окремих компонент, що функціонують на різних платформах в різних обчислювальних середовищах, підпорядкованих різним організаціям через повідомлення по мережі. Це в свою чергу потребує узгоджених правил (протоколів) прийому/передачі даних, їхніх форматів як на фізичному, так і на логічному рівні, які підтримуються різними платформами, а також метаопису компонент та їхніх інтерфейсів.

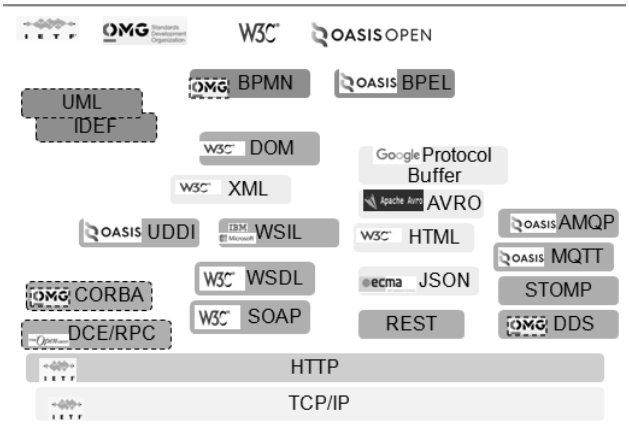

Рис. 5. Стандарти СОА

Для впорядкування досвіду та досягнення сумісності продуктів підтримки локальних мереж від різних виробників, які стрімко почали розвиватись, міжнародна організація із стандартизації OSI (Open system interconnection) розробила базову модель мережевої взаємодії, яку офіційно опублікувала 1983 року. Паралельно в процесі розвитку глобальних мереж Internet Інженерною радою Інтернету IETF (Internet Engeniering Task Force) була сформована альтернативна модель TCP/IP і відповідний стек протоколів, які поступово стали домінуючими, яка складалась із 4 шарів (замість 7 рівнів OSI):

Канальний рівень (network access layer) визначає параметри фізичного середовища передачі та спосіб кодування даних.

Міжмережевий рівень (Internet layer) регулює передачу даних з однієї мережі в іншу. На цьому рівні працюють маршрутизатори, які спрямовують пакети до потрібної мережі за її адресою.

Транспортний рівень визначає механізми перевірки правильності пересилки та доставки пакетів потрібному застосуванню в правильному порядку з підтвердженням доставки (TCP) або без підтвердження (UDP).

Прикладний рівень забезпечує узгоджені правила та формати обміну інформації між прикладними застосуваннями. Залежно від призначення прикладного застосування можуть бути різні протоколи прикладного рівня. Наприклад, протокол HTTP для інтернет-браузерів, протокол FTP - для передавання файлів, протоколу, SMTP для електронної пошти, в тому числі усі стандарти, що регламентують сервісно-орієнтовану взаємодію.

Протоколи прикладного рівня для взаємодії прикладних застосувань через мережу почали з'являтися із 1970-х років. Зокрема, стандарт DCE/RPC визначив механізм віддаленого виклику процедури, що виконується в іншому адресному просторі, з передачею необхідних параметрів та отриманням результатів через мережу, аналогічно передачі параметрів по значенню через стек ОП в монолітному застосуванні.

Пізніше компанія Microsoft розповсюдила свою компонентну технологію COM взаємодії різних об'єктних застосувань в одному адресному просторі (так званий механізм OLE) на розподілені системи DCOM, додавши механізм віддаленого виклику методів об'єкта, існуючого в застосуванні, що виконується на іншому комп'ютері (сервері) з маршалінгом (передачею даних між клієнтом та сервером через мережу) на тих же засадах, що і віддалений виклик процедур (RPC), використовуючи метаопис інтерфейсу методів об'єктів спеціальною мовою опису інтерфейсу (Interface Definition Language). Однак ця технологія та стандарт дозволяє поєднувати тільки застосування під операційною системою Windows.

Для формування кросплатформених механізмів взаємодії прикладних застосувань групою OMG (Object Management Group), до складу якої входять представники найбільших виробників програмного забезпечення, було розроблено технологію та відповідний стандарт CORBA (Common Object Reqest Broker). Технологія заснована на використанні спеціальної мови опису інтерфейсу (IDL-Interface Definition Language) з об'єктами, які виконуються на серверах розподіленої системи та проблемно-незалежних компонент (ORB core), які забезпечують створення/виявлення екземплярів об'єктів та віддаленого виклику їхніх методів, використовуючи програмні адаптери (Stubs/Sceleton), які генеруються автоматично за описом інтерфейсу мовою IDL.

Водночас швидко розвивається Всесвітня павутина WWW, заснована на доступі до текстової та графічної інформації на основі обміну контентом у форматі HTML між браузером та сервером через протокол прикладного рівня HTTP. Завдяки цьому підтримка протоколу HTTP вбудована практично в усі операційні системи та платформи.

Платформонезалежна базова модель структурованого документа (DOM), запропонована консорціумом W3C, що опікується стандартами WEB, та призначена для роботи з документами мовою розмітки документа HTML отримала реалізації практично усіма мовами програмування та широко використовувалась не тільки в браузерах для роботи з HTML-документами, а й в прикладних застосуваннях для роботи з документами в форматі розширюваної мови розмітки XML, також запропонована консорціумом для текстового представлення структурованих даних опису об'єктів.

Тому компанією Microsoft 1998 року було запропоновано та передано консорціуму W3C для подальшого супроводу та розповсюдження спрощений протокол SOAP (Simple Object Access Protokol) взаємодії прикладних застосувань через Інтернет на основі віддаленого виклику процедур обміну даними в текстовому форматі XML. Цей протокол було доповнено ще двома взаємопов'язаними специфікаціями, що забезпечують процеси публікації прикладних застосувань в Інтернет-середовищі (Вебсервісів) та їх використання:

SOAP – визначає порядок виконання обміну повідомленнями, синтаксис та основну структуру повідомлення, правила їхнього формування та інтерпретації у відправника та приймальника.

WSDL - визначає формальний опис програмного інтерфейсу (API) WEB сервісу, який по суті є XML схемою SOAP повідомлень за правилами XSD

UDDI - визначає стандартний API обмін з реєстром та XML-схему онтологічного опису SOAP WEB сервісів, в якому постачальники сервісів можуть розміщувати інформацію про себе, сервіси, які вони забезпечують, та порядок їх використання, в тому числі посилання на WSDL специфікацію сервісу.

Крім цієї трійки стандартів, регламентуючих основні етапи взаємодії SOAP сервісів, W3C визначила низку так званих WS-* XSD специфікацій API відносно різних аспектів організації взаємодії між сервісами – (автентифікація, безпека, розподілені транзакції тощо).

Протокол SOAP покладається на формат XML, обмежує його застосування, обумовлене накладними витратами у передаванні даних в інших форматах, більш пристосованих для конкретної прикладної галузі застосування сервісів. Зокрема, одним з найпоширених споживачів сервісів є WEB застосування, які працюють у середовищі браузерів та викликають сервіси скріптами мовою JAVAscript, прийнятої за стандартну, та яка має підтримку практично в усіх браузерах. У стандарті цієї мови визначений інший формат текстового уявлення структурованих даних JSON, відмінний від XML, та більш компактний, що ускладнює використання SOAP сервісів у WEB застосуваннях.

Тому 2000 року Рой Філдинг в своїй дисертації [6] виклав основні принципи технології взаємодії застосувань через інтернет на основі прямого використання протоколу HTTP без обмеження на формат передачі даних.

Ця технологія або архітектурний стиль отримала назву REST (REpresentational State Transfer). І, хоча не отримала «офіційного» статусу стандарту, але швидко стала домінуючою в галузі сервіс-орієнтованих застосувань завдяки забезпеченню корисних властивостей:

- широке розповсюдження базового HTTP протоколу;
- надійність (через відсутність необхідності зберігати стан клієнта під час взаємодії);
- продуктивність (через використання кешу);
- масштабування (через використання посередників – брокерів та балансувальників);

 простота інтерфейсів (відсутність необхідності підтримки формального опису).

Відсутність обмежень на формат повідомлень дозволяє використовувати будь-який найбільш доцільний формат повідомлень, виходячи з призначення сервісу, в тому числі той самий XML, як в SOAP. Однак більшої популярності набув формат JSON за рахунок своєї лаконічності в порівнянні з XML (що забезпечує більшу продуктивність через зменшення трафіку) та підтримці в JavaScript (вбудованої мови браузерів). Крім того, він дозволяє використання бінарних форматів, таких як Avro або Protocol Buffers, що здатні в деяких випадках ще більше скоротити трафік при взаємодії.

Асинхронна взаємодія застосувань передбачає застосування спільного сховища або брокера для тимчасового зберігання повідомлень. Водночас постачальники та споживачі повідомлень мають використовувати узгоджені протоколи та формати, що дозволяють виявити призначені їм повідомлення в спільному середовищі та обробити їх. Для цього авторитетні організації із розробки стандартів СОА пропонують протоколи підтримки асинхронної взаємодії за різними схемами, такими як крапка – крапка, або публікація/підписка:

AMQP – Advancing Message Queuing Protocol,

MQTT – Message Queuing Telemetry Transport,

STOMP – Simple/Streaming Text Oriented Protocol,

DDS- Data Distribution Service.

Розробка сервіс-орієнтованих застосувань покладається в основному на композицію процесів із готових рішень, що реалізують окремі операції – частини процесу. Для цього потрібно спочатку декомпозувати бізнес-процеси, що автоматизуються на окремі операції, визначити потрібні сервіси, які можуть їх виконати та знайти готові рішення, або розробити нові в разі відсутності потрібних. Така робота потребує методів моделювання процесів, зрозумілих як системним аналітикам–фахівцям у предметній галузі, так і програмістам, що їх реалізують. Для проєктування монолітних застосувань використовувались відповідні графічні нотації, що дозволяють наочно представити склад операцій їхньої взаємозв'язки та послідовність виконання. В структурних застосуваннях це нотації IDEF в об'єктно-орієнтованих - UML. Сервіс-орієнтовані застосування часто використовують подіє-орієнтовані механізми взаємодії, які не підтримуються в цих нотаціях. Тому групою OMG було розроблено нотацію BPMN (Buisines Process Modeling Notation), яка усуває цей недолік. Крім графічного представлення ця специфікація визначає еквівалентний формальний опис процесів мовою XML відповідної схеми.

Якщо в таку модель додати специфікацію реалізацій та інтерфейсів складових сервісів, а також умов, що визначають послідовність їх виконання, то таку модель можна інтерпретувати для виконання процесу. Таку специфікацію BPEL (Buisines Process Execution Language) для оркестрування SOAP сервісів, засновану мовою XML, також було запропоновано IBM та передано в OASIS для супроводу.

Програмна інженерія сервіс-орієнтованої парадигми

Технологія розробки, впровадження та супроводу сервіс-орієнтованих систем суттєво відрізняється від розробки систем монолітної архітектури через їхню розподілену природу. Сервіс-орієнтований підхід починає застосовуватись тоді, коли предметна галузь системи перетинає межі організацій та підрозділів та її об'єм починає перевищувати наявні обчислювальні можливості комп'ютерів а супровід стає погано керовану через необхідність реагування на зміни великої кількості процесів, поєднаних в монолітному застосуванні.

Для виконання процесів сервіс-орієнтовані системи використовують компоненти, які знаходяться в зоні відповідальності різних підрозділів і навіть організацій, і тому потребують узгодження програмних інтерфейсів для інформаційної взаємодії та послідовності їх виконання і відповідно проєктування та декомпозиції складних процесів.

Декомпозиція сервіс-орієнтованої системи на окремі компоненти – сервіси заснована на предметно-орієнтованому проєктуванні, згідно з яким до функціональності сервісу залучаються операції, пов'язані з відносно самостійною групою ресурсів, що гарантує слабку зв'язаність сервісу.

В основу декомпозиції системи на сервіси можна покласти принципи єдиної відповідальності (Single Responsibility Principle, SRP) та узгоджених змін (Common Closure Principle, CCP), запозичені з об'єктно-орієнтованого проєктування.

Згідно із принципом SRP кожен сервіс повинен мати лише одне призначення і відповідно єдину причину для зміни, а згідно із принципом CCP причини зміни операцій сервісу мають бути однакові.

Дотримання цих принципів декомпозиції дозволить мінімізувати кількість сервісів, які потрібно буде редагувати та розгортати зі зміною якоїсь вимоги.

Окремі операції спроєктованої системи можуть бути вже реалізовані різними провайдерами в сервісах, розгорнутих на їхніх ресурсах та доступних для використання . Щоб скористатися таким готовим сервісом, клієнт повинен бути обізнаним із функціоналом, програмним інтерфейсом (API), адресою сервісу та умовами надання. А для цього провайдер сервісу повинен опублікувати такий опис на ресурсах, доступних потенційним користувачам.

Для SOAP WEB сервісів був запропонований стандарт UDDI онтологічних мета описів сервісів та інтерфейс доступу до них для створення як публічних, так і відомчих репозиторіїв. Цей стандарт був відтворений у різних реалізаціях як репозиторіїв, так і їхніх клієнтів, що забезпечували публікацію мета-описів сервісів, пошук сервісів за елементами специфікації, отримання опису API.

Однак згодом Rest сервіси значно потіснили SOAP, тому UDDI репозиторії не виправдали очікуваного поширення. 2006р. IBM, Microsoft та SAP оголосили про закриття своїх загальнодоступних UDDI вузлів. 2007 року завершена підтримка UDDI в OASIS. 2010 року Microsoft оголосила про припинення підтримки UDDI в Windows Server. Тому пошук потрібних сервісів та їхніх постачальників може здійснюватися в основному за допомогою загальних пошукових механізмів інтернету на основі ключових слів, а отримання опису їх API здійснювати зі знайденого сайту. Питання створення репозиторіїв для забезпечення пошуку потрібних сервісів за їхнім онтологічним описом залишається відкритим. Водночас онтологічні схеми в таких репозиторіях не повинні обмежуватись одним стандартом та передбачати в своєму описі визначення протоколів та форматів взаємодії. Додавання в онтологію опису крім синтаксичного опису API ще семантики функціональності та повідомлень сервісу та існуючих екземплярів може розширити призначення репозиторію не тільки для пошуку потрібних сервісів, а й для виявлення потрібних сервісів в процесі виконання з балансуванням навантаження та оптимізацією витрат. Доцільно розглянути підхід створення розподілених репозиторіїв на базі WSIL (Wev Servcies inspection language), запропонований сумісно IBM та Microsoft.

Композитні сервіси, що реалізують поєднання окремих сервісів для виконання складних процесів відповідно до спроектованої бізнес-логіки, можуть розроблятися на будь-якій платформі з використанням бібліотек підтримки стандартних протоколів взаємодії . Для організації асинхронних відкладених викликів сервісів можна використовувати готові брокери повідомлень. Крім того, для композиції сервісів, спроєктованих у стандартних нотаціях (BPMN, BPEL), можна реалізовувати за допомогою спеціалізованих платформ та інтерпретаторів, що їх підтримують.

Деякі операції можуть потребувати втручання людини для вводу даних, необхідних для виконання сервісу та аналізу результатів його виконання для ухвалення рішень.

Такі операції реалізуються здебільшого WEB застосуванням, побудованим за AJAX технологією.

Водночас інтерфейс користувача для браузера формується за допомогою

спеціалізованих фреймворків, а для виклику сервісів використовуються засоби підтримки HTTP протоколу, присутні практично на усіх платформах.

Якщо для операцій не знайшлося готової реалізації, потрібно створити відповідні компоненти сервіси. Окремий сервіс розробляється як неінтерактивне монолітне застосування, яке відстежує стан ресурсів в окремій базі даних та видачу цього стану за запитами через програмні інтерфейси.

Сервіс розробляється, переважно, в об'єктно-орієнтованій парадигмі з використанням бібліотек підтримки протоколів та стандартних форматів повідомлень для обраної платформи розробки.

Прослуховування мережевого каналу, прийом та відправку повідомлень, як правило, делегується HTTP-серверу, взаємодія з яким здійснюється за обумовленими протоколами (CGI, JavaServlet …), підтримка яких вбудована у відповідні бібліотеки.

Методологія структурного проєктування та програмування SADT , що застосовувалась для розробки складних монолітних систем, рано чи пізно стикається з проблемою росту термінів випуску версій системи у зв'язку з неконтрольованим зростанням кількості залежностей. В результаті система не взмозі реагувати на зміни бізнес правил та умов виконання процесів у прийнятні терміни.

Розробка складних систем у сервіс-орієнтованій архітектурі взмозі швидко реагувати на зміни бізнес правил та умов використання за рахунок того, що вона складається з відносно невеликих частин. Команда ж розробників сервісу може бути невеликою та розвивати й підтримувати сервіс у разі зміни внутрішніх правил виконання операцій сервісу незалежно від інших пов'язаних сервісів, доки зовнішній узгоджений програмний інтерфейс залишається незмінним. Однак зберегти API вдається не завжди, тому для забезпечення працездатності усієї СОА-системи у разі заміни версії сервісу, потрібно забезпечувати сумісність версій або підтримку всіх версій API, доки вони використовуються його клієнтами. Подолати такі негаразди допомагає специфікація семантичного версіонування, згідно з якою номер версії формується з трьох частин, які інкрементуються під час випуску нової версії відповідно до характеру змін:

Major – під час внесення в API несумісних змін (видаляються параметри або змінюється їхня семантика).

Minor – під час зміни API з підтримкою зворотної сумісності (додавання нових атрибутів до запиту або відповіді; додавання нових операцій. Водночас сервіси повинні надавати значення за замовченням для нових атрибутів, а клієнти можуть ігнорувати будь-які зайві атрибути).

Patch - виправлення помилок з підтримкою зворотної сумісності.

Відповідність версії API сервісу та клієнта здійснюється на основі номера версії, що вказується в повідомленні. Вносячи мажорні зміни, важливо підтримувати і попередні версії API, для чого номер версії може бути частиною адреси сервісу.

Така методологія швидкого реагування на зміни бізнес-правил та постійного вдосконалення системи отримала назву екстремального програмування (AGILE), заснованого на 12 простих принципах [10], одним з яких є принцип тісної взаємодії розробників та користувачів системи (DEVelop & OPerationS).

Для дотримання цих принципів максимально автоматизується взаємодія розробників та користувачів, а також операції технологічного циклу випуску версій сервісів із застосуванням відповідних засобів для планування модифікацій та їх реалізації, тестування та розгортання.

Висновки

Сервіс-орієнтована парадигма побудови складних розподілених систем є закономірним результатом еволюції підходів до розробки від процедурних та об'єктно-орієнтованих монолітних систем в єдиному адресному просторі до взаємодіючих мережею слабко зв'язаних компонентів у розподіленому середовищі.

Сервіс-орієнтована парадигма уможливлює створення прикладних систем шляхом інтеграції готових компонент

в єдиний технологічний ланцюг без необхідності поєднання їх в монолітне застосування та розгортання на власних ресурсах. Використання їх, як є, на ресурсах провайдера на основі стандартів взаємодії та узгодженого програмного інтерфейсу (API) є практично доцільним. Rest API на основі HTTP та JSON домінує та практично витіснив рішення на основі стандартів взаємодії SOAP. Зокрема, загальні реєстри UDDI публікації онтологічних описів SOAP сервісів, які дозволяли здійснювати пошук готових сервісів, необхідних для компонування прикладної системи, стали неактуальними. Хоча потреба в пошуку готових рішень залишається. Тому створення аналогічних загальнодоступних реєстрів готових рішень з онтологією опису, не обмеженою протоколом SOAP, є актуальним завданням.

Функціонал, для якого не знайдено відповідного готового рішення розробляється як на традиційних компілюючих платформах, так і в сучасних інтерпретуючих платформах WEB програмування в об'єктно-орієнтованій парадигмі з дотриманням стандартів взаємодії SOAP та REST.

Відпрацьована формальна методологія SADT організації розробки монолітних застосувань витіснена більш гнучкою методологією екстремального програмування AGILE, яка значно скорочує тривалість технологічного циклу розробки/впровадження, та дозволяє швидше реагувати на зміни правил виконання бізнес процесів. Дотримання принципів екстремального програмування потребує використання спеціалізованих програмних засобів організації розробки включно із плануванням та відстеженням завдань, а також – автоматизацією тестування та розгортанням версій системи.

Література

1. А.Пискун, Краткий путеводитель по семейству нотаций IDEF. https://www.antonpiskun.pro/kratkij-putevoditel-po-semejstvu-notaczij-idef/
2. Donald Bell, UML basics: An introduction to the Unified Modeling Language. https://cs.nyu.edu/~jcf/classes/g22.2440-001_sp06/handouts/UMLBasics.pdf
3. С.Фейт, TCP/IP Архітектура, протоколи, реалізація. ISBN 5-85582-072-6 https://coollib.cc/b/163834-sidni-m-feyt-tcp-ip-arhitektura-protokolyi-realizatsiya-vklyuchaya-ip-versii-6-i-ip-security/read
4. Internet Engineering Task Force (IETF) Request for Comments: 7540 May 2015 Hypertext Transfer Protocol Version 2 (HTTP/2) https://datatracker.ietf.org/doc/html/rfc7540
5. D.Tidwell, J.Snell, P.Kulchenko, Programming Web Services with SOAP. https://theswissbay.ch/pdf/Gentoomen%20Library/Programming/CSharp/OReilly%20-%20Programming%20Web%20Services%20with%20Soap.pdf
6. R.Th.Fielding, Architectural Styles and the Design of Network-based Software Architectures, https://ics.uci.edu/~fielding/pubs/dissertation/top.htm
7. OASIS. Reference Model for Service Oriented Architecture 1.0 Committee Specification 1, 2 August 2006 https://groups.oasis-open.org/higherlogic/ws/public/download/19679/soa-rm-cs.pdf/latest
8. S.Gilbert, N.Lynch, Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services https://web.archive.org/web/20160213135959/http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.1495&rep=rep1&type=pdf
9. К.Ричардсон, Микросервисы: Патерны разработки и рефакторинга. Издательский дом "Питер", 2020. https://coderbooks.ru/mikroservisy-patterny-razrabotki-i-refaktoringa
10. Agile-маніфест розробки програмного забезпечення, https://agilemanifesto.org/iso/uk/manifesto.html

References

1. A.Piskun, A Brief Guide to the Notation Family IDEF. https://www.antonpiskun.pro/kratkij-putevoditel-po-semejstvu-notaczij-idef/
2. Donald Bell, UML basics: An introduction to the Unified Modeling Language. https://cs.nyu.edu/~jcf/classes/g22.2440-001_sp06/handouts/UMLBasics.pdf
3. S.Fate, TCP/IP Architecture, protocols, implementation. ISBN 5-85582-072-6

https://coollib.cc/b/163834-sidni-m-feyt-tcp-ip-arhitektura-protokolyi-realizatsiya-vklyuchaya-ip-versii-6-i-ip-security/read
4. Internet Engineering Task Force (IETF) Request for Comments: 7540 May 2015 Hypertext Transfer Protocol Version 2 (HTTP/2) https://datatracker.ietf.org/doc/html/rfc7540
5. D.Tidwell, J.Snell, P.Kulchenko, Programming Web Services with SOAP. https://theswissbay.ch/pdf/Gentoomen%20Library/Programming/CSharp/OReilly%20-%20Programming%20Web%20Services%20with%20Soap.pdf
6. R.Th.Fielding, Architectural Styles and the Design of Network-based Software Architectures, https://ics.uci.edu/~fielding/pubs/dissertation/top.htm
7. OASIS. Reference Model for Service Oriented Architecture 1.0 Committee Specification 1, 2 August 2006 https://groups.oasis-open.org/higherlogic/ws/public/download/19679/soa-rm-cs.pdf/latest
8. S.Gilbert, N.Lynch, Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services https://web.archive.org/web/20160213135959/http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.1495&rep=rep1&type=pdf
9. K.Richardson, Microservices: Patterns of development and refactoring. Publishing house "Piter", 2020. https://coderbooks.ru/mikroservisy-patterny-razrabotki-i-refaktoringa
10. Agile-software development manifesto, https://agilemanifesto.org/iso/uk/manifesto.html

Одержано: 12.04.2025
Внутрішня рецензія отримана: 19.04.2025
Зовнішня рецензія отримана: 28.04.2025

Про авторів:

[1]*Андон Пилип Іларіонович*,
академік НАН України.
https://orcid.org/0009-0000-1737-5579.

[1]*Суботін Сергій Васильович*,
науковий співробітник.
https://orcid.org/0000-0002-5958-0259.

[1]*Перконос Петро Іванович*,
науковий співробітник.
https://orcid.org/0000-0002-5958-0260.

[1]*Твердохліб Євген Миколайович*,
кандидат технічних наук,
старший науковий співробітник.
https://orcid.org/0000-0001-6594-5468.

Місце роботи авторів:

[1]Інститут програмних систем
НАН України,
тел. +38-044-522-62-42
E-mail: ukrprog@isofts.kiev.ua
www.iss.nas.gov.ua