DOI: 10.20998/2079-0023.2024.01.14
УДК 004.9

Д. І. ГОЛЬДІНЕР, аспірант, Харківський національний університет радіоелектроніки, м. Харків, Україна; e-mail: denys.holdiner@nure.ua; ORCID: https://orcid.org/0000-0002-1456-1867

РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ МОДЕЛЮВАННЯ СИСТЕМ МАСОВОГО ОБСЛУГОВУВАННЯ ПІД ІМПЛЕМЕНТАЦІЮ МОВОЮ ПРОГРАМУВАННЯ GO

Предметом дослідження статті є – методи та підходи до організації архітектури програмної реалізації, призначеної для моделювання поведінки систем масового обслуговування. Мета роботи – спроєктувати архітектуру програмного забезпечення під реалізацію мовою Go, призначену для відтворення поведінки систем масового обслуговування різних типів, без урахування виходу з ладу окремих каналів обслуговування, з використанням паралельних обчислень. У статті вирішуються наступні завдання: розглянути підстави для проєктування архітектури та зробити висновок про її доцільність; розробити вимоги до майбутнього програмного продукту задля більш ефективного використання ресурсів та чіткого визначення успішного завершення роботи; провести аналіз підходів до організації архітектури програмного забезпечення й прийняття обґрунтованого рішення щодо застосування одного з них; спроєктувати загальну схему алгоритму з урахуванням всіх вимог; визначити компоненти системи, що моделюється, та їхні взаємодії; побудувати процесні діаграми з урахуванням особливостей мови програмування Go; визначити спосіб та контракти взаємодії з програмним забезпеченням. Для досягнення поставлених цілей дослідження використовуватимуться такі методи: мова програмування Go, конкаренсі, архітектурні UML діаграми, C4 діаграми, процесні діаграми. Було здобуто такі результати: визначено вимоги до програмного забезпечення моделювання СМО; розглянуто поширені підходи до організації архітектури та проведено для них порівняльний аналіз; розроблено структуру майбутньої програми на необхідних рівнях абстракції; вперше запропоновано архітектуру програмного продукту для моделювання різноманітних систем масового обслуговування із застосуванням паралельних обчислень та підходу конкаренсі під реалізацію мовою програмування Go.

Ключові слова: архітектура програмного забезпечення, комп'ютерне моделювання, система масового обслуговування, мова програмування Go, конкаренсі, паралелізм.

Вступ. Першим, і безперечно базовим кроком при роботі над якісним програмним забезпеченням – є його архітектурне рішення. Адже, як дім не будують без креслень – програмний продукт не пишуть без архітектури. Саме архітектура системи визначає компоненти, що прийматимуть участь в обробці даних, їхню будову, а також спосіб взаємодії [1]. Вона являє собою чіткий рецепт, що буде однаково сприйнятий будь-яким інженером, і дозволить уникнути непорозумінь при реалізації задуманого. Відповідно, ми уникаємо залежності від конкретних особистостей, розблоковуючи потенціал до розширення команди розробки, а також нівелюємо ризики, пов'язані зі зміною ключових членів колективу інженерів.

Разом з тим, на початку роботи над новим продуктом у більшості випадків існує тільки часткове розуміння потреб та призначення майбутнього програмного забезпечення. Це пов'язано зі складністю проєктованих систем та незліченною кількістю сторонніх чинників, що впливають на них. У зв'язку з цим, розуміння реальних потреб виникає на етапі проєктування під впливом дослідження проблематики, а також правильної постанови запитань до бізнес доменів [2].

Не можна забувати й про подальший розвиток програмного забезпечення. Згодом дуже ймовірно постане потреба в одному або декількох одночасно підкласів масштабування:

- Збільшення навантаження й пропускної здатності окремих частин програми;
- Зміна або розширення користувацьких потреб з подальшою надбудовою функціоналу;
- Розширення команди або декомпозиція зон відповідальності з подальшою делегацією повноважень між кількома автономними командами розробки.

Кожен з наведених різновидів масштабування потребує завчасно закладеного підґрунтя, що надає можливість безшовної, поетапної розбудови програмного продукту, без необхідності глибокої переробки вже існуючого функціоналу [3].

Разом з тим, передбачити абсолютно всіх потенційних сценаріїв розвитку подій неможливо. Відповідно, пошук балансу між гнучкістю та складністю – є черговою задачею, що закладено у процес побудови архітектури. Зазвичай вирішення полягає у завчасному застосуванні абстракцій, що відокремлюватимуть конкретну реалізацію від функціоналу, що очікується.

Гарна ізоляція компонентів дозволяє підвищити надійність та стійкість програмного забезпечення до помилок виконання, а також знизити потенційний вплив змін, що вносяться в окремі частини програми, локалізуючи їх у рамках фіксованих абстракцій [1].

Додатковим приводом до того, щоб приділити достатньо уваги проєктуванню перед початком роботи – є ціна виправлення помилки. На етапі побудови архітектури задля виправлення недоліків проєктування, необхідно переписати документацію та внести зміни до діаграм і блок-схем, на відміну від кількості часу та зусиль, що вимагаються для модифікації вже існуючої програми. І чим більш пізня фаза розробки розглядається – тим складніше буде виправити недолік [1]. Отже, приділивши більше часу й зусиль до розробки архітектури програмного продукту, ми значно економимо час та ресурси, необхідні на внесення правок у майбутньому.

Окрім того, цілком можливо, що під час проєктування або незабаром, після закінчення процесу

побудови архітектури, проєкт буде визнаний недоцільним. Адже, знайдеться інший шлях до вирішення потреби, суттєво дешевший або швидший. Відповідно, задача архітектурного рішення полягає, зокрема, у загальній оцінці витрат, ризиків, а також в обґрунтуванні необхідності саме такого шляху до забезпечення потреб.

Роль маяку доповнює перелік задач, що покликана вирішувати архітектура програмного забезпечення. Доволі часто трапляється, що під час роботи над імплементацією проєкту, виникають нові іспити та необхідно підлаштовуватись під змінені потреби [2]. Архітектура системи забезпечує орієнтир, що має використовуватись для звірки відповідності чергового рішення загальному курсу й стратегії розвитку.

Мета дослідження. За визначенням архітектура програмного забезпечення зобов'язана задовольняти потребам чітко визначеної задачі. Саме тому є вкрай важливим приділити достатню увагу специфікації проблеми, що вирішується. У нас на меті стоїть побудова програмного забезпечення для моделювання поведінки систем масового обслуговування. При цьому є велика кількість різновидів СМО, що поділяються за наступними критеріями [4]:

- Наявність одного чи більшої кількості каналів обслуговування, що надають послугу;
- Наявність або відсутність черги очікування для новоприбулих заявок, на випадок, якщо всі наявні канали обслуговування зайняті;
- Кінцевість або необмежений розмір черги очікування;
- Політика пріоритизації вимог, що визначає, яка саме вимога з очікуючих буде приступати до обслуговування в разі звільнення каналу;
- Наявність обмежень часу перебування в системі або часу обслуговування.

У рамках даного дослідження нас цікавить у першу чергу моделювання багатоканальної системи з обмеженою чергою та відмовами новоприбулим вимогам, у разі відсутності місця для очікування. Але при цьому ми закладатимемо можливість розширення та підтримки всіх інших різновидів СМО у майбутньому. Кінцевою метою програмного продукту є універсальна програма, що надає можливість моделювати різноманітні процеси, порівнювати поведінку систем зі змінними режимами та параметрами [5]. Не останню роль гратиме швидкодія моделювання, яка має надати можливість отримувати швидше результат відтворення поведінки складних систем та оптимізувати процес завдяки результатам експериментів. Допоміжним підходом у досягненні мети – буде застосування паралельних обчислень. А залучення підходу конкаренсі – наблизить поведінку моделі до природнього досліджуваного процесу.

Постановка задачі. У даному розділі ми перелічимо ті чіткі вимоги, які ми встановлюємо як ключові для результуючого програмного забезпечення.

1. Можливість перед початком роботи програми визначити основні характеристики системи, що описуватимуть її поведінку, а саме – кількість каналів обслуговування, ємність черги. На даному етапі моделюються СМО з обмеженою чергою.
2. Можливість визначати максимальний час перебування заявки у системі, а також граничну тривалість обслуговування. У випадку перевищення даного часу обробка переривається та вимога покидає систему з помилкою.
3. Обслуговування полягає у виконанні функції, заданої перед початком роботи програми. Вона є єдиною для всіх вимог і використовує для розрахунків вхідні параметри, що визначаються заявкою. Сама функція може мати довільну логіку, але має очікувати вхідні параметри й повертати результат обробки або помилку.
4. Забезпечити підрахунок та збір інформації щодо кількості провалених задач задля подальшого аналізу ефективності системи.
5. Обслуговування черги заявок у послідовності їхнього надходження (FIFO).
6. Співпадіння послідовності повернення результатів виконання вимог із їхнім надходженням не гарантується системою.

Сформульовані таким чином вимоги вписуються у визначення систем масового обслуговування та мають надавати вичерпний функціонал для моделювання широкого спектру процесів [5].

Визначення типу архітектури. Першим значним роздоріжжям на шляху до побудови архітектурного рішення є відповідь на запитання про тип програмного забезпечення. Існує кілька підходів до того, як буде викликатись та виконуватись застосунок [6]. Умовно їх можна оцінити за наступними критеріями:

- Чи має програмне забезпечення свою незалежну область пам’яті та планувальник виконання.
- Чи є застосунок доступним неперервно, чи він запускається за вимогою на час обробки;
- Яким саме чином забезпечується взаємодія з функціоналом і передача параметрів.

За даними ознаками можна поділити підходи на кілька груп, що наведені у табл. 1.

Таблиця 1 – Класифікація видів програмного забезпечення

	Окремий рантайм	Доступність	Пряма взаємодія
Утиліта	+	-	-
Серверлес	+	-	-
Сервер (HTTP/RPC)	+	+	-
Пакет	-	+	+

Командна утиліта має свій рантайм, але при цьому її необхідно запускати кожен раз, коли з'являється необхідність взаємодії, що погіршує її доступність. Крім того, з недоліків маємо ускладнену комунікацію через передачу параметрів та результатів у вигляді параметрів командного рядку.

Серверлес – це технологія що пропонується представниками великих хмарних провайдерів. По суті являє собою гібрид командної утиліти та серверу. За такого підходу відповідальність за запуск програми та

комунікацію беруть на себе відповідні сервіси провайдера. З суттєвих недоліків такого підходу – велика кількість посередників, слабко типізовані контракти та нестабільність часу очікування через те, що функції, які певний час не викликаються, переводяться у гібернацію. І наступний раз необхідно чекати на їхній перезапуск. Проблему контактів можна вирішити із застосуванням додаткових сервісів, але це в значній мірі ускладнює систему [7].

Наступною групою є сервери. Вони працюють неперервно й незалежно від клієнтського застосунку, що спрощує раптову взаємодію та робить час відповіді прогнозованим. Однак, і в цьому випадку для використання у контрактах ми обмежені типами даних, що можуть приймати текстовий формат. У випадку з функцією такий підхід вимагатиме перетворення тексту запиту на програмний код та підключення його до виконання сервером. Такі дії значно збільшують складність та додають вразливостей системі.

У мові програмування Go найменший рівень абстракції, а також інкапсуляції забезпечується пакетом [8]. При цьому пакети є частиною рантайму загального застосунку та на них поширюються всі можливості по використанню системи типів мови програмування Go. Отже, ми можемо визначити тип параметра як функцію та застосувати дженерік типи для абстрагування від фактичних значень зі збереженням жорсткої типізації.

Кожен з перелічених підходів має свої переваги та недоліки. Однак, враховуючі вимогу щодо гнучкості функції, яка описуватиме логіку обслуговування, перед нами постає необхідність визначати саму функцію на боці клієнтського коду й передавати до функціоналу, що обслуговуватиме процес моделювання. Це можна реалізувати за допомогою кодогенерації. Однак за такого підходу ми значно ускладнюємо реалізацію. Отже, наша стратегія полягатиме в тому, щоб реалізувати найпростіший і найбільш природній для мови програмування Go варіант із забезпеченням ізоляції відповідальності. Це дозволить відокремити спосіб комунікації в окремі абстракції та зробити можливим перевикористання однієї логіки за потреби у багато різних способів. Ми приймаємо рішення про розробку архітектури програмного продукту як пакету з делегацією відповідальності за спосіб комунікації рівню взаємодії.

Вирішення проблеми. Для початку необхідно визначитись із загальними компонентами, що приймають участь у побудові алгоритму [9], а саме:

- Приймач заявок – по суті є публічним API пакетом. Задача полягає в прийомі вимог, загальній валідації та постанові до черги. У випадку переповнення черги – заявці буде відмовлено в обслуговуванні;
- Черга очікування – приймає та утримує вимогу до звільнення каналу обслуговування. Надає перевагу заявкам, що надійшли раніше;
- Канал обслуговування – безпосередньо обробник заявок. За раз може опрацьовувати не більше однієї вимоги. Перед початком виконання перевіряє чи не вичерпано час перебування заявки у системі. У випадку виникнення помилки при роботі або перевищення часу обслуговування має переривати виконання. Одразу після звільнення переходять у режим очікування чергової заявки;
- Пост-обробник результатів – відокремлює вдалі заявки від таких, якім було відмовлено в обслуговуванні. У подальшому реєструє причини провалів, а також може виконати інші допоміжні задачі, такі, як ведення обліку подій.
- Потік результатів – повертає користувачу API результат обробки заявок з додаванням додаткової інформації щодо причин відмови на випадок помилки під час обслуговування.

Такий розподіл компонентів повністю лягає на теорію масового обслуговування і забезпечує вичерпну функціональність щодо моделювання СМО [5]. Загальну схему компонентів та послідовність обробки заявки [9] зображено на рис. 1.

Наступним етапом необхідно підготувати цикл діаграм, що описуватимуть майбутню імплементацію з інженерної точки зору. Для цього скористаємось моделлю C4 – простим, прозорим і легким у використанні інструментом для розробників та архітекторів. Він покликаний стандартизувати поетапне проєктування складних систем, де ми починаємо

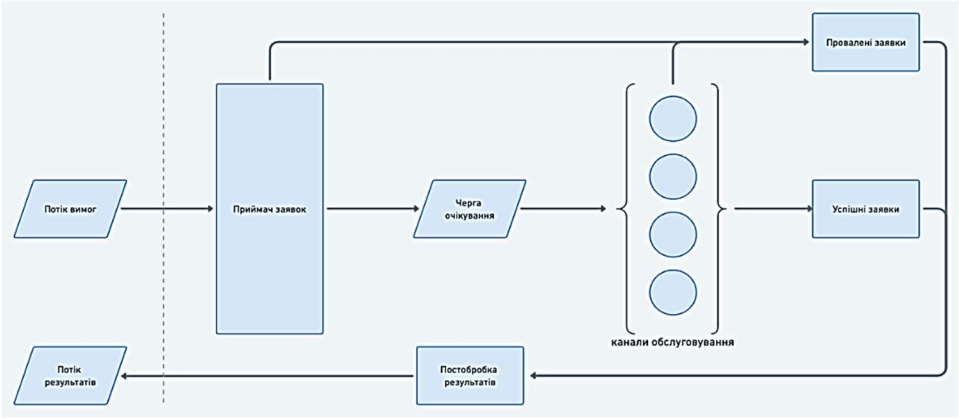

Рис. 1. Загальна схема алгоритму

занурення із загального вигляду системи та наближаємось з кожним наступним етапом ближче до реалізації. Така модель включає в себе всього 4 кроки.

Найвищій рівень абстракції – це С1. На ньому відображається контекстна діаграма із загальним зображенням системи. На рис. 2 можемо спостерігати дві сутності: користувача та кінцевий програмний продукт, що містить експеримент. Пакет, архітектуру якого ми закладаємо в рамках дослідження, на даному етапі не має значення оскільки він інтегрований всередину експерименту.

Рис. 2. Рівень моделі С1

На наступному рівні, також відомому як контейнерний, нас вже цікавить загальна будова системи без надмірних подробиць, але на рівні, необхідному для розуміння взаємодії глобальних складових. Дану діаграму зображено на рис. 3.

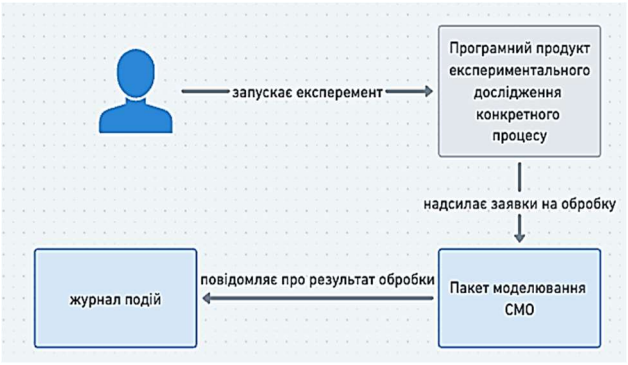

Рис. 3. Рівень моделі С2

Нарешті, час перейти до компонентної діаграми. Її роль в описі системи полягає в зображенні структурної будови досліджуваних компонентів. Нас найбільш цікавить саме пакет, архітектуру якого ми і будуємо. З рис. 4 випливає, що ключову роль системи відіграватиме канал обслуговування. Саме в цьому вузлі буде відбуватися обробка заявки, а також всі додаткові умови щодо часу перебування. Роль оркестрації та застосування конкаренсі має опрацьовувати окремий пакет [10]. Відокремлення реалізації паралелізму дозволить перевикористати даний функціонал для інших потреб, а також обмежити доступ користувачам.

Наступним рівнем абстракції буде діаграма коду, на ній будуть зображені подробиці реалізації специфічні для конкретної мови програмування Go. На рівні С4 найбільш цікавими для нас є дві зони відповідальності: симуляція СМО та організація паралельних обчислень шляхом застосування підходу конкаренсі [11].

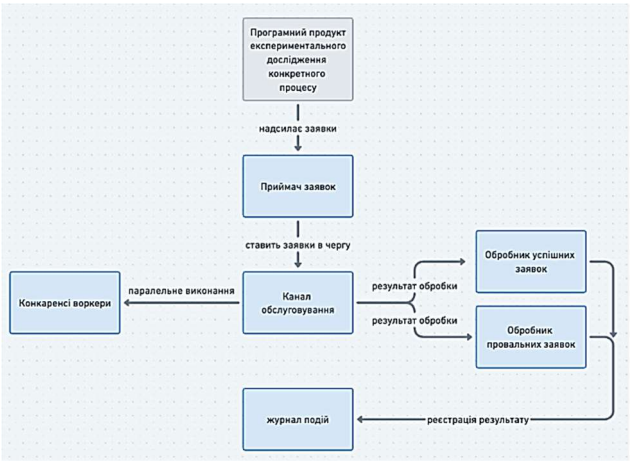

Рис. 4. Рівень моделі С3

Основна ідея полягає в чіткому розподілені обов'язків і побудова абстракцій на місцях перетину. Відповідно до визначених вимог, будемо будувати два окремих пакети, кожен з яких буде надавати публічне API, і при цьому ховати всі деталі реалізації. При цьому сама заявка має бути обгорнута опціями, які будуть додавати додаткові властивості процесу без прямого в нього втручання. Дана діаграма наведена на рис. 5.

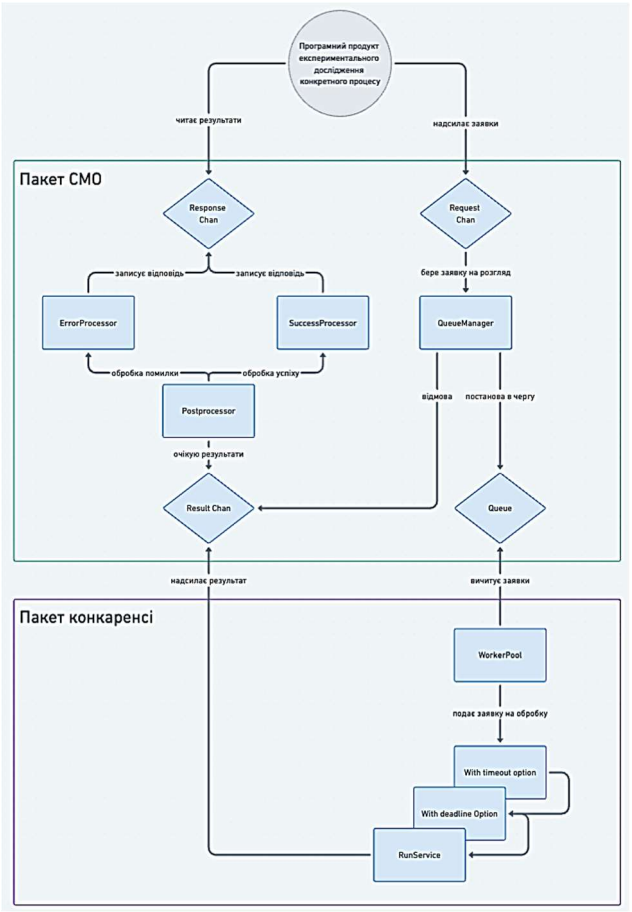

Рис. 5. Рівень моделі С4

У мові програмування Go асинхронна комунікація може бути забезпечена за рахунок вбудованого типу "канал" [8]. Ця структура даних надає можливість

організувати безпечний доступ до заявок, які очікують у буфері й будуть поступати одержувачам у послідовності, що співпадає з надходженням до черги. Оскільки запис у переповнений канал у мові Go неможливий – це дає нам зручну обставину для відмови. Максимальний час перебування в системі може бути обмежений із застосуванням контексту з тайм-аутом. Після вичерпання часового ліміту буде подано сигнал, який буде доступний в горутині, що обслуговує заявку, і буде надавати право переривання обробки. Кількість каналів обслуговування для системи буде визначатись кількістю горутин, які задіяні в пулі обробки. Самі по собі канали СМО можуть бути реалізовані за допомогою патерну паралельного програмування "worker" [10], реалізованого за допомогою легких програмних потоків – горутин. Такий підхід чудово гармонує з іншим дуетом підходів fan-in та fan-out [10][12]. Вони описують взаємодію між паралельними процесами та організацію асинхронної комунікації за умов різної кількості споживачів та генераторів повідомлень. Задля відокремленої та гнучкої пост-обробки ми додаємо кілька додаткових елементів, які будуть тимчасово розділяти успішні та провалені заявки. І після здійснення необхідних кроків передавати у канал відповіді заявнику.

Отже, дана система може бути реалізована мовою програмування Go, і в повній мірі задовільнити вимоги, сформовані у постановці задачі.

Гарною практикою при проєктуванні API пакетів у мові програмування Go – є зменшення кількості способів взаємодії, а також максимальне їх спрощення [10]. Метою такого підходу є унеможливлення хибного використання функціоналу та скорочення часу необхідного для засвоєння можливостей пакету. У нашому випадку кількість функцій, що доступні до використання клієнтам, може бути доведено до всього лиш однієї. Це буде операція по ініціалізації СМО із заданими параметрами й передачею каналу, що буде використовуватись для подачі заявок. У відповідь дана функція (конструктор) поверне змінну типу канал для читання результатів, і вся подальша взаємодія буде відбуватись за допомогою двох каналів. Як результат, замість кооперації через надання спільного доступу до певних ділянок пам'яті кільком паралельним процесам, ми перебудовуємо механізм на взаємодію через спілкування повідомленнями. Такий підхід було введено і детально описано в роботі Ентоні Хоара у рамках його роботи з "Communicating Sequential Processes", або скорочено CSP [13]. Для завершення роботи з СМО буде достатньо закриття каналу для надходження заявок, що призведе до лагідного згортання всіх процесів. Як і було зазначено у вимогах, всі характеристики системи мають задаватись через структуру конфігурації перед початком роботи та бути використані при ініціалізації процесу

Висновки. У ході дослідження було розглянуто підстави до дизайну архітектури програмного забезпечення, призначеного для моделювання систем масового обслуговування. Аргументи були достатньо вагомими для прийняття рішення про доцільність проєктування. Для отримання задовільних результатів було сформовано ряд вимог щодо майбутньої реалізації та оцінені їх пріоритети. З метою досягнення високої якості й зручності у використанні майбутньої програми, було розглянуто декілька можливих підходів до архітектури. З урахуванням особливостей мови програмування Go, було обрано шлях розробки застосунку у вигляді пакету. Для поставленої задачі було підготовлено загальну діаграму алгоритму опрацювання заявок, а також спроєктовано всі чотири рівні моделі C4. Була також врахована вимога щодо паралельного обслуговування заявок СМО з використанням підходів конкаренсі. Вся проєктна документація розроблена під імплементацію мовою Go з використанням вбудованого функціоналу. У ході дослідження вперше було запропоновано архітектуру програмного забезпечення, призначену для моделювання різноманітних систем масового обслуговування із застосуванням паралельних обчислень і підходів конкаренсі під імплементацію мовою програмування Go.

Список використаної літератури

1. Conery R., Hanselman S., Haack P., Guthrie S. *Microsoft Application Architecture Guide, 2nd Edition: Designing Applications on the .NET Platform*. Microsoft Press, 2009. 560 p.
2. Кірхар Н. В. Застосування технології архітектурного проектування програмного забезпечення. *Проблеми інформатизації та управління.* 2019. Т. 1, № 64. С. 50–56. DOI: https://doi.org/10.18372/2073-4751.61.14038
3. Форкун Ю., Мартинюк В., Яшина О. Метод розробки та проектування архітектурної складової програмного застосунку. *Вимірювальна та обчислювальна техніка в технологічних процесах.* 2023. № 4. С.87–93. DOI: https://doi.org/10.31891/2219-9365-2023-76-11
4. Гнеденко Б. В., Коваленко И. Н. Введение в теорию массового обслуживания. Москва: Наука, 1987. 336 с.
5. Литвинов А. Л. *Теорія систем масового обслуговування.* Харків: ХНУМГ ім. О. М. Бекетова, 2018. 141 с.
6. Морозов А., Вакалюк Т., Кубрак Ю., Зосімович Д. Аналіз факторів впливу на архітектури програмних систем. *Information Technology: Computer Science, Software Engineering and Cyber Security.* 2022. № 1, С. 44–52. DOI: https://doi.org/10.32782/IT/2022-1-7
7. System Architecture. URL: https://rubygarage.org/blog/monolith-soa-microservices-serverless (дата звернення: 03.03.2024).
8. Donovan A. A. A., Kernighan B. W. *The Go programming language*. New York: Addison-Wesley Professional, 2015. 400 p.
9. Лісняк А. О., Мильцев О. М., Мухін В. В., Чопорова О. В. *Архітектура та проектування програмного забезпечення: метод. рек-ції до лаб. зан. для здобувачів ступ. вищ. осв. бакалавра спец. 121 «Інженерія програмного забезпечення» осв.-проф. прогр. «Програмна інженерія»*. Запоріжжя: ЗНУ, 2022. 53 с. URL: https://dspace.znu.edu.ua/xmlui/handle/12345/11642 (дата звернення: 01.04.2024).
10. Castro Contreras M. *Go Design Patterns.* Birmingham: Packt, 2017. 402 p.
11. Hoare C. A. R. *Communicating sequential processes.* New Jersey: Prentice Hall, 1985. 235 p. DOI: https://doi.org/10.1145/359576.359585
12. Cox-Buday K. *Concurrency in Go: Tools and Techniques for Developers 1st Edition.* Sebastopol: O'reilly Media, 2017. 236 p.
13. Hoare C. A. R. *Communicating sequential processes. Communications of the ACM.* 1978. Vol. 21. № 8. P. 666–677. DOI: https://doi.org/10.1145/359576.359585.

References (transliterated)

1. Conery R., Hanselman S., Haack P., Guthrie S. *Microsoft Application Architecture Guide, 2nd Edition: Designing Applications on the .NET Platform*. Microsoft Press, 2009. 560 p.

2. Kirkhar N. V. Zastosuvannia tekhnolohii arkhitekturnoho proektuvannia prohramnoho zabezpechennia [Application of software architectural design technology]. *Problemy informatyzatsii ta upravlinnia* [Problems of informatization and management] 2019, vol. 1, no. 64, pp. 50–56. DOI: https://doi.org/10.18372/2073-4751.61.14038 (In Ukr.)
3. Forkun Yu., Martyniuk V., Yashyna O. Metod rozrobky ta proektuvannia arkhitekturnoi skladovoi prohramnoho zastosunku [The method of development and design of the architectural component of the software application]. *Vymiriuvalna ta obchysliuvalna tekhnika v tekhnolohichnykh protsesakh* [Measuring and computing equipment in technological processes]. 2023, no. 4, pp. 87–93. DOI: https://doi.org/10.31891/2219-9365-2023-76-11 (In Ukr.)
4. Gnedenko B. V., Kovalenko I. N. *Vvedenie v teoriyu massovogo obsluzhivaniya* [Introduction to Queuing Theory]. Moskva, Nauka, 1987. 336 p. (In Russ.)
5. Lytvynov A. L. *Teoriia system masovoho obsluhovuvannia* [Queuing Theory System]. Kharkiv, KhNUMH im. O. M. Beketova, 2018. 141 p. (In Ukr.)
6. Morozov A., Vakaliuk T., Kubrak Yu., Zosimovych D. Analiz faktoriv vplyvu na arkhitektury prohramnykh system [Analysis of influencing factors on the architecture of software systems]. *Information Technology: Computer Science, Software Engineering and Cyber Security.* 2022, no. 1, pp. 44–52. DOI: https://doi.org/10.32782/IT/2022-1-7 (In Ukr.)
7. System Architecture. URL: https://rubygarage.org/blog/monolith-soa-microservices-serverless (accessed 03.03.2024).
8. Donovan A. A. A., Kernighan B. W. *The Go programming language*. New York, Addison-Wesley Professional, 2015. 400 p.
9. Lisniak A. O., Myltsev O. M., Mukhin V. V., Choporova O. V. *Arkhitektura ta proektuvannia prohramnoho zabezpechennia: metod. rek-tsii do lab. zan. dlia zdobuvachiv stup. vyshch. osv. bakalavra spets. 121 "Inzheneriia prohramnoho zabezpechennia" osv.-prof. prohr."Prohramna inzheneriia"* [Software architecture and design: methodical recommendations for laboratory classes for higher education bachelor degree holders, specialty 121 "Software engineering" of the educational and professional program "Software engineering"]. Zaporizhzhia: ZNU, 2022. 53 p. URL: https://dspace.znu.edu.ua/xmlui/handle/12345/11642 (accessed 01.04.2024). (In Ukr.)
10. Castro Contreras M. *Go Design Patterns*. Birmingham, Packt, 2017. 402 p.
11. Hoare C. A. R. *Communicating sequential processes*. New Jersey, Prentice Hall, 1985. 235 p. DOI: https://doi.org/10.1145/359576.359585
12. Cox-Buday K. *Concurrency in Go: Tools and Techniques for Developers 1st Edition*. Sebastopol, O'reilly Media, 2017. 236 p.
13. Hoare C. A. R. *Communicating sequential processes. Communications of the ACM*. 1978, vol. 21, no. 8, pp. 666–677. DOI: https://doi.org/10.1145/359576.359585.

Надійшла (received) 08.05.2024

UDC 004.9

D. I. GOLDINER, National University of Radioelectronics, Kharkiv, Ukraine, Graduate student, Kharkiv, Ukraine; e-mail: denys.holdiner@nure.ua; ORCID: https://orcid.org/0000-0002-1456-1867

SOFTWARE ARCHITECTURE SYSTEM DESIGN FOR THE MASS SERVICE SYSTEMS MODELING TO BE IMPLEMENTED IN GO PROGRAMMING LANGUAGE

The subject of the article is the methods and approaches to organizing the architecture of software implementation designed for modeling the behavior of mass service systems. The goal of the work is to design a software architecture for implementation in Go language, intended to replicate the behavior of various types of mass service systems, without considering the failure of individual service channels, using parallel computing. The article addresses the following tasks: consider the basis for designing the architecture and conclude its appropriateness; develop requirements for the future software product for more effective resource use and clear definition of successful completion; analyze the approaches to organizing software architecture and make a justified decision on the application of one of them; design a general algorithm scheme taking into account all requirements; identify the components of the modeled system and their interactions; build process diagrams considering the specifics of the Go programming language; define the method and contracts of interaction with the software. The research will utilize the following methods: Go programming language, concurrency, architectural UML diagrams, C4 diagrams, process diagrams. The following results were obtained: the requirements for the software for modeling mass service operations (SMO) were defined; common approaches to organizing architecture were considered and a comparative analysis was conducted; the structure of the future program was developed at the necessary levels of abstraction; for the first time, an architecture of the software product for modeling various mass service systems using parallel computing and the concurrency approach under the implementation in the Go programming language was proposed.

Keywords: software architecture, computer modeling, mass service system, Go programming language, concurrency, parallelism.

Повне ім'я автора / Author's full name

Автор 1 / Author 1: Гольдінер Денис Ігорович / Goldiner Denys Ihorovych