3. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
4. en.wikipedia.org/wiki/Batch_normalization
infoworld.com/article/3269878/what-is-an-api-application-programming-interfaces-explained.html

УДК 004.4

ГАВРИЛЯК А.В.,
ЛІЩУК К.І.

АРХІТЕКТУРНІ ШАБЛОНИ ТА СТИЛІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Як і будь-яка інша складна структура, програмне забезпечення має будуватися на міцному фундаменті. Неправильне визначення ключових сценаріїв, неправильне проектування загальних питань або нездатність виявити довгострокові наслідки основних рішень можуть поставити під загрозу всю програму. Сучасні інструменти та платформи спрощують завдання щодо створення додатків, але не усувають потреби у ретельному їх проектуванні на підставі конкретних сценаріїв та вимог. Неправильно вироблена архітектура обумовлює нестабільність програмного забезпечення, неможливість підтримувати існуючі чи майбутні бізнес-вимоги, складності при розгортанні чи управлінні серед виробничої експлуатації.

Проектування систем має здійснюватися з урахуванням потреб користувача, системи (IT-інфраструктури) та бізнес-цілей. Для кожної з цих складових визначаються ключові сценарії та виділяються важливі параметри якості (наприклад, надійність або масштабованість), а також основні області задоволеності та незадоволеності. По можливості необхідно виробити та врахувати показники успішності у кожній із цих областей.

Ключові слова: архітектура, архітектурні шаблони, архітектурні стилі, програмне забезпечення, побудова програмного забезпечення

Like any other complex structure, software must be built on a solid foundation. Misidentifying key scenarios, misdesigning common issues, or failing to identify the long-term consequences of major decisions can jeopardize the entire application. Modern tools and platforms make it easy to build applications, but they do not obviate the need for careful design based on specific scenarios and requirements. An improperly designed architecture results in software instability, inability to support existing or future business requirements, and difficulty in deploying or managing in a production environment.

Systems design should be based on user needs, system (IT infrastructure) and business goals. For each of these dimensions, key scenarios are identified and important quality parameters (such as reliability or scalability) are highlighted, as well as major areas of satisfaction and dissatisfaction. Where possible, indicators of success in each of these areas should be developed and recorded.

Keywords: architecture, architectural patterns, architectural styles, software, software construction

1. Вступ

Як і будь-яка інша складна структура, програмне забезпечення має будуватися на міцному фундаменті. Неправильне визначення ключових сценаріїв, неправильне проектування загальних питань або нездатність виявити довгострокові наслідки основних рішень можуть поставити під загрозу всю програму. Сучасні інструменти та платформи спрощують завдання щодо створення додатків, але не усувають потреби у ретельному їх проектуванні на підставі конкретних сценаріїв та вимог. Неправильно вироблена архітектура обумовлює нестабільність програмного забезпечення, неможливість підтримувати існуючі чи майбутні бізнес-вимоги, складності при розгортанні чи управлінні серед виробничої експлуатації.

Проектування систем має здійснюватися з урахуванням потреб користувача, системи (IT-інфраструктури) та бізнес-цілей. Для кожної з цих складових визначаються ключові сценарії та виділяються важливі параметри якості (наприклад, надійність або масштабованість), а також основні області задоволеності та незадоволеності. По можливості необхідно виробити та врахувати показники успішності у кожній із цих областей.

В даних тезах описуються узагальнені шаблони та принципи, які застосовуються при проектуванні додатків сьогодні. Часто їх називають архітектурними стилями чи парадигмами. До них відносять такі шаблони як клієнт/сервер, багаторівнева архітектура, компонентна архітектура, архітектура, що базується на шині повідомлень, та сервісно-орієнтована архітектура (service-oriented architecture, SOA). Для кожного стилю пропонується огляд, основні принципи, основні переваги та відомості, які допоможуть вибрати архітектурні стилі [1].

2. Основні архітектурні стилі

Клієнт/сервер – система поділяється на дві програми, де клієнт виконує запити на сервер. У багатьох випадках у ролі сервера виступає база даних, а логіка програми представлена процедурними зберігання.

Компонентна архітектура - дизайн програми розкладається на функціональні або логічні компоненти з можливістю повторного використання, що надають ретельно опрацьовані інтерфейси зв'язку.

Багатошарова архітектура - функціональні області програми поділяються на багатошарові групи (рівні).

Шина повідомлень - архітектурний стиль, що наказує використання програмної системи, яка може приймати та відправляти повідомлення по одному або більше каналах зв'язку, так що програми отримують можливість взаємодіяти, не маючи конкретних відомостей один про одного.

N-рівнева/3-рівнева - функціональність виділяється в окремі сегменти, багато в чому аналогічно багатошаровому стилю, але в даному випадку сегменти фізично розташовуються на різних комп'ютерах.

Об'єктно-орієнтована - парадигма проектування, заснована на розподілі відповідальності програми або системи між окремими багаторазово використовуваними та самостійними об'єктами, що містять дані та поведінку.

Сервісно-орієнтована архітектура - описує програми, що надають та споживають функціональність у вигляді сервісів за допомогою контрактів та повідомлень.

3. Архітектура клієнт/сервер

Клієнт/серверна архітектура описує розподілені системи, що складаються з окремих клієнта та сервера та мережі, що їх з'єднує. Найпростіша форма системи клієнт/сервер, яка називається 2-рівневою архітектурою – це серверний додаток, до якого безпосередньо звертаються багато клієнтів [1].

На сьогодні прикладами архітектурного стилю клієнт/сервер можуть бути Веб-додатки, що виконуються в Інтернеті або внутрішніх мережах організацій; настільні програми для операційної системи Microsoft Windows®, що виконують доступ до мережевих сервісів даних; програми, що виконують доступ до віддалених сховищ даних (такі як програми читання електронної пошти, FTP-клієнти та засоби доступу до баз даних); інструменти та утиліти для роботи з віддаленими системами (такі як засоби управління системою та засоби моніторингу мережі).

До інших різновидів стилю клієнт/сервер відносяться [2]:

Системи клієнт-черга-клієнт. Цей підхід дозволяє клієнтам обмінюватись даними з іншими клієнтами через чергу на сервері. Клієнти можуть читати дані з та відправляти дані на сервер, який виступає в ролі простої черги для зберігання даних. Завдяки цьому клієнти можуть розподіляти та синхронізувати файли та відомості. Іноді таку архітектуру називають пасивною чергою.

Одноранговi (Peer-to-Peer, P2P) програми. Створений на базі клієнт-черга-клієнт, стиль P2P дозволяє клієнту та серверу обмінюватися ролями з метою розподілу та синхронізації файлів та даних між безліччю клієнтів. Ця схема розширює стиль клієнт/сервер, додаючи множинні відповіді на запити, дані, що спільно використовуються, виявлення ресурсів і стійкість при видаленні учасників мережі.

Основні переваги архітектурного стилю клієнт/сервер:

Велика безпека. Всі дані зберігаються на сервері, який зазвичай забезпечує більший контроль безпеки ніж клієнтські комп'ютери.

Централізований доступ до даних. Оскільки дані зберігаються лише на сервері, адміністрування доступу до даних набагато простіше, ніж у будь-яких інших архітектурних стилях.

Простота обслуговування. Ролі та відповідальність обчислювальної системи розподілені між кількома серверами, які спілкуються один з одним по мережі. Завдяки цьому клієнт гарантовано залишається необізнаним і не схильним до впливу подій, що відбуваються з сервером (ремонт, оновлення або переміщення).

Тим не менш, традиційна 2-рівнева архітектура клієнт/сервер має безліч недоліків, включаючи тенденцію тісного зв'язування даних та бізнес-логіки програми на сервері, що може мати негативний вплив на розширюваність та масштабованість системи, і залежність від центрального сервера, що негативно позначається на надійність системи. Для вирішення цих проблем архітектурний стиль клієнт/сервер був розвинений у більш універсальний 3-рівневий (або N-рівневий), що описується нижче, у якому усунуті деякі недоліки, властиві 2-рівневій архітектурі клієнт/сервер, та забезпечують додаткові переваги.

4. Компонентна архітектура

Компонентна архітектура визначає підхід до проектування та розробки систем з використанням методів проектування програмного забезпечення. Основна увага в цьому випадку приділяється розкладанню дизайну на окремі функціональні або логічні компоненти, що надають чітко визначені інтерфейси, що містять методи, події та властивості.

Основний принцип компонентного стилю – застосування компонентів, що мають такі якості:

Придатність для повторного використання. Як правило, компоненти проектуються із забезпеченням можливості їх повторного використання у різних сценаріях різних додатків. Однак деякі компоненти створюються спеціально для конкретного завдання.

Замінюваність. Компоненти можуть легко замінюватися іншими подібними компонентами.

Незалежність від контексту. Компоненти проектуються для роботи в різних середовищах та контекстах. Спеціальні відомості, такі як дані про стан, повинні не включатися або вилучатися компонентом, а передаватися до нього.

Розширюваність. Компонент може розширювати існуючі компоненти задля забезпечення нової поведінки.

Інкапсуляція. Компоненти надають інтерфейси, що дозволяють стороні використовувати їх функціональність, не розкриваючи при цьому деталі внутрішніх процесів або внутрішні змінні або стан.

Незалежність. Компоненти проектуються із мінімальними залежностями від інших компонентів. Таким чином, компоненти можуть бути розгорнуті в будь-якому відповідному середовищі без впливу на інші компоненти або системи.

Основні переваги компонентного архітектурного стилю:

Простота розгортання. Існуючі версії компонентів можуть замінюватись новими сумісними версіями, не впливаючи на інші компоненти або систему в цілому.

Найменша вартість. Використання компонентів сторонніх виробників дозволяє розподіляти витрати на розробку та обслуговування.

Простота розробки. Для забезпечення заданої функціональності компоненти реалізують широко відомі інтерфейси, що дозволяє розробляти без впливу інші частини системи

Можливість повторного використання. Застосування компонентів, що багаторазово використовуються, означає можливість розподілу витрат на розробку та обслуговування між декількома додатками або системами.

Спрощення з технічного погляду. Компоненти полегшують систему через використання контейнера компонентів та його сервісів. Як приклади сервісів, що надаються контейнером, можна навести активацію компонентів, управління життєвим

циклом, організацію черги викликів методів, обробку подій та транзакції.

5. Багатошарова архітектура

Багатошарова архітектура забезпечує угруповання пов'язаної функціональності програми в різних шарах, що вибудовуються вертикально, поверх один одного. Шари слабко пов'язані, і з-поміж них здійснюється явний обмін даними. Правильне поділ програми на шари допомагає підтримувати суворий поділ функціональності, що у свою чергу забезпечує гнучкість, а також зручність і простоту обслуговування [1].

Розглянемо загальні принципи проектування з використанням багатошарової архітектури[2]:

Абстракція. Багатошарова архітектура представляє систему як єдине ціле, забезпечуючи при цьому достатньо деталей для розуміння ролей та відповідальності окремих верств та відносин між ними.

Інкапсуляція. Під час проектування немає необхідності робити будь-які припущення про типи даних, методи та властивості або реалізації, оскільки всі ці деталі приховані в рамках шару.

Чітко певні функціональні верстви. Розділення функціональності між шарами дуже чітке. Верхні шари, такі як шар подання, надсилають команди нижнім шарам, таким як бізнес-шар і шар даних, і можуть реагувати на події, що виникають у цих шарах, забезпечуючи можливість передачі даних між шарами вгору та вниз.

Висока зв'язність. Чітко визначені межі відповідальності для кожного шару та гарантоване включення до шару лише функціональності, безпосередньо пов'язаної з його завданнями, допоможе забезпечити максимальну зв'язність у межах шару.

Можливість повторного використання. Відсутність залежностей між нижніми та верхніми шарами забезпечує потенційну можливість їх повторного використання в інших сценаріях.

Слабке зв'язування. Для забезпечення слабкого зв'язування між шарами зв'язок між ними ґрунтується на абстракції та подіях.

Прикладами багатошарових додатків можуть бути бізнес-додатки (line-of-business, LOB), такі як системи бухгалтерського обліку та управління замовниками; Веб-додатки та Веб-сайти підприємств; настільні або смарт-клієнти підприємств із централізованими серверами додатків для розміщення бізнес-логіки.

Основними перевагами багатошарового архітектурного стилю є:

Абстракція. Рівні забезпечують можливість внесення змін на абстрактному рівні. Використовуваний рівень абстракції кожного шару може бути збільшений або зменшений.

Ізоляція. Оновлення технологій можуть бути ізольовані в окремих шарах, що допоможе скоротити ризик та мінімізувати вплив на всю систему.

Керованість. Поділ основних функцій допомагає ідентифікувати залежності та організувати код у секції, що підвищує керованість.

Продуктивність. Розподіл шарів за декількома фізичними рівнями може покращити масштабованість, відмовостійкість і продуктивність.

Можливість повторного використання. Ролі підвищують можливість повторного використання. Наприклад, MVC Контролер часто може використовуватися повторно з іншими сумісними Уявленнями для забезпечення характерного для ролі або настроєного для користувача подання одних і тих же даних та функціональності.

Тестованість. Поліпшення тестованості є результатом наявності певних інтерфейсів шарів, а також можливості перемикання між різними реалізаціями інтерфейсів шарів. Шаблони Separated Presentation дозволяють створювати під час тестування фіктивні об'єкти, що імітують поведінку реальних об'єктів, таких як Модель, Контролер або Подання.

6. Архітектура, заснована на шині повідомлень

Заснована на шині повідомлень архітектура описує принцип використання програмної системи, яка може приймати та надсилати повідомлення по одному або більше каналах зв'язку, забезпечуючи таким чином додаткам можливість взаємодії без необхідності знання конкретних деталей один про одного [1]. Це стиль проектування, в якому взаємодії між програмами здійснюються шляхом передачі (зазвичай

асинхронної) повідомлень через загальну шину. У типових реалізаціях архітектури, заснованої на шині повідомлень, використовується маршрутизатор повідомлень, або шаблон Publish/Subscribe (Публікація/Підписка) і система обміну повідомленнями, така як Message Queuing (Черга повідомлень) [3].

До основних переваг архітектури, заснованої на шині повідомлень, належать [4]:

Розширюваність. Можливість додавати або видаляти програми з шини без впливу на наявні програми.

Невисока складність. Програми спрощуються, тому що кожному з них необхідно знати лише, як обмінюватися даними з шиною.

Гнучкість. Приведення набору додатків, що складають складний процес, або схем зв'язку між додатками у відповідність бізнес-вимогам, що змінюються, або вимогам користувача просто шляхом внесення змін до конфігурації або параметрів, що керують маршрутизацією.

Слабке зв'язування. Крім наданого додатком інтерфейсу для зв'язку з шиною повідомлень, немає жодних інших залежностей із самим додатком, що забезпечує можливість зміни, оновлення та заміни його іншим додатком, що надає такий самий інтерфейс.

Масштабованість. Можливість підключення до шини множини екземплярів однієї програми для забезпечення одночасної обробки множини запитів.

Простота програми. Незважаючи на те, що реалізація шини повідомлень ускладнює інфраструктуру, кожному додатку доводиться підтримувати лише одне підключення до шини повідомлень, а не безліч підключень до інших програм.

7. N-рівнева / 3-рівнева архітектура

N-рівнева та 3-рівнева архітектура є стилями розгортання, що описують поділ функціональності на сегменти, багато в чому аналогічно багатошаровій архітектурі, але в даному випадку ці сегменти можуть фізично розміщуватися на різних комп'ютерах, їх називають рівнями. Дані архітектурні стилі було створено з урахуванням компонентно-ориентированного підходу і, зазвичай, зв'язку використовують методи платформи, а чи не повідомлення.

N-рівнева архітектура зазвичай має щонайменше трьох окремих логічних частин, кожна з яких фізично розміщується різних серверах. Кожна частина відповідає за певну функціональність. При використанні багатошарового підходу шар розгортається на рівні, якщо функціональність, що надається цим шаром, використовується більш ніж одним сервісом або додатком рівня.

Прикладом N-рівневого/3-рівневого архітектурного стилю може бути типова фінансова Веб-додаток з високими вимогами до безпеки. Бізнес-шар у цьому випадку має бути розгорнутий за міжмережевим екраном, через що доводиться розгортати шар представлення на окремому сервері у прикордонній мережі. Інший приклад – типовий насичений клієнт, у якому шар представлення розгорнуто на клієнтських комп'ютерах, а бізнес-шар та шар доступу до даних розгорнуті одному чи більше серверних рівнях.

Основними перевагами N-рівневого/3-рівневого архітектурного стилю є:

Зручність підтримки. Рівні не залежать один від одного, що дозволяє виконувати оновлення або зміни, не впливаючи на програму в цілому.

Масштабованість. Рівні організовуються на підставі розгортання шарів, тому масштабувати програму досить просто.

Гнучкість. Управління та масштабування кожного рівня може виконуватись незалежно, що забезпечує підвищення гнучкості.

Доступність. Програми можуть використовувати модульну архітектуру, яка дозволяє використовувати в системі компоненти, що легко масштабуються, що підвищує доступність.

8. Об'єктно-орієнтована архітектура

Об'єктно-орієнтована архітектура – це парадигма проектування, заснована на поділі відповідальності додатку чи системи на самостійні придатні повторного використання об'єкти, кожен із яких містить дані і поведінка, які стосуються цього об'єкту. При об'єктно-орієнтованому проектуванні система розглядається не як набір підпрограм та процедурних команд, а як набори об'єктів,

що взаємодіють. Об'єкти відокремлені, незалежні та слабко пов'язані; обмін даними між ними відбувається через інтерфейси шляхом виклику методів та властивостей інших об'єктів та відправки/приймання повідомлень [1]. Основними принципами об'єктно-орієнтованого архітектурного стилю є:

Абстракція. Дозволяє перетворити складну операцію на узагальнення, що зберігає основні характеристики операції. Наприклад, абстрактний інтерфейс може бути широко відомим описом, який підтримує операції доступу до даних через використання простих методів, таких як Get (Отримати) та Update (Оновити). Інша форма абстракції – метадані, використовувані задля забезпечення зіставлення двох форматів структурованих даних.

Композиція. Об'єкти можуть бути утворені іншими об'єктами та за бажанням можуть приховувати ці внутрішні об'єкти від інших класів або надавати їх як прості інтерфейси.

Спадкування. Об'єкти можуть успадковуватись від інших об'єктів і використовувати функціональність базового об'єкта або перевизначати її для реалізації нової поведінки. Більше того, успадкування спрощує обслуговування та оновлення, оскільки зміни, які вносяться до базового об'єкта, автоматично поширюються на всі об'єкти, що успадковуються від нього.

Інкапсуляція. Об'єкти надають функціональність лише через методи, властивості та події та приховують внутрішні деталі, такі як стан та змінні, від інших об'єктів. Це спрощує оновлення або заміну об'єктів та дозволяє виконувати ці операції без впливу на інші об'єкти та код, потрібно лише забезпечити сумісні інтерфейси.

Поліморфізм. Дозволяє перевизначати поведінку базового типу, що підтримує операції у додатку, шляхом реалізації нових типів, які взаємозамінні для існуючого об'єкта.

Відділення. Об'єкти можуть бути відокремлені від споживача шляхом визначення абстрактного інтерфейсу, реалізованого об'єктом та зрозумілого споживача. Це дозволяє забезпечувати альтернативні реалізації, не впливаючи на споживачів інтерфейсу.

9. Сервісно-орієнтована архітектура

Сервісно-орієнтована архітектура (Service-oriented architecture, SOA) забезпечує можливість надавати функціональність програми у вигляді набору сервісів та створювати програми, що використовують програмні сервіси. Сервіси слабо пов'язані, тому що використовують засновані на стандартах інтерфейси, які можуть бути викликані, опубліковані та виявлені. Основне завдання сервісів у SOA – надання схеми та взаємодії з додатком за допомогою повідомлень через інтерфейси, областю дії яких є програма, а не компонент чи об'єкт [1].

Основними принципами архітектурного стилю SOA є:

Сервіси автономні. Обслуговування, розробка, розгортання та контроль версій кожного сервісу відбувається незалежно від інших.

Сервіси можуть бути розподілені. Сервіси можуть розміщуватись у будь-якому місці мережі, локально або віддалено, якщо мережа підтримує необхідні протоколи зв'язку.

Сервіси слабо пов'язані. Кожен сервіс зовсім не залежить від інших і може бути замінений або оновлений без впливу на програми, що його використовують, за умови надання сумісного інтерфейсу.

Сервіси спільно використовують схему та контракт, але не клас. При обміні даними послуги спільно застосовують контракти і схеми, але з внутрішні класи.

Сумісність ґрунтується на політиці. Політика в цьому випадку означає опис характеристик, таких як транспорт, протокол та безпека.

Основними перевагами SOA-архітектури є:

Погодження предметних галузей. Повторне використання спільних сервісів зі стандартними інтерфейсами розширює технологічні та бізнес-можливості, а також скорочує вартість.

Абстракція. Сервіси є автономними, доступ до них здійснюється за формальним контрактом, що забезпечує слабке зв'язування та абстракцію.

Можливість виявлення. Сервіси можуть надавати описи, що дозволяє іншим програмам та сервісам виявляти їх та автоматично визначати інтерфейс.

Можливість взаємодії. Оскільки протоколи та формати даних ґрунтуються на галузевих стандартах, постачальник та споживач сервісу можуть створюватися та розгортатися на різних платформах.

Раціоналізація. Сервіси забезпечують певну функціональність, усуваючи необхідність її дублювання у додатках.

Висновки

Було досліджено узагальнені шаблони та принципи, які застосовуються при проектуванні додатків. До них відносять такі шаблони як клієнт/сервер, багаторівнева архітектура, компонентна архітектура, архітектура, що базується на шині повідомлень, та сервісно-орієнтована архітектура (service-oriented architecture, SOA). Для кожного стилю написаний огляд, основні принципи, основні переваги та відомості.

Список літератури

1 Руководство Microsoft © по проектированию архитектуры приложений – 2009р, https://studfile.net/preview/6413815/page:11/.
2 «Layered Versus Client-Server» (Сравнение многоуровневой архитектуры с архитектурой клиент-сервер) – 2009р, https://docs.microsoft.com/en-us/previous-versions/bb421529(v=msdn.10)?redirectedfrom=MSDN.
3 «Microsoft Enterprise Service Bus (ESB) Guidance» (Руководство по Microsoft Enterprise Service Bus (ESB)), http://www.microsoft.com/biztalk/solutions/soa/esb.mspx.
4 «Message Bus» (Шина сообщений), http://msdn.microsoft.com/enus/library/ms978583.aspx

УДК 004.627

ПОРТЯНИЙ І.С.,
ОЛІЙНИК Ю.О.,
ПОСПЄЛОВА К.І.

КОДУВАННЯ РАСТРОВИХ ЗОБРАЖЕНЬ НА ОСНОВІ ПОДІБНОСТІ ФРАГМЕНТІВ

Дана робота присвячена створенню методу кодування зображень на основі визначення подібності фрагментів шляхом використання нейронних мереж для виділення ознак фрагментів, алгоритмів машинного навчання для пошуку подібних фрагментів. У роботі використано згорткові нейронні мережі та класифікатор KNN (k-Nearest Neighbors) для кодування зображень. Проведено порівняння розміру закодованого зображення з вхідним.

КЛЮЧОВІ СЛОВА: КОДУВАННЯ ЗОБРАЖЕНЬ, ЗГОРТКОВІ НЕЙРОННІ МЕРЕЖІ, СТИСНЕННЯ ЗОБРАЖЕНЬ, ДЕКОДУВАННЯ ЗОБРАЖЕНЬ

This paper reviews the method of image encoding based on determining the similarity of fragments by using neural networks to extract features of fragments, machine learning algorithms to search for similar fragments. Used convolutional neural networks and KNN (k-Nearest Neighbors) classifier for image encoding. Compare the size of the encoded image with the input.

KEYWORDS: IMAGE ENCODING, CONVOLUTIONAL NEURAL NETWORKS, IMAGE COMPRESSION, IMAGE DECODING

1. Вступ

Сучасний світ постійно змінюється, у наш час відбувається стрімкий розвиток технологій та застосування їх у будь-яких сферах життя. За останні роки значних успіхів було досягнуто в сфері комп’ютерного зору та, відповідно, нейронних мереж, що працюють з зображеннями. Розвиток цих сфер розширює можливості для роботи з картинками.

Слід зазначити, що одним з найбільш важливих і визначальних аспектів як для зберігання, так і для передачі є стиснення вихідної інформації. Графічні дані займають достатньо велику кількість дискового