**Лабораторна робота №8-1.**  **Архітектура програмного забезпечення. Побудова функціональної, структурної та укрупненої схеми архітектури ПС**

***Мета:*** отримати практичні навички в обґрунтованому виборі компонентів архітектури програмного забезпечення для реалізації завдання та його нефункціональних властивостей.

**КОНТРОЛЬНІ ПИТАННЯ:**

1. Що таке архітектура ПЗ?

2. Що таке описова та розпорядча архітектура ПЗ?

3. Що таке архітектурний дрейф та ерозія?

4. Що таке відновлення архітектури?

5. Що таке компонент ПЗ?

6. Що таке з’єднувач ПЗ?

7. Що таке архітектурна конфігурація або топологія?

8. Що таке нефункціональна властивість ПЗ?

9. Що таке ефективність ПЗ?

10. Що таке складність ПЗ?

11. Що таке надійність ПЗ?

12. Що таке адаптованість ПЗ?

***Завдання:***

1. Опрацювати теоретичний матеріал до лабораторної роботи. Зосередити увагу на основних поняттях у галузі архітектури ПЗ.

2. На основі визначених функціональних та нефункціональних вимог, які необхідно реалізовувати в програмній системі побудувати функціональну, структурну і укрупнену схему архітектури ПС. Тема визначена на попередніх лабораторних (обрана власна тема).

3. Обрати компоненти та з’єднувачі, які будуть використовуватися та обґрунтувати їх вибір.

4. Зробити:

* Опис застосування, що розробляється з точки зору користувача.
* Опис основних функціональних та нефункціональних вимог, використовуючи результати попередніх лабораторних робіт.
* Функціональна, структурна і укрупнена схема архітектури ПС.
* Опис компонентів, які будуть використовуватися: сервіси, які забезпечують, основні архітектурні рішення. Обґрунтувати їх вибір.

По закінченню лабораторну роботу потрібно здати на перевірку викладачеві, надіславши електронною поштою на адресу [**t.i.lumpova@gmail.com**](mailto:t.i.lumpova@gmail.com). Якщо викладач знаходить помилки чи неточності, він може повернути роботу на доопрацювання.

Файл з роботою повинен мати назву в такому форматі:

**ОРІ <Номер групи><Номер лекції / лабораторної> [літера позначення типу роботи L – лекція, R – лабораторна]<Прізвище англійською>**.. Наприклад, **ОРІ4101R**buts.doc.

Не копіюйте фрагментів з різних інформаційних джерел, подумайте і викладіть свою точку зору. При наявності робіт-"близнюків" відповідь буде зараховуватися першому за часом надсилання.

Тему в заголовку листа записати

**ОРІ <Номер групи>-ЛР<Номер лабораторної>-<Прізвище >**

**Строк виконання цієї роботи ІПЗ-41 – 07.10.2024**

**ІПЗ-42 – 07.10.2024**

Всі запитання, що виникнуть, надсилайте на електронну адресу викладача, Тему в заголовку листа записати

**ОРІ <Номер групи>-Запитання-<Прізвище >**.

**Методичні рекомендації**

***Архітектура програмного забезпечення (ПЗ)*** – це множина основних проектних рішень про ПЗ. Архітектура ПЗ є планом розробки майбутнього програмного рішення, а також основою для подальшого життєвого циклу ПЗ. Проектні рішення охоплюють всі аспекти розроблюваного ПЗ, такі як структуру, поведінку, взаємодію з іншим ПЗ та нефункціональні властивості. «Основний» означає ступінь важливості, який надає проектному рішенню статус архітектурного. Тобто не всі проектні рішення є архітектурними, саме тому вони не впливають на архітектуру. Проте важливість проектних рішень у великій мірі залежить від цілей, які переслідуються при розробці ПЗ. Аспект тимчасовості архітектури означає, що у будь який момент часу ПЗ має тільки одну архітектуру і вона буде змінюватися з часом.

***Розпорядча системна архітектура складається***з проектних рішень, що були прийняті перед конструюванням ПЗ. Також її можна назвати продуманою чи призначеною архітектурою.

***Описова архітектура ПЗ визначає***те, як система була побудована. Це «реалізована» чи «введена у використання» архітектура. Під час змін у ПЗ, в ідеалі її розпорядчу архітектуру змінюють першою. На практиці під час змін ПЗ, змінюють її описову архітектуру. Зазвичай це трапляється через неуважність або інертність розробників.

***Архітектурний дрейф***– це введення основних проектних рішень до описової архітектури ПЗ, які не включені в розпорядчу архітектуру, але які не конфліктують з її рішеннями.

***Архітектурна ерозія*** – це введення архітектурних рішень до описової архітектури ПЗ, при цьому рішення не повинні порушувати його розпорядчої архітектури.

***Відновлення архітектури***– це процес відтворення архітектури на основі компонентів ПЗ, що були отримані на етапі її конструювання. ПЗ не може виконувати свого призначення до того, як воно буде розгорнуте та налаштоване*.* Модулі ПЗ фізично розміщують на пристроях, на яких вони будуть виконуватися. Архітектурне подання розгортання часто є критичним для оцінювання чи буде система відповідати вимогам. Можливі критерії оцінки: використання пам’яті, споживання енергії, вимоги до пропускної спроможності мережі та інші.

Архітектура ПЗ має бути композицією елементів, котрі репрезентують обробку інформації, інформацію, та комунікацію.

***Компонент***– це архітектурна одиниця, яка відповідає за функціональність ПЗ та/або зберігання даних, надає доступ до своїх сервісів за допомогою зовнішнього інтерфейсу і має чітко визначену залежність від контексту у якому цей компонент застосовують. У складних системах взаємодія між компонентами є більш важливою, ніж функціональність окремих частин ПЗ.

***З’єднувач***– це архітектурна одиниця, що відповідає за здійснення взаємодії між компонентами. У багатьох системах з’єднувачі репрезентовані звичайним викликом процедур, або спільним доступом до даних, але в цілому з’єднувачі можуть бути більш складними рішеннями. Компоненти та з’єднувачі, скомпоновані у спеціальному порядку, складають готову архітектуру ПЗ.

***Архітектурна конфігурація*** *або топологія* – це специфікація з’єднань між з’єднувачами та компонентами.

***Нефункціональні властивості системи***(НВС) – це обмеження на те, як ПЗ реалізує і доставляє свою функціональність. Наприклад: ефективність, складність, розширюваність, надійність. Забезпечення необхідної функціональності часто є досить складним завданням через потреби ринку, конкуренцію, жорсткі терміни, обмежені бюджети, тощо. Однак успіх системи цілковито залежить від НВС.

***Роль архітектури***– це забезпечення НВС на рівні архітектурних блоків: компонентів, з’єднувачів, конфігурації.

***Ефективність***– це якість, яка відображає здатність ПЗ до задоволення вимог продуктивності при одночасній мінімізації використання його ресурсів.

***Складність*** вказує до якої міри ПЗ або однин з його компонентів, містить проектні рішення чи реалізацію, які важко зрозуміти і перевірити.

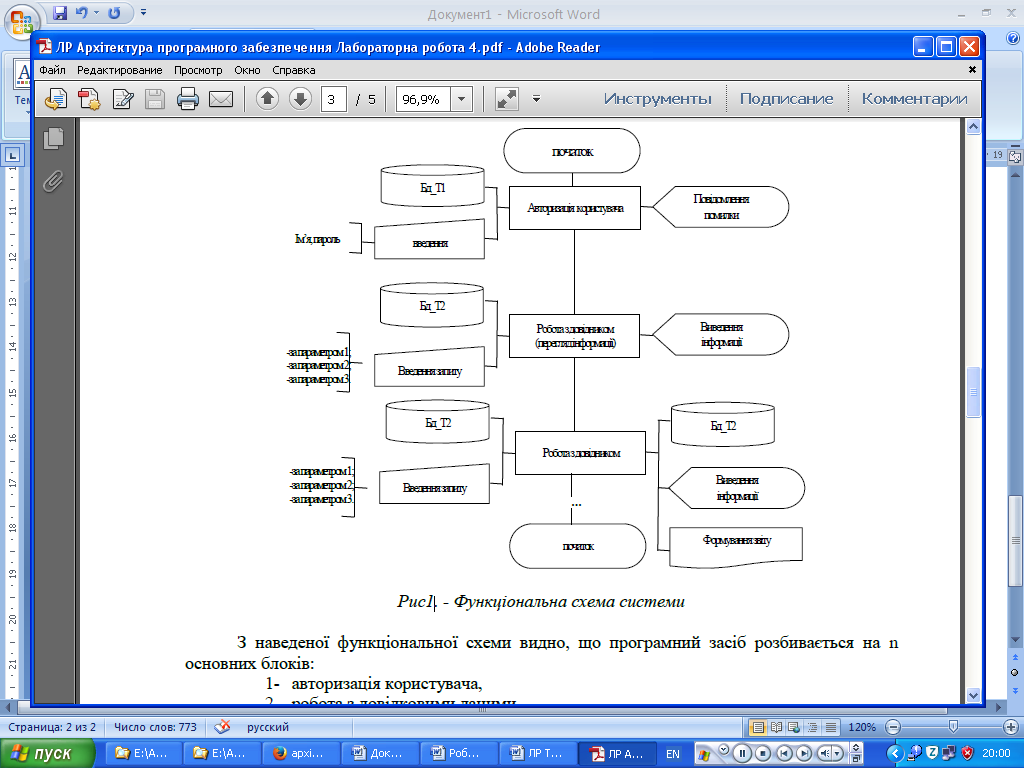
***Масштабованість ПЗ*** – це можливість системи бути зміненою з урахуванням нових вимог.

***Неоднорідність***– це якість ПЗ, що передбачає, що ПЗ складається з декількох різнорідних компонентів або функціонує в декількох різнорідних обчислювальних середовищах одночасно.

***Пристосовуваність***– є здатністю ПЗ до задоволення нових вимог і пристосовуватися до нових умов роботи під час його життєвого циклу.

***Надійність***– це набір властивостей ПЗ, що дозволяє розраховувати, що ПЗ буде функціонувати так, як було заплановано.

***Функціональна схема*** будується з метою розуміння всіх функцій, що виконує програма.

****

З наведеної функціональної схеми видно, що програмний засіб розбивається на n основних блоків:

1- авторизація користувача,

2- робота з довідковими даними,

3- робота з даними по запитах,

4- …

Дані передаються або з БД, або вводяться з клавіатури. Після кожного блоку передбачено перегляд результатів роботи даного блоку. Дані, які змінюються в процесі роботи зберігаються в БД.

При виборі архітектури програмного засобу були поставлені такі завдання і вимоги:

- створення структури даних, що чітко відображають специфіку предметної області;

- моделювання реально існуючих процесів;

- забезпечення оптимальності структур даних;

- поділ і угруповання функцій програмного засобу на підзадачі;

- забезпечення максимальної надійності програмного засобу;

- забезпечення функціональної повноти відповідно до постановки завдання;

- мінімізація інформаційних потоків усередині системи, що дозволяє скоротити час обробки інформації;

- забезпечення наочності модельованих процесів шляхом візуалізації.

Так само в розробляється програмний засіб в якості основних вимог була закладена простота і зручність використання.

В результаті була вибрана модульна структура з функціональною зв'язностю і низьким зчепленням. Тобто структури даних і функції винесені в модулі за функціональною ознакою, що забезпечує реалізацію конкретних підзадач в рамках окремого модуля. Даний підхід дозволяє спростити контроль над збереженням цілісності логіки, а так само спрощує супровід і модернізацію програмного комплексу. Низьке зчеплення модулів[[1]](#footnote-1) дозволяє проводити модернізацію і налагодження кожного модуля окремо, а так само проводити розширення функціональності програмного комплексу шляхом створення додаткових модулів і приєднання його в загальну структуру шляхом підключення його до головного модулю. Все вище перераховане забезпечує значну гнучкість у використанні програмного засобу.

****

Опис модулів програмної системи:

Модуль A в цьому модулі відбувається авторизація користувача, якщо авторизація пройшла успішно модуль запускає модуль MENU;

Модуль MENU основний модуль програмної системи з нього здійснюється зв'язок з іншими модулями ПС;

Модуль AB\_1 викликається з модуля MENU для додавання даних. Після своєї роботи модуль передає управління модулю MENU;

Модуль AB\_2 викликається з модуля MENU для зміни даних. Після своєї роботи модуль передає управління модулю MENU;

Модуль AB\_3 викликається з модуля MENU для видалення даних. Після своєї роботи модуль передає управління модулю MENU;

Модуль AC\_1 …

Модуль AC\_2 …

Модуль AC\_3 …

Модуль AC\_4 …

….

Укрупнену схему надано на рис.3.

**Сервіс: базові поняття**

Сервіс визначається, з одного боку, як відкритий компонент, який може бути елементом швидкої композиції в прикладні програми. З іншого боку, сервіс пропонується як готовий ресурс, який реалізує деякі додаткові можливості, необхідні всім різнорідним програмами для технічної підтримки, потрібної потенційним користувачам. Як правило, описи сервісів містять в собі інформацію про їхні можливості, інтерфейси, поведінку і характеристики. Завдяки такому опису користувач може знайти сервіси, вибрати потрібні і інтегрувати їх в композиційну структуру, як готовий ресурс. Зазвичай розглядається три види сервісів:

1) загальні системні сервіси, які є в кожному загальносистемному середовищі для підтримки процесів проектування і реалізації розподіленої ПС (РПС) на основі сформульованих моделей ПС та РПС;

2) об'єктні сервіси, які підтримують об'єкти і класи, операції ЖЦ, послуги необхідні для розробки РПС в об'єктно-орієнтованому середовищі;

3) веб-сервіси, які базуються на інформаційних ресурсах Інтернет та забезпечують створення елементів РПС шляхом композиції або інтеграції компонентів ПС і сервісів, здатних до функціонування в Інтернеті або Інтранеті.

Сервіси можуть забезпечувати реалізацію базових функцій управління компонентами в середовищі:

1) пошук компонентів;

2) доступ до їх ресурсів;

3) організація обміну інформацією між компонентами;

4) динамічне управління функціонуванням, обумовленим сукупністю компонентів в розподіленому інформаційному середовищі.

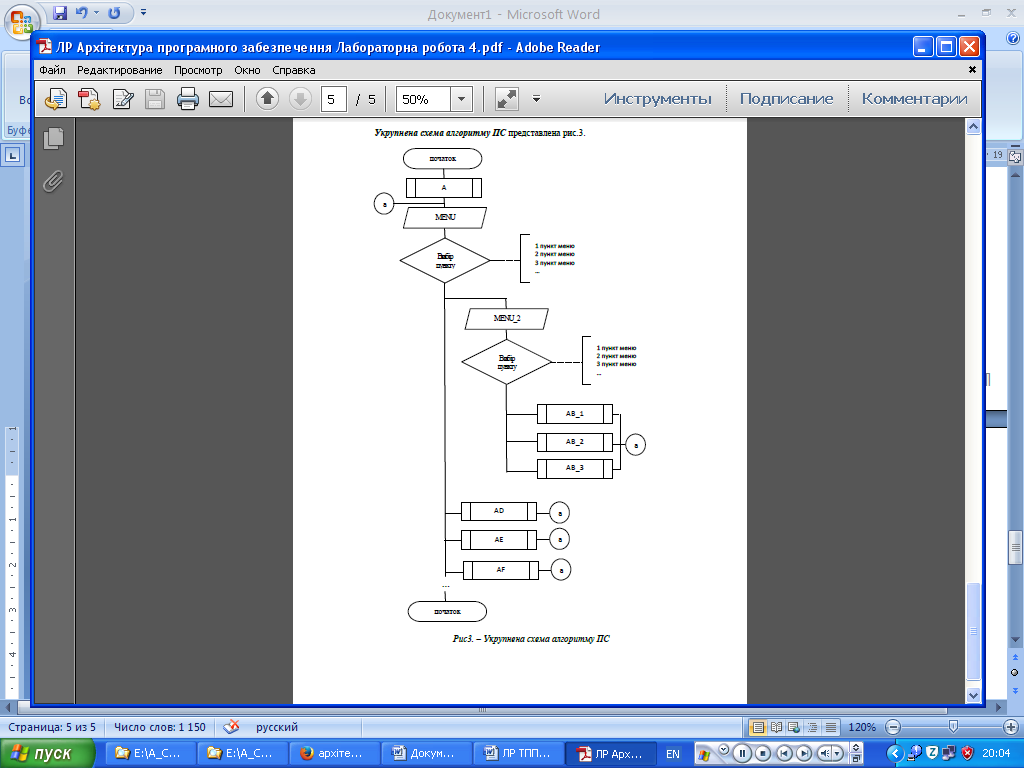
Модель сервісів РПС базується на уніфікації та сумісності, що дозволяє розглядати РПС як набір сервісів, їх функціональність і взаємодію.

Уніфікація сервісів досягається шляхом:

1) типізації функціональності сервісів та інших характеристик;

2) застосування уніфікованих мов для опису сервісів і їх взаємодії;

3) використання стандартних базових технологій.

****

**6.5.Архітектура N-рівневої системи**

ПП складається з деякої впорядкованої сукупності програмних підсистем, названих рівнями (шарами), такий, що:

* на кожному рівні (шарі) нічого не відомо про властивості (і навіть існуванні) подальших (вищих) шарів;
* кожен рівень (шар) може взаємодіяти по управлінню (звертатися до компонентів) з безпосередньо попереднім (нижчим) шаром через заздалегідь певний інтерфейс, нічого не знаючи про внутрішню будову всіх попередніх шарів;
* кожен рівень (шар) має в своєму розпорядженні певні ресурси, які він або приховує від інших шарів, або надає безпосередньо подальшому шару (через вказаний інтерфейс) деякі їх абстракції.

В n-рівневій програмній системі кожен рівень може реалізувати деяку абстракцію даних. Зв'язки між рівнями обмежені передачею значень параметрів звернення кожного рівня до суміжного нижнього рівня і видачею результатів цього звернення від нижнього до верхнього. Тут неприпустимо використання глобальних даних між рівнями.

Як приклад розглянемо використання такої архітектури для побудови операційної системи. Таку архітектуру застосував Дейкстра при побудові операційної системи THE. Ця операційна система складається з чотирьох шарів (див. рис. 6.1). На нульовому шарі проводиться обробка всіх переривань і виділення центрального процесора програмам (процесам) в пакетному режимі. Тільки цей рівень обізнаний про мультипрограмні аспекти системи. На першому шарі здійснюється управління сторінковою організацією пам'яті. Всім вищестоящим шарам надається віртуальна безперервна (не сторінкова) пам'ять. На другому шарі здійснюється зв'язок з консоллю (пультом управління) оператора. Тільки цей шар знає технічні характеристики консолі. На третьому шарі здійснюється буферизація вхідних і вихідних потоків даних і реалізуються так звані абстрактні канали введення і виводу, так що прикладні програми не знають технічних характеристик пристроїв введення і виводу.

|  |  |  |
| --- | --- | --- |
| Комп'ютер | Прикладні програми |  |
| 3: Управління вхідними і вихідними потоками даних |
| 2: Забезпечення зв'язку з консоллю оператора |
| 1: Управління пам'яттю |
| 0: Диспетчеризація і синхронізація процесів |

Рис. 6.1. Архітектура операційної системи THE.

## 6.6.Багатошарова архітектура

Терміни шар (layer) і рівень(tier) часто змішують. Однак -

• шар позначає логічне розділення функціональності,

• рівень позначає фізичне розгортання на різних системах.

Багаторівнева архітектура забезпечує угрупування зв'язаної функціональності застосування в різних шарах, що вибудовуються вертикально, поверх один одного. Функціональність кожного шару об'єднана загальною роллю або відповідальністю. Шари слабо зв'язані, і між ними здійснюється явний обмін даними. Правильне розділення застосування на шари допомагає підтримувати строге розділення функціональності, що у свою чергу, забезпечує гнучкість, а також зручність і простоту обслуговування.

Багатошарова архітектура описана як перевернута піраміда повторного використання, в якій кожен шар агрегує відповідальності і абстракції рівня, розташованого безпосередньо під ним. При строгому розділенні на шари компоненти одного шару можуть взаємодіяти тільки з компонентами того ж шару або компонентами шаруючи, розташованого прямо під даним шаром. Вільніше розділення на шари дозволяє компонентам шари взаємодіяти з компонентами того ж і всіх шарів, що пролягають нижче.

Шари застосування можуть розміщуватися фізично на одному комп'ютері (на одному рівні) або бути розподілені по різних комп'ютерах (n-уровней), і зв'язок між компонентами різних рівнів здійснюється через строго певні інтерфейси. Наприклад, типове Веб-сервер-застосування складається з шару уявлення (функціональність, пов'язана з UI), бізнес-шару (обробка бізнес-правил) і шаруючи даних (функціональність, пов'язана з доступом до даним, часто практично повністю реалізовується за допомогою високорівневих інфраструктур доступу до даним).

Загальні принципи проектування з використанням багатошарової архітектури:

* Абстракція. Багатошарова архітектура представляє систему як єдине ціле, забезпечуючи при цьому достатньо деталей для розуміння ролей і відповідальностей окремих шарів і відносин між ними.
* Інкапсуляція. Під час проектування немає необхідності робити які-небудь припущення про типи даних, методи і властивості або реалізацію, оскільки всі ці деталі приховані в рамках шару.
* Чітко певні функціональні шари. Розділення функціональності між шарами дуже чітка. Верхні шари, такі як шар уявлення, посилають команди нижнім шарам, таким як бізнес-шар і шар даних, і можуть реагувати на події, що виникають в цих шарах, забезпечуючи можливість передачі даних між шарами вгору і вниз.
* Висока зв'язність. Чітко певні межі відповідальності для кожного шару і гарантоване включення в шар тільки функціональності, безпосередньо пов'язаній з його завданнями, допоможе забезпечити максимальну зв'язність в рамках шару.
* Можливість повторного використання. Відсутність залежностей між нижніми і верхніми шарами забезпечує потенційну можливість їх повторного використання в інших сценаріях.
* Слабке скріплення. Для забезпечення слабкого скріплення між шарами зв'язок між ними грунтується на абстракції і подіях.

Прикладами багатошарових застосувань можуть служити бізнес-застосунки (line-of-business, LOB), такі як системи бухгалтерського обліку і управління замовниками; Веб-сервера-застосування і Веб-сайти підприємств; настільні або смарт-клієнти підприємств з централізованими серверами застосувань для розміщення бізнес-логики.

Багатошарова архітектура підтримується поряд шаблонів проектування. Наприклад, під назвою Separated Presentation (Відділення уявлення) об'єднується ряд шаблонів, що розділяють взаємодію користувача з UI, уявлення, бізнес-логіку і дані застосування, з якими працює користувач. Відділення уявлення дозволяє створювати UI в графічних дизайнерах, тоді як розробники пишуть код, що управляє. Таке розділення функціональності на ролі підвищує можливість тестування поведінки окремих ролей.

Можливість застосування багатошарової архітектури необхідно розглянути, якщо у вашому розпорядженні є вже готові рівні, відповідні для повторного використання в інших застосування; якщо є застосування, що надають відповідні бізнес-процеси через інтерфейси сервісів; або якщо створюється складне застосування і попереднє проектування вимагає розділення, щоб групи могли зосередитися на різних ділянках функціональності. Багатошарова архітектура також буде доречна, якщо застосування повинне підтримувати різні типи клієнтів і різні пристрої, або якщо потрібно реалізувати складні і/або бизнес-правила, що настроюються, і процеси.

**6.7.N-рівнева / 3-рівнева архітектура**

N-рівнева і 3-рівнева архітектура є стилями розгортання, що описують розділення ПП на сегменти, багато в чому аналогічно багатошаровій архітектурі, але в даному випадку ці сегменти можуть фізично розміщуватися на різних комп'ютерах, тому їх називають рівнями. Дані архітектурні стилі були створені на базі компонентно-орієнтованого підходу і, як правило, для зв'язку використовують методи платформи, а не повідомлення.

Характеристиками N-рівневої архітектури застосування є функціональна декомпозиція застосування, сервісні компоненти і їх розподілене розгортання, що забезпечує підвищену масштабованість, доступність, керованість і ефективність використання ресурсів. Кожен рівень абсолютно незалежний від всіх останніх, окрім тих, з якими він безпосередньо є сусідами. N-ному рівню вимагається лише знати, як обробляти запит від n+1 рівня, як передавати цей запит на n-1 рівень (якщо такий є), і як обробляти результати запиту. Для забезпечення кращої масштабованості зв'язок між рівнями зазвичай асинхронна.

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

Прикладом N- /3-рівневої архітектури може служити типове фінансове Веб-сервер-застосування з високими вимогами до безпеки. Бізнес-шар в цьому випадку повинен бути розгорнений за міжмережевим екраном, із-за чого доводиться розгортати шар уявлення на окремому сервері в прикордонній мережі.

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

* Зручність підтримки. Рівні не залежать один від одного, що дозволяє виконувати оновлення або зміни, не роблячи впливу на застосування в цілому.
* Масштабованість. Рівні організовуються на підставі розгортання шарів, тому масштабувати застосування задоволено просто.
* Гнучкість. Управління і масштабування кожного рівня може виконуватися незалежно, що забезпечує підвищення гнучкості.
* Доступність. Застосування можуть використовувати модульну архітектуру, яка дозволяє використовувати в системі легко масштабовані компоненти, що підвищує доступність.

Застосування N-/3-рівневої архітектури оправдано, якщо вимоги по обробці рівнів ПЗ відрізняються настільки сильно, що може виникнути перекіс в розподілі ресурсів, або істотно різняться вимоги по безпеці рівнів. Напр., конфіденційні дані не повинні зберігатися на рівні представлення, але можуть розміщуватися на бізнес-рівні або рівні даних. N-/3-рівнева архітектура також підійде у випадку, якщо потрібно забезпечити можливість сумісного використання бізнес-логіки різними ПП і є достатня кількість устаткування для виділення необхідного числа серверів для кожного рівня.

Використовуйте тільки 3 рівні, якщо:

* створюєте застосування для внутрішньої мережі організації, де всі сервери розташовуватимуться в закритій мережі;
* або Інтернет-застосування, вимоги по безпеці якого не забороняють розгортання бізнес-логіки на Веб-сервері, або сервері застосувань.

Розглянете можливість застосування N рівнів, якщо:

* відповідно вимогам по безпеці бізнес-логіка не може бути розгорнена в прикордонній мережі,
* або якщо застосування інтенсивно використовує ресурси, і для розвантаження сервера необхідно перенести цю функціональність на інший сервер.

**6.8.Компонентна архітектура**

Компонентна архітектура описує підхід до проектування і розробки систем з використанням методів проектування програмного забезпечення. Основна увага в цьому випадку приділяється розкладанню дизайну на окремі функціональні або логічні компоненти, певні інтерфейси, що надають чітко, містять методи, події і властивості. В даному випадку забезпечується вищий рівень абстракції, чим при об'єктно-орієнтованій розробці, і не відбувається концентрації уваги на таких питаннях, як протоколи зв'язку або загальний стан.

Компоненти повинні володіти наступними якостями:

* Повторне використання. Як правило, компоненти проектуються із забезпеченням можливості їх повторного використання в різних сценаріях різних застосувань. Проте деякі компоненти створюються спеціально для конкретного завдання.
* Взаємозамінність. Компоненти можуть без зусиль замінюватися іншими подібними компонентами.
* Незалежність від контексту. Компоненти проектуються для роботи в різних середовищах і контекстах. Спеціальні відомості, такі як дані про стан, повинні не включатися або витягуватися компонентом, а передаватися в нього.
* Розширюваність. Компонент може розширювати існуючі компоненти для забезпечення нової поведінки.
* Інкапсуляція. Компоненти надають інтерфейси, що дозволяють зухвалій стороні використовувати їх функціональність, не розкриваючи при цьому деталі внутрішніх процесів або внутрішні змінні або стан.
* Незалежність. Компоненти проектуються з мінімальними залежностями від інших компонентів. Таким чином, компоненти можуть бути розгорнені в будь-якому відповідному середовищі без впливу на інші компоненти або системи.

Зазвичай в застосуваннях використовуються компоненти призначеного для користувача інтерфейсу (їх часто називають елементами управління), такі як таблиці і кнопки, а також допоміжні або службові компоненти, що надають певний набір функцій, використовуваних в інших компонентах. До іншого типу поширених компонентів відносяться ресурсоємні компоненти, доступ до яких здійснюється нечасто, і активація яких виконується «точно вчасно» (just-in-time, JIT) (зазвичай використовується в сценаріях з видаленими або розподіленими компонентами); і компоненти з чергою, виклики методів яких можуть виконуватися асихронно за рахунок застосування черги повідомлень, для зберігання і пересилки.

Компоненти залежать від платформи:

• Об'єктна модель програмних компонентів (component object model, COM) в Windows

• Об'єктна модель розподілених програмних компонентів (distributed component object model, DCOM) в Windows

• Загальна архітектура брокера об'єктних запитів (Common Object Request Broker Architecture, CORBA)

* Серверні компоненти Java (Enterprise JavaBeans, EJB) на інших платформах.

Використовувана компонентна архітектура описує механізми розміщення компонентів і їх інтерфейсів, передачі повідомлень або команд між компонентами і, в деяких випадках, збереження стану.

Основні переваги компонентної архітектури:

* Простота розгортання. Існуючі версії компонентів можуть замінюватися новими сумісними версіями, не роблячи впливу на інші компоненти або систему в цілому.
* Менша вартість. Використання компонентів сторонніх виробників дозволяє розподіляти витрати на розробку і обслуговування.
* Простота розробки. Для забезпечення заданої функціональності компоненти реалізують широко відомі інтерфейси, що дозволяє вести розробку без впливу на інші частини системи.
* Можливість повторного використання. Застосування багато разів використовуваних компонентів означає можливість розподілу витрат на розробку і обслуговування між декількома застосуваннями або системами.
* Спрощення з технічної точки зору. Компоненти спрощують систему через використання контейнера компонентів і його сервісів. Як приклади сервісів, що надаються контейнером, можна привести активацію компонентів, управління життєвим циклом, організацію черги викликів методів, обробку подій і транзакції.

**6.9.Об'єктно-орієнтована архітектура**

Об'єктно-орієнтована архітектура – це парадигма проектування, заснована на розділенні ПЗ або окремої системи на самостійні, придатні для повторного використання, об'єкти, кожен з яких містить дані і поведінку, що відносяться до цього об'єкту.

При об'єктно-орієнтованому проектуванні система розглядається не як набір підпрограм і процедурних команд, а як набори взаємодіючих об'єктів. Об'єкти відособлені, незалежні і слабо зв'язані; обмін даними між ними відбувається через інтерфейси шляхом виклику методів і властивостей інших об'єктів і відправки/прийому повідомлень.

Основними принципами об'єктно-орієнтованого архітектурного стилю є:

* Абстракція. Дозволяє перетворити складну операцію в узагальнення, що зберігає основні характеристики операції. Наприклад, абстрактний інтерфейс може бути широко відомим описом, що підтримує операції доступу до даним через використання простих методів, таких як Get (Отримати) і Update (Відновити). Інша форма абстракції – метадані, використовувані для забезпечення зіставлення двох форматів структурованих даних.
* Композиція. Об'єкти можуть бути утворені іншими об'єктами і за бажанням можуть приховувати ці внутрішні об'єкти від інших класів або надавати їх як прості інтерфейси.
* Наслідування. Об'єкти можуть успадковуватися від інших об'єктів і використовувати функціональність базового об'єкту або перевизначати її для реалізації нової поведінки. Більш того, спадкоємство спрощує обслуговування і оновлення, оскільки зміни, що вносяться до базового об'єкту, автоматично розповсюджуються на всі успадковані від нього об'єкти.
* Інкапсуляція. Об'єкти надають функціональність тільки через методи, властивості і події і приховують внутрішні деталі, такі як стан і змінні, від інших об'єктів. Це спрощує оновлення або заміну об'єктів і дозволяє виконувати ці операції без впливу на інші об'єкти і код, потрібно лише забезпечити сумісні інтерфейси.
* Поліморфізм. Дозволяє перевизначати поведінку базового типу, що підтримує операції в застосуванні, шляхом реалізації нових типів, які є взаємозамінними для існуючого об'єкту.
* Відділення. Об'єкти можуть бути відокремлені від споживача шляхом визначення абстрактного інтерфейсу, що реалізовується об'єктом і зрозумілого споживачеві. Це дозволяє забезпечувати альтернативні реалізації, не роблячи впливу на споживачів інтерфейсу.

Зазвичай, об'єктно-орієнтований стиль використовується для опису об'єктної моделі, що підтримує складні наукові або фінансові операції, або описи об'єктів, що представляють реальні артефакти наочної області (такі як покупець або замовлення).

До основних переваг об'єктно-орієнтованої архітектури відносяться:

* Зрозумілість. Забезпечується ближча відповідність застосування реальним об'єктам, що робить його зрозумілішим.
* Можливість повторного використання. Забезпечується можливість повторного використання через поліморфізм і абстракцію.
* Тестованість. Забезпечується покращувана тестованість через інкапсуляцію. Розширюваність. Інкапсуляція, поліморфізм і абстракція гарантують, що зміни в уявленні даних не вплинуть на інтерфейси, що надаються об'єктами, що могло б обмежити можливості зв'язку і взаємодії з іншими об'єктами.
* Висока зв'язність. Розміщуючи в об'єкті тільки функціонально близькі методи і функції і використовуючи для різних наборів функцій різні об'єкти, можна досягти високого рівня зв'язності.

## 6.10. Сервісно-орієнтована архітектура

Сервісно-орієнтована архітектура (Service-oriented architecture, SOA) забезпечує можливість надавати функціональність застосування у вигляді набору сервісів і створювати застосування, що використовують програмні сервіси. Сервіси слабо зв'язані, тому що використовують засновані на стандартах інтерфейси, які можуть бути викликані, опубліковані і виявлені.

Основне завдання сервісів в SOA – надання схеми і взаємодії із застосуванням за допомогою повідомлень через інтерфейси, областю дії яких є застосування, а не компонент або об'єкт. Не слід розглядати SOA-сервіс як компонентний постачальник сервісів.

SOA-архітектура може забезпечити упаковку бізнес-процесів в сервіси, що підтримують можливість взаємодії і що використовують для передачі інформації широкий діапазон протоколів і форматів даних. Клієнти і інші сервіси можуть виконувати доступ до локальних сервісів, що виконуються на тому ж рівні, або до видалених сервісів по мережі.

Основними принципами архітектурного стилю SOA є:

* Сервіси автономні. Обслуговування, розробка, розгортання і контроль версій кожного сервісу відбувається незалежно від інших.
* Сервіси можуть бути розподілені. Сервіси можуть розміщуватися в будь-якому місці мережі, локально або видалено, якщо мережа підтримує необхідні протоколи зв'язку.
* Сервіси слабо зв'язані. Кожен сервіс абсолютно не залежить від останніх і може бути замінений або оновлений без впливу на застосування, що його використовують, за умови надання сумісного інтерфейсу.
* Сервіси спільно використовують схему і контракт, але не клас. При обміні даними сервіси спільно використовують контракти і схеми, але не внутрішні класи.
* Сумісність заснована на політиці. Політика, в даному випадку, означає опис характеристик, таких як транспорт, протокол і безпека.

Типові сервісно-орієнтовані ПП забезпечують сумісне використання інформації, виконання багатоетапних процесів (системи резервування і онлайн-магазины), надання спеціальних галузевих даних або сервісів між організаціями і створення складених застосувань, які об'єднують дані з багатьох джерел.

Основними перевагами SOA-архітектури є:

* Узгодження наочних областей. Повторне використання загальних сервісів із стандартними інтерфейсами розширює технологічні і бизнес-возможности, а також скорочує вартість.
* Абстракція. Сервіси є автономними, доступ до них здійснюється по формальному контракту, що забезпечує слабке скріплення і абстракцію.
* Можливість виявлення. Сервіси можуть надавати описи, що дозволяє іншим застосуванням і сервісам виявляти їх і автоматично визначати інтерфейс.
* Можливість взаємодії. Оскільки протоколи і формати даних грунтуються на галузевих стандартах, постачальник і споживач сервісу можуть створюватися і розгортатися на різних платформах.
* Раціоналізація. Сервіси забезпечують певну функціональність, усуваючи необхідність її дублювання в застосуваннях.

1. **Зчеплення модуля** – це міра його залежності по способу передачі даних від інших модулів. Чим слабше зчеплення модуля з іншими модулями, тим сильніше його незалежність від інших модулів. [↑](#footnote-ref-1)