
# Основні принципи проектування архітектури ПЗ

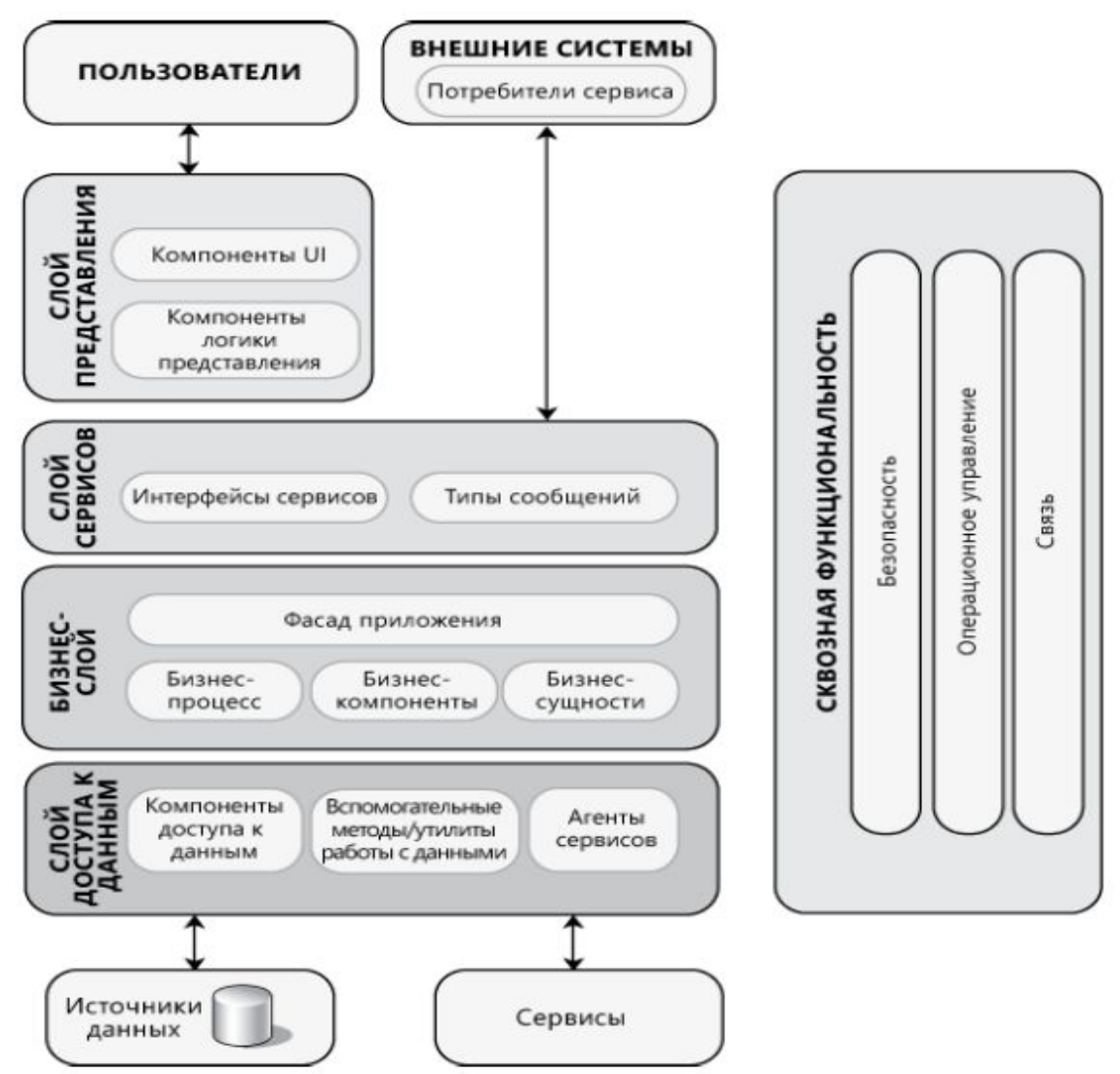
# Зміст

# Вступ

Архітектура ПЗ часто описується як організація або структура системи, де система являє набір компонентів, що виконують певну функцію або набір функцій.

Таку організацію функціональності часто називають угрупованням компонентів по «функціональним областям».

# Типова архітектура ПЗ




Деякі функціональні області використовуються не тільки для групування компонентів, деякі з них присвячені взаємодії і організації спільної роботи компонентів

# Основні принципи проектування

Під час роботи над архітектурою застосунку, особливо на початкових стадіях, необхідно опрацьовувати ітеративно всі основні принципи проектування.

Це допоможе створити архітектуру, яка буде
слідувати перевіреним підходам,
забезпечить:
мінімізацію витрат,
простоту обслуговування,
зручність використання і
розширюваність.

# Поділ функцій

Розділіть застосунок, по можливості, на окремі компоненти з мінімальним перекриттям функціональності.

Важливим фактором є граничне зменшення кількості точок дотику, що забезпечує високе зчеплення (high cohesion) і слабку зв'язаність (low coupling).

Неправильне розмежування функціональності може призвести до високої пов'язаності і складнощів взаємодії, навіть незважаючи на слабке перекриття функціональності окремих компонентів.

# Принцип єдиності відповідальності

Кожен окремо взятий компонент або модуль повинен відповідати тільки за одну конкретну властивість / функцію або сукупність пов'язаних функцій.

# Принцип мінімального знання

Принцип мінімального знання - також відомий як Закон Деметера (Law of Demeter, LoD).

Компоненту або об'єкту не повинні бути відомі внутрішні деталі інших компонентів або об'єктів.

# Не повторюйтесь

Не повторюйтесь (Do not repeat yourself, DRY).

Певна функціональність повинна бути реалізована тільки в одному компоненті і не повинна дублюватися в жодному іншому компоненті.

# Мінімізуйте проектування наперед

Проектуйте тільки те, що необхідно.

У деяких випадках, коли вартість розробки або витрати в разі невдалого дизайну дуже високі, може знадобитися повне попереднє проектування і тестування.

В інших випадках, особливо при гнучкій розробці, можна уникнути масштабного проектування наперед (big design upfront, BDUF).

Якщо вимоги до застосунку чітко не визначені, або існує ймовірність зміни дизайну з часом, намагайтеся не витрачати багато сил на проектування завчасно. Цей принцип називають YAGNI ( «You are not gonna need it»).

# Мінімізуйте проектування наперед

Мета архітектора ПЗ при проектуванні програми або системи - максимальне спрощення дизайну через його розбиття на функціональні області.
Наприклад, призначений для користувача інтерфейс (user interface, UI), виконання бізнес-процесів або доступ до даних - все це різні функціональні області.

Компоненти в кожній з цих областей повинні реалізовувати дану конкретну функціональність і не повинні змішувати в собі код різних функціональних областей.

Так в компонентах UI:
- не повинно бути коду прямого доступу до джерела даних;
- для отримання даних в них повинні використовуватися або бізнес-компоненти, або компоненти доступу до даних.

# Мінімізуйте проектування наперед

Також необхідно проаналізувати співвідношення витрат / вигод для інвестицій в застосунок.

У деяких випадках може бути доцільним спростити структуру і дозволити, наприклад, зв'язування елементів UI з результуючими даними.

Загалом, оцінюйте реалізацію функціональності також і з комерційної точки зору.

# Мінімізуйте проектування наперед

Автор лекції: необхідно притримуватись принципів архітектури доти, доки ви здійснюєте проектування, і створений вами зразок можливо є найкращим для подальшого розвитку системи. Але він може бути не найкращим з точки зору користувача (наприклад: низька швидкість обробки даних), тому після створення зразка, що має зрілий (достатній) рівень для застосування необхідно розглянути проект з точки зору покращення його якості навіть, якщо це призведе до погіршення архітектури.

Версію з гарною архітектурою надалі зберігайте, вона вам може знадобитися для розвитку системи, а покращену версію, з точки зору якості, надавайте користувачу.

# Мінімізуйте проектування наперед

Наприклад:
- перенесення створення об'єктів на початок застосунку, для зменшення часу на створення зазначених об'єктів під час виконання завдань, особливо в циклах;
- об'єднання (зрощення) різних функціональностей в багатофункціональні компоненти з меншим часом на відпрацювання;
- спрощення схеми БД, відмова від певних дій щодо нормалізації, для зменшення часу та пам'яті на обробку інформації;
- перехід від обробки даних порціями до обробки даних одним шматком;
- спеціальна обробка коду застосунку (заміна імен на букви) (особливо коду для транслятора (JavaScript)) для зменшення часу та пам'яті при обробці коду.

# Практики проектування

## Дотримуйтеся однаковості шаблонів проектування в рамках одного шару (рівня)

По можливості, в рамках одного логічного рівня структура компонентів, що виконують певну операцію, повинна бути однаковою.
Наприклад, якщо для об'єкта, який виступає в ролі шлюзу до таблиць або представлень бази даних, вирішено використовувати шаблон Table Data Gateway (Шлюз таблиці даних), не треба включати ще один шаблон, скажімо, Repository (Сховище), який використовує іншу парадигму доступу до даних і ініціалізації бізнес-сутностей.

Однак для задач з більш широким діапазоном вимог може знадобитися застосувати різні шаблони, наприклад, для застосунку, що включає підтримку бізнес-транзакцій і складання звітів.

# Практики проектування

**Не дублюйте функціональність в застосунку**

Та чи інша функціональність повинна забезпечуватися тільки одним компонентом, її дублювання в будь-якому іншому місці є неприпустимим.

Це забезпечує зчеплення компонентів і спрощує їх оптимізацію в разі потреби зміни окремої функціональної можливості.

Дублювання функціональності в застосунку може ускладнити реалізацію змін, знизити зрозумілість програми та створити потенційну можливість для неузгодженостей.

# Практики проектування

**Віддавайте перевагу композиції ніж успадкованості**

За можливості, для повторного використання функціональності застосовуйте композицію, а не успадкування, тому що спадкування збільшує залежність між батьківським і дочірніми класами, обмежуючи, таким чином, можливості повторного використання останніх.

Це також сприятиме зменшенню глибини ієрархій успадкування, що спростить роботу з ними.

# Практики проектування

**Застосовуйте певний стиль написання коду та певну угоду про присвоєння імен для розробки**

Поцікавтеся, чи має організація сформульований стиль написання коду і угоди про присвоєння імен.

Якщо немає, необхідно дотримуватися загальноприйнятих стандартів. В цьому випадку ви отримаєте однакову модель, всі учасники групи зможуть без зусиль працювати з кодом, написаним не ними, тобто код стане більш простим і зручним в обслуговуванні.

# Практики проектування

**Забезпечувати якість системи під час розробки за допомогою методик автоматизованого аналізу (тестування) якості (QA)**

Використовуйте в процесі розробки модульне тестування та інші методики автоматизованого Аналізу якості (Quality Analysis), такі як аналіз залежностей і статичний аналіз коду. Чітко визначайте показники поведінки і продуктивності для компонентів і підсистем і використовуйте автоматизовані інструменти QA в процесі розробки, щоб гарантувати відсутність несприятливого впливу локальних рішень з проектування або реалізації на якість всієї системи.

# Практики проектування

**Враховуйте умови експлуатації програми**

Визначте необхідні IT-інфраструктурні показники і експлуатаційні дані, щоб гарантувати ефективне розгортання і роботу програми.
Приступайте до проектування компонентів і підсистем застосунку, тільки маючи чітке уявлення про їх індивідуальні експлуатаційні вимоги, що істотно спростить загальне розгортання та експлуатацію.

Використання автоматизованих інструментів QA при розробці гарантовано забезпечить отримання необхідних експлуатаційних характеристик компонентів і підсистем програми

# Шари застосунку

**Розділіть функціональні області**

Розділіть застосунок, по можливості, на окремі функції з мінімальним перекриттям функціональності.

Основна перевага такого підходу - незалежна оптимізація функціональних можливостей. Крім того, збій однієї з функцій не призведе до збою інших, оскільки вони можуть виконуватися незалежно одна від одної.

Такий підхід також спрощує розуміння і проектування програми та полегшує управління складними взаємопов'язаними системами

# Шари застосунку

**Явно визначайте зв'язок між шарами**

Рішення, в якому кожен шар застосунку може взаємодіяти або має залежності з усіма іншими шарами, є складним для розуміння і управління.

Приймайте явні рішення про залежність між шарами і про потоки даних між ними.

# Шари застосунку

**Реалізуйте слабке зв'язування шарів за допомогою абстракції**

Це можна реалізувати, визначаючи інтерфейсні компоненти з добре відомими вхідними та вихідними характеристиками, такі як фасад, які перетворять запити в формат, зрозумілий компоненту шару.

Крім того, також можна визначати загальний інтерфейс або абстракцію, що спільно використовується (протилежність залежності), яка повинна бути реалізована компонентами інтерфейсу, використовуючи інтерфейси або абстрактні базові класи

# Шари застосунку

**Не змішуйте різні типи компонентів на одному логічному рівні**

Починайте з ідентифікації функціональних областей і потім групуйте компоненти, асоційовані з кожною з цих областей в логічні рівні.

Наприклад, шар UI не повинен включати компоненти виконання бізнес-процесів, в нього повинні входити тільки компоненти, що використовуються для обробки користувальницького введення запитів

# Шари застосунку

**Дотримуйтесь єдиного формату даних в рамках шару або компонента**

Змішування форматів даних ускладнить реалізацію, розширення і обслуговування програми.
Будь-яке перетворення одного формату даних в інший вимагає реалізації коду перетворення і тягне за собою витрати на обробку.

# Компоненти, модулі та функції

**Компонент або об'єкт не повинен покладатися на внутрішні дані інших компонентів або об'єктів**

Кожен метод, що викликається компонентом або методом іншого об'єкта або компонента, повинен мати у своєму розпорядженні достатні відомості про те, як обробляти запити, що надходять і, в разі необхідності, як перенаправляти їх до відповідних підкомпонентів або інших компонентів.

Це сприяє створенню більш зручних в обслуговуванні застосунків та застосунків, що легко адаптуються

# Компоненти, модулі та функції

**Не перевантажуйте компонент функціональністю**

Наприклад, компонент UI не повинен включати код для доступу до даних або забезпечувати додаткову функціональність.
Перевантажені компоненти часто мають безліч функцій і властивостей, поєднуючи бізнес-функціональність і наскрізну функціональність, такі як протоколювання і обробка виключень. В результаті виходить дуже нестійкий до помилок і складний в обслуговуванні дизайн.

Застосування принципів виключної відповідальності і поділу функціональності допоможе уникнути цього

# Компоненти, модулі та функції

**Розберіться з тим, як буде здійснюватися зв'язок між компонентами**

Це вимагає розуміння сценаріїв розгортання, які має підтримувати створюваний застосунок.

Необхідно визначити, чи будуть всі компоненти виконуватися в рамках одного процесу або необхідно забезпечити підтримку зв'язку через фізичні кордони або межі процесу, ймовірно, шляхом реалізації інтерфейсів взаємодії на основі повідомлень

# Компоненти, модулі та функції

**Максимально ізолюйте наскрізну функціональність від бізнес-логіки застосунку**

Наскрізна функціональність - це аспекти безпеки, обміну інформацією або керованості, такі як протоколювання і інструментування. Змішання коду, що реалізує ці функції, з бізнес-логікою може призвести до створення дизайну, який буде складно розширювати і обслуговувати.

Внесення змін до наскрізної функціональності зажадає переробки всього коду бізнес-логіки. Розгляньте можливість використання інфраструктур і методик (таких як аспект-орієнтоване програмування), які допоможуть в реалізації такої функціональності

# Компоненти, модулі та функції

**Визначайте чіткий контракт для компонентів**

Компоненти, модулі та функції повинні визначати контракт або специфікацію інтерфейсу, що чітко обумовлює їх використання і поведінку.

Контракт повинен описувати, як інші компоненти можуть здійснювати доступ до внутрішньої функціональності компонента, модуля або функції, і поведінку цієї функціональності з точки зору попередніх умов, постумов, побічних ефектів, винятків, ефективності його роботи та інших факторів

# Основні питання проектування

Основні питання проектування:
- визначення типу застосунку;
- вибір стратегії розгортання;
- вибір відповідної технології;
- вибір показників якості;
- рішення щодо шляхів реалізації наскрізної функціональності.

# Визначення типу застосунку

Вибір відповідного типу застосунку - ключовий момент процесу проектування застосунку.
Цей вибір визначається конкретними вимогами і обмеженнями середовища. Від багатьох застосунків потрібна підтримка безлічі типів клієнтів і можливість використання більш одного базового архетипу.

Основні типи застосунків:
- застосунок для мобільних пристроїв;
- насичені клієнтські програми для виконання переважно на клієнтських ПК;
- насичені клієнтські програми для розгортання з Інтернету з підтримкою насичених UI і мультимедійних сценаріїв;
- сервіси, розроблені для забезпечення зв'язку між слабко зв'язаними компонентами;
- веб-застосунки для виконання переважно на сервері в сценаріях з постійним підключенням.

# Визначення типу застосунку

Крім того, є більш спеціальні типи застосунків. До них відносяться:
- Програми та сервіси, що розміщуються в центрах обробки даних (ЦОД) і в хмарі.
- Офісні бізнес-застосунки (Office Business Applications, OBAs), що інтегрують технології Microsoft Office і Microsoft Server.
- Бізнес-застосунки SharePoint (SharePoint Line of Business, LOB), що забезпечують доступ до бізнес-даних і до функціональних можливостей через портал.

# Вибір стратегії розгортання

Застосунок може розгортатися в різних середовищах, кожна з них має особистий набір обмежень, такі як фізичний розподіл компонентів по серверам, обмеження по використанню мережевих протоколів, налаштування міжмережевих екранів та маршрутизаторів та інше.

Існує декілька загальних схем розгортання, які описують переваги та мотиви використання ряду розподілених та нерозподілених сценаріїв.

При вибору стратегії необхідно найти компроміс між вимогами застосунку і відповідними схемами розгортання, обладнанням, що підтримується і обмеженнями, що накладаються середовищем на варіанти розгортання

# Вибір відповідних технологій

Ключовим фактором при виборі технологій для застосунку є тип застосунку, що розробляється, а також кращі варіанти топології розгортання програми і архітектурні стилі.

Вибір технологій також визначається політиками організації, обмеженнями середовища, кваліфікацією ресурсів і таке інше.

Необхідно порівняти можливості технологій, що обираються до вимог застосунку, беручи до уваги всі ці фактори.

# Вибір показників якості

Показники якості, такі як безпека, продуктивність, зручність і простота використання, допомагають сфокусувати увагу на критично важливих проблемах, які повинен вирішувати створюваний дизайн.

Залежно від конкретних вимог може знадобитися розглянути всі показники якості або тільки деякі з них.
Наприклад, питання безпеки та продуктивності необхідно врахувати при розробці кожної програми, тоді як проблеми можливості взаємодії або масштабованості стоять далеко не перед всіма проектами.

Насамперед, необхідно зрозуміти поставлені вимоги і сценарії розгортання, щоб знати, які показники якості важливі для створюваного застосунку.

Не можна також забувати про можливість конфлікту між показниками якості. Наприклад, часто вимоги безпеки йдуть врозріз з продуктивністю або зручністю використання.

# Вибір показників якості

При проектуванні з урахуванням показників якості слід керуватися таким:

Показники якості - це властивості системи, відокремлені від її функціональності.

З технічної точки зору, реалізовані показники якості відрізняють хорошу систему від поганої.

Існує два типи показників якості: вимірювані під час виконання і ті, оцінити які можна тільки за допомогою перевірки.

Необхідно провести аналіз і знайти оптимальне співвідношення між показниками якості.

Питання, на які необхідно відповісти при розгляді показників якості:
Які основні показники якості програми? Визначте їх в ході процесу розробки.
Які основні вимоги для реалізації цих показників? Чи піддаються вони кількісному визначенню?
Які критерії приймання, що будуть свідчити про виконання вимог?

# Рішення про шляхи реалізації наскрізної функціональності

Наскрізна функціональність представляє ключову область дизайну, не пов'язану з конкретним функціоналом програми.

Наприклад, необхідно розглянути можливості реалізації централізованих або загальних рішень для таких аспектів:
механізм протоколювання, що забезпечує можливість кожному шару вести журнал в загальному сховищі або в різних сховищах, але таким чином, щоб результати могли бути зіставлені (порівнянні) згодом;
механізми аутентифікації і авторизації, що забезпечують передачу посвідчень на різні рівні для надання доступу до ресурсів;
інфраструктура управління винятками, яка буде функціонувати в кожному шарі і між рівнями, якщо виключення поширюються в рамках системи;
підхід до реалізації зв'язків, що використовується для забезпечення обміну інформацією між шарами;
загальна інфраструктура кешування, що дозволяє кешувати дані в шарі представлення, бізнес-шарі і шарі доступу до даних.

# Рішення про шляхи реалізації наскрізної функціональності

Основні аспекти наскрізний функціональності, які необхідно розглянути при створенні архітектури додатків:

**Інструментування і протоколювання**.

Забезпечуйте управління і моніторинг всіх критично важливих для бізнес-логіки і системи подій. Протоколюється достатня кількість відомостей для відтворення подій в системі без включення конфіденційних даних.

**Аутентифікація**.

Визначтеся з тим, як буде проходити аутентифікація користувачів і передача автентифікованих посвідчень між шарами.

# Рішення про шляхи реалізації наскрізної функціональності

**Управління винятками.**
Перехоплюйте виключення на функціональних, логічних і фізичних межах і уникайте розкриття конфіденційних відомостей кінцевим користувачам.

**Зв'язок.**
Виберіть відповідні протоколи, зведіть до мінімуму кількість викликів по мережі і захистіть передачу конфіденційних даних по мережі.

**Кешування.**
Визначте, що має кешуватися і де для поліпшення продуктивності і скорочення часу відгуку застосунку. При проектуванні кешування не забудьте врахувати особливості Веб-структури застосунку і структури застосунку.

# Заключна частина

Не зважаючи на те, що Ви дуже добре розумієте важливість архітектури, будьте готові до того, що під час життєвого циклу Вашого застосунку, Вам необхідно буде неодноразово змінювати вашу архітектуру

# Основні принципи проектування архітектури ПЗ

**Дякую за увагу**