**Лекція 6. Архітектура програмного забезпечення**

Архітектура програмного забезпечення як концепція була уперше визначена в роботі Дейкстры 1968 і Дэвидом Парнасом на початку 1970-х. Ці учені робили наголос на тому, що структура програмного забезпечення відіграє важливу роль і визначення правильної структури є критичним чинником в процесі розробки.

Основною ідеєю програмної архітектури є ідея зниження складності системи шляхом абстракції і розмежування повноважень.

Стандарт "ANSI/IEEE 1471-2000 : Рекомендована практика і архітектурний опис програмних систем"  є першим стандартом в області архітектури ПО, в  2007 був адаптований організацією стандартизації ISO як ISO/IEC 42010.

"Архітектура ― це фундаментальна організація системи, втілена в компонентах, їх взаємозв'язках, середовищі, і принципах, що управляють їх дизайном і еволюцією". [IEEE 1471]

У цьому стандарті також визначаються такі терміни, пов'язані з цим визначенням:

*Система* - це набір компонентів, об'єднаних для виконання певної функції або набору функцій. Термін "система" охоплює окремі додатки, системи в традиційному сенсі, підсистеми, системи систем, лінійки продуктів, сімейства продуктів, цілі корпорації та інші агрегації, що мають відношення до даної теми. Система існує для виконання однієї або більше місій у своєму оточенні.

*Оточення*, чи контекст, визначає хід і обставини економічних, експлуатаційних, політичних та інших впливів на систему.

*Місія* - це застосування або дія, для якого одна чи кілька зацікавлених осіб планують використовувати систему у відповідності з деяким набором умов.

*Зацікавлена особа* - це фізична особа, група або організація (або її категорії), які зацікавлені в системі або мають пов'язані з нею завдання.

*Компонент* - модульна частина системи, яка інкапсулює її вміст; втілення компонента є заміщуваним у його оточенні. Поведінка компонента визначається в термінах наданого і необхідного інтерфейсів. Таким чином, компонент використовується в якості типу, відповідність якого описується цими двома інтерфейсами, тим, що надається і необхідним (об'єднуючи як статичну, так і динамічну їх семантику).

Область програмної архітектури ділиться на кілька підобластей. Робоча група IFIP 2.10 (International Federation of Information Processing - Міжнародна федерація з обробки інформації) визначає наступні п'ять.

Архітектурний проект: як ми створюємо архітектуру?

Аналіз: як ми на основі архітектури відповідаємо на питання про властивості кінцевого продукту?

Реалізація: як ми створюємо систему на базі архітектурного опису?

 Подання: як ми створюємо надійні артефакти, щоб "донести" архітектуру до людей і машин?

Економіка: які архітектурні проблеми впливають на бізнес-рішення?

Програмна архітектура тісно пов'язана і з іншими дисциплінами. Це проектування і багаторазове використання ПЗ, розробка систем і системна архітектура, корпоративна архітектура і зворотне проектування, розробка вимог і питання якості.

**Складові елементи Архітектури ПЗ**

Архітектура програмної системи складається з трьох взаємодіючих елементів:

1) Структура - статична складова, яка показує розподіл відповідальності між подсистемами,

2) Поведінка - динамічна складова, взаємозв'язки і взаємодія між цими структурами, і 3) Стиль - принципи і керівництво которыме використовувалися і використовуватимуться при визначенні структури.

**Структура**

Структурні елементи архітектури охоплюють статичний набір підсистем і компонентів що становлять систему а також відповідальності цих підсистем і їх организіцию  у більші підсистеми. Важливо розглядати також залежності між компонентами і іншими зовнішніми системами. На цьому рівні і з цієї точки зору організація архітектури важлива також для координування робіт і планировния проекту. Часто  команда розробників проекту організовується згідно із структурою ― різні групи відповідають за реалізацію різних підсистем. Постійність у вибраній архітектурі проекту і " архітектурі" команди розробників", працюючих над проектом дозволяє ефективно управляти плануванням і розробкою.

**Поведінка**

Поведінка визначає способи за допомогою яких підсистеми і окремі компоненти взаємодіють один з одним згідно з системними вимогами.

**Стиль**

Стиль визначає принципи і шаблони(патерни) вживані для побудови структури і визначення поведінки архітектури. Стиль включає концепції(архітектурних патернів напр.), технічні рішення і оцінок(простота дизайну, напр.).

Стиль використовується для визначення словників понять, написання керівництва, визначення обмежень архітектури, що використовується надалі для ефективного ухвалення рішень  і спрощує розуміння предмета при обговоренні архітектури. Важливо щоб стиль був визначений і представлений в документації явним чином, оскільки подальша еволюція продукту залежить від розуміння чинників, які вплинули на існучу архітектуру.

Визначення архітектури програмного застосування це процес визначення структурованого рішення, яке задовольнятиме усім технічним і експлуатаційним вимогам, і, в той же час мати оптимального за такими якісними характеристиками як продуктивність, безпеку, керованість. Процес визначення архітектури включає ряд кроків по ухваленню  рішень. Ці рішення базуються на безлічі впливаючих чинників, і кожне з цих рішень  може мати істотний вплив на якість, продуктивність, простоту подальшої підтримки в експлуатації(maintainability), і загальний успіх додатка.

Архітектура фокусується на тому як основні елементи і компоненти усередині додатка повинні використовуватися або взаємодіють між собою. Вибір конкретних структур цих і обчислювальних алгоритмів і реалізація індивідуальних компонентів відносяться до питань дизайну програмного забезпечення. Завдання архітектура, що розглядаються при виборі, і визначення дизайну додатка часто перекриваються. Тому ці два процеси побудови програмного застосування не розділяються, а комбінуються. У одних випадках вимагається приймати архітектурні рішення, в інших рішення торкаються дизайну і визначення того як можна реалізувати вибрану архітектуру.

Архітектура надає рішення для нефунциональных вимог, які пред'являються до додатка, тоді як дизайн відповідає за рішення що відповідають функціональним вимогам проекту.

Згідно із стандартом IEEE функціональні і нефункціональні вимоги визначаються таким чином:

**Функціональна вимога** ― ця вимога до системи/програмному застосуванню, яке повинне виконуватися відповідним компонентом. Це вимоги, які визначають поведінку і дії системи, що становлять фундаментальний процес або перетворення, які програмні і апаратні компоненти системи роблять над вхідними даними для отримання вихідних даних. *Приклад: "Система роздруковує рахунок"*

**Нефункціональна вимога** ― ця вимога до програмного забезпечення, яке описує не то що робить програма, але як програма виконує функції. Наприклад продуктивність ПЗ, вимоги до зовнішніх інтерфейсів, вимоги до якості програмного забезпечення. Частий ці вимоги оцінюються суб'єктивно із-за складності їх тестування. *Приклад: "Система роздруковує рахунок швидко"*

Розробка архітектури є 1-м етапом боротьби зі складністю ПЗ, на якому реалізується принцип виділення незалежних компонент.

**1. Основні завдання розробки архітектури ПЗ**:

- виділення програмних підсистем і відображення в них зовнішніх функцій (заданих в функціональній специфікації);

- встановлення способів взаємодії між виділеними програмними підсистеми.

Архітектура программного продукту (ПП) розробляється на основі функціональної специфікації (ФС). Тобто, проводиться подальша конкретизація функціональної специфікації, яка призводить до формування архітектури.

Більшість описів архітектури є рисунками, в яких прямокутниками показані виконувані компоненти, а стрілками - взаємодія серед цих компонентів. Ці схеми відображають існуючий досвід створення систем, подібних до тої, що проектується.

Основні питання для розробника архітектури ПП:

1.Як користувач буде використовувати ПП?

2.Як ПП буде розгортатися і обслуговуватися при експлуатації?

3.Які висуваються вимоги до таких атрибутів ПП, як: якість, безпека, продуктивність, можливість паралельної обробки, інтернаціоналізація і конфігурація?

4.Як спроектувати ПП, щоб він залишався гнучким і зручним в обслуговуванні протягом усього періоду експлуатації?

5.Які основні архітектурні елементи, які можуть впливати на ПП зараз або після його розгортання?

Необхідно пам'ятати, що архітектура повинна:

1.Формувати структуру системи, але приховувати її.

2.Розкривати всі можливі варіанти використання і сценарії ПП.

3.Відповідати вимогам усіх зацікавлених сторін.

4.Виконувати вимоги, як по функціональності, так і по якості.

**2. Основні принципи проектування архітектури**

При проектуванні архітектури слід керуватися наступними рекомендаціями:

1.Враховуйте можливі майбутні зміни. Продумайте, як з часом може знадобитися змінити ваш ПП, щоб він відповідав новим вимогам і завданням, і передбачте необхідну гнучкість.

2.Створюйте архітектуру, яка зменшує ризики. Використовуйте засоби проектування, системи моделювання, такі як (Unified Modeling Language, UML), і засоби візуалізації, коли необхідно виявити ризики, ухвалити архітектурні і проектні рішення, та проаналізувати їх наслідки. Проте, не створюйте дуже формалізовану модель, вона може обмежити можливості для виконання ітерацій і адаптації дизайну.

3.Використовуйте колективну роботу та візуалізацію. Для побудови хорошої архітектури критично важливим є ефективний обмін інформацією про дизайн, ухвалювані рішення і зміни. Використовуйте моделі, представлення та інші способи візуалізації архітектури для ефективного обміну інформацією і зв'язку зі всіма зацікавленими сторонами, а також для забезпечення швидкого повідомлення про зміни в архітектурі.

4.Визначіть ключові інженерні рішення. Намагайтесь зрозуміти ключові інженерні рішення і області, в яких найвірогідніше можуть виникнути помилки. На самому початку проекту приділіть достатню кількість часу і уваги для ухвалення правильних рішень. Це забезпечить створення гнучкішого дизайну, внесення змін до якого не буде потребувати повної його переробки.

Приступаючи до роботи над архітектурою ПЗ, необхідно пам'ятати про основні принципи проектування. Це допоможе створити архітектуру, яка буде відповідати перевіреним підходам, забезпечить мінімізацію витрат, простоту обслуговування, зручність використання і розширюваність.

Розглянемо **основні принципи**:

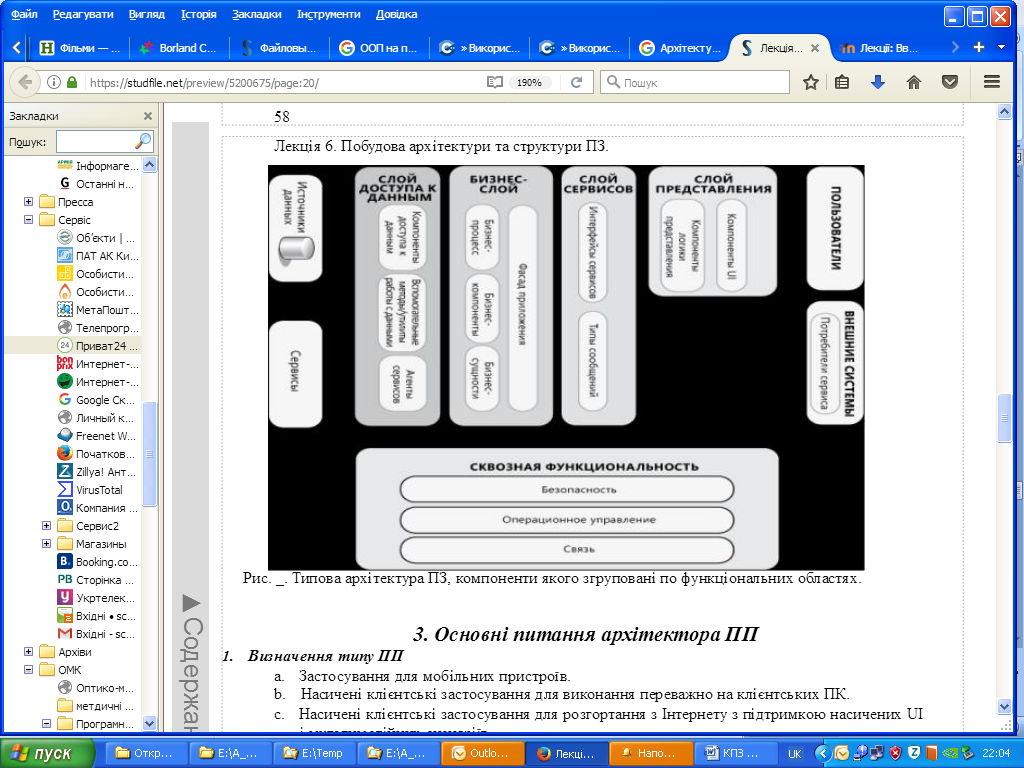
1.Принцип розділення функцій. ПЗ повинно бути розділене на окремі компоненти з мінімальним перекриттям функціональності. Важливим чинником є граничне зменшення кількості точок зіткнення, що забезпечить високу зв'язність (high cohesion) і слабке зчеплення (low coupling). Невірне розмежування функціональності може привести до високої зв'язаності й складнощі взаємодії, навіть не дивлячись на слабке перекриття функціональності окремих компонентів.

2.Принцип єдиності відповідальності. Кожен окремо узятий компонент або модуль повинен відповідати тільки за одну конкретну властивість/функцію або сукупність зв'язаних функцій.

3.Принцип мінімального знання (також відомий як Закон Деметера (Law of Demeter, LOD)). Будь-якій компоненті або об'єкту не повинні бути відомі внутрішні деталі інших компонентів або об'єктів.

4.Принцип неповторюваності (Don’t repeat yourself, DRY). Намір повинен бути відображений тільки один раз. У проектуванні архітектури ПЗ це означає, що певна функціональність повинна бути реалізована тільки в одному компоненті (об’єкті) і не повинна дублюватися ні в одному іншому місці.

5.Мінімізуйте попереднє проектування. Проектуйте тільки те, що є необхідним. Лише в окремих випадках, коли вартість розробки (або втрат у разі невдалого дизайну) дуже високі, допускається повне попереднє проектування і тестування. У інших випадках, особливо при гнучкій розробці, можна уникнути масштабного попереднього проектування. Якщо вимоги до ПП розпливчасті і чітко не визначені, або існує вірогідність зміни дизайну з часом, намагайтесь не витрачати багато сил на попереднє проектування.



## 3. Основні питання архітектора ПП

1. Визначення типу ПП

a. Застосування для мобільних пристроїв.

b. Насичені клієнтські застосування для виконання переважно на клієнтських ПК.

c. Насичені клієнтські застосування для розгортання з Інтернету з підтримкою насичених користувальницького інтерфкйсу (UI) і мультимедійних сценаріїв.

d. Сервіси, розроблені для забезпечення зв'язку між слабо зв'язаними компонентами.

e. Веб-сервера-застосування для виконання переважно на сервері в сценаріях з постійним підключенням.

f. Застосування і сервіси, що розміщуються в центрах обробки даних (ЦОД) і в хмарі.

g. Офісні бизнес-додатки (Office Business Applications, OBAs), інтегруючі технології Microsoft Office і Microsoft Server.

2. Вибір стратегії розгортання

a. Застосування може розгортатися в різноманітних середовищах, кожна з яких матиме власний набір обмежень, таких як фізичний розподіл компонентів по серверах, обмеження по мережевих протоколах, настройки міжмережевих екранів і маршрутизаторів і т.д. Існує декілька загальних схем розгортання.

b. При виборі стратегії необхідно знайти компроміс між вимогами застосування і відповідними схемами розгортання, підтримуваним устаткуванням, і обмеженнями, що накладаються середовищем на варіанти розгортання.

3. Вибір відповідних технологій

a. Ключовим чинником при виборі технологій для застосування є тип ПП, що розробляється, а також переважні варіанти топології його розгортання і архітектурні стилі.

b. Вибір технологій також визначається політиками організації, обмеженнями середовища, кваліфікацією ресурсів і т.д.

c. Необхідно порівняти можливості вибираних технологій з вимогами до ПП і зважити на всі ці чинники.

4. Вибір показників якості

a. Такі показники якості, як безпека, продуктивність, зручність і простота використання, допомагають сфокусувати увагу на критично важливих проблемах, які повинен вирішувати дизайн ПП.

b. Залежно від конкретних вимог може знадобитися розглянути всі згадані показники якості або тільки деякі з них. Напр., питання безпеки і продуктивності необхідно врахувати при розробці кожного модуля, тоді як проблеми взаємодії чи масштабованості стоять далеко не перед усіма проектами.

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

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

5. Рішення про шляхи реалізації наскрізної функціональності

a. Протоколювання. Забезпечуйте управління і моніторинг всіх подій, які критично важливі для бізнес-логіки і системи. Протоколюйте достатню кількість відомостей для відтворення подій в системі без включення конфіденційних даних.

b. Автентифікація (процедура встановлення належності користувачеві інформації в системі пред'явленого ним ідентифікатора). Визначіть те, як буде відбуватись автентифікація користувачів і передача автентифікованих посвідчень між рівнями.

c. Авторизація (керування рівнями та засобами доступу до певного захищеного ресурсу в галузі цифрових технологій (наприклад, автоматизована система контролю доступу) та ресурсів системи залежно від ідентифікатора і пароля користувача або надання певних повноважень (особі, програмі) на виконання деяких дій у системі обробки даних). На кожному рівні і на їх межах забезпечте відповідну авторизацію з необхідною деталізацією.

d. Управління виключеннями. Перехоплюйте усі виключення на функціональних, логічних і фізичних рівнях та уникайте розкриття конфіденційних відомостей кінцевим користувачам.

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

f. Кешування[[1]](#footnote-1). Визначте, що і де повинно кешуватися для поліпшення продуктивності і скорочення часу відгуку ПЗ. При проектуванні кешування не забудьте врахувати особливості Веб-серверів та web-застосувань.

**4. Архітектурні стилі/парадигми**

|  |  |  |
| --- | --- | --- |
|  | Архітектурний стиль/парадигма | Опис |
| 1. | Цілісна програма | Представляє вироджений випадок архітектури ПП: до складу ПЗ входить тільки одна програма |
| 2. | Шина повідомлень | Архітектурний стиль програмної системи, яка може приймати і відправляти повідомлення поодинці або пакетами по каналах зв'язку. Тоді ПЗ дістають можливість взаємодіяти, не маючи в своєму розпорядженні конкретних відомостей один про одного. |
| 3. | Клієнт/сервер | Система розділяється на дві частини, де клієнт виконує запити до сервера. У багатьох випадках в ролі сервера виступає СУБД, а бізнес-логіка представлена сторид-процедурами на сервері БД. |
| 4. | Об'єктно орієнтована - | Парадигма проектування, заснована на розподілі відповідальності застосування або системи між окремими багато разів використовуваними і самостійними об'єктами, що містять дані і поведінку. |
| 5. | Проблемно- орієнтована | Об'єктно-орієнтований архітектурний стиль, орієнтований на моделювання сфери ділової активності і визначальний бизнес-объекты на підставі суті цієї сфери. |

|  |  |  |
| --- | --- | --- |
| 6. | Багатошарова архітектура | Функціональні області ПП розділяються на багатошарові групи (рівні). |

|  |  |  |
| --- | --- | --- |
| 7. | N-рівнева | Функціональність виділяється в окремі сегменти, багато в чому аналогічно багатошаровому стилю, але в даному випадку сегменти фізично розташовуються на різних комп'ютерах. |

|  |  |  |
| --- | --- | --- |
| 8. | Компонентна архітектура | ПЗ розкладається на функціональні або логічні компоненти з можливістю повторного їх використання, завдяки ретельно пропрацьованим інтерфейсам. |

|  |  |  |
| --- | --- | --- |
| 9. | Сервіс-орієнтована архітектура (SOA) | Описує застосування, що за допомогою контрактів і повідомлень надають і використовують функціональність у вигляді сервісів. |

**5. Поєднання архітектурних стилів**

Архітектура програмної системи практично ніколи не обмежена лише одним архітектурним стилем, часто вона є поєднанням архітектурних стилів, створюючих повну систему. Наприклад, може існувати SOA-дизайн, що складається з сервісів, при розробці яких використовувалася багатошарова архітектура і об'єктно-орієнтований архітектурний стиль.

Поєднання архітектурних стилів також корисно при побудові Інтернет Веб-застосувань, де можна досягти ефективного розділення функціональності за рахунок застосування багатошарового архітектурного стилю. Таким чином можна відокремити логіку уявлення від бизнес-логики і логіків доступу до даним. Вимоги безпеки організації можуть обумовлювати або 3-рівневе розгортання застосування, або розгортання з більш ніж трьома рівнями. Рівень уявлення може розгортатися в прикордонній мережі, розташованій між внутрішньою мережею організації і зовнішньою мережею. Як модель взаємодії на рівні уявлення може застосовуватися шаблон уявлення з відділенням (різновид багатошарового стилю), така як Model-View-Controller (MVC)5. Також можна вибрати архітектурний стиль SOA і реалізувати зв'язок між Веб -сервером-сервером і сервером застосувань за допомогою обміну повідомленнями.

Створюючи настільне застосування, можна реалізувати клієнт, який відправлятиме запити до програми на сервері. В цьому випадку розгортання клієнта і сервера можна виконати за допомогою архітектурного стилю клієнт/сервер і використовувати компонентну архітектуру для подальшого розкладання дизайну на незалежні компоненти, що надають відповідні інтерфейси. Застосування об'єктно-орієнтованого підходу до цих компонентів підвищить можливості повторного використання, тестування і гнучкість.

## 6. Основні класи архітектур ПП

## 6.1. Цілісна програма

Таку архітектуру вибирають зазвичай у тому випадку, коли ПП повинен виконувати одну яскраво виражену функцію і її реалізація не представляється дуже складною. Така архітектура не вимагає якого-небудь опису (окрім фіксації класу архітектури), оскільки відображення зовнішніх функцій на цю програму є тривіальним, а визначати спосіб взаємодії не потрібно (через відсутність якої-небудь зовнішньої взаємодії програми, окрім як взаємодії її з користувачем, а останнє описується в документації до ПП).

## 6.2. Архітектура прикладних ПП (ППП)

Складається з деякого набору незалежних програм, таких, що:

• будь-яка з цих програм може бути активізована (запущена) користувачем;

• при виконанні активізованої програми інші програми цього набору не можуть бути активізовані до тих пір, поки не закінчить виконання активізована програма;

• всі програми цього набору стосуються одного і того ж інформаційного середовища.

Таким чином, програми цього набору по управлінню ніяк не взаємодіють - взаємодія між ними здійснюється тільки через загальне інформаційне середовище.

## 6.3. Архітектура, заснована на шині повідомлень

Архітектура на шині повідомлень описує принцип використання ПП, який може приймати і відправляти повідомлення поодинці або більш каналам зв'язку, забезпечуючи, таким чином, застосуванням можливість взаємодії без необхідності знання конкретних деталей один про одного. Це стиль проектування, в якому взаємодії між окремими ПЗ здійснюються шляхом передачі (зазвичай асинхронною) повідомлень через загальну шину по загальних схемах і інфраструктурі.

Шина повідомлень забезпечує можливість обробляти:

* Взаємодію, засновану на повідомленнях. Вся взаємодія між застосуваннями грунтується на повідомленнях, що використовують відомі схеми.
* Складну логіку обробки. Складні операції можуть виконуватися як частина багатокрокового процесу шляхом поєднання ряду менших операцій, кожна з яких підтримує певні завдання.
* Зміни логіки обробки. Взаємодія з шиною реалізується по загальних схемах і із застосуванням звичайних команд, що забезпечує можливість вставки або видалення застосувань на шині для зміни використовуваної для обробки повідомлень логіки.
* Інтеграцію з різними інфраструктурами. Використання моделі зв'язку за допомогою повідомлень, заснованого на загальних стандартах, дозволяє взаємодіяти із застосуваннями, розробленими для різних інфраструктур, таких як Microsoft .NET і Java.

Шини повідомлень використовуються для забезпечення складних правил обробки. Такий дизайн забезпечує архітектуру, що підключається, яка дозволяє вводити застосування в процес або покращувати масштабованість, підключаючи до шини декілька екземплярів одного і того ж застосування.

Основні переваги архітектури шини повідомлень:

* Розширюваність. Можливість додавати або видаляти застосування з шини без впливу на існуючі застосування.
* Невисока складність. Застосування спрощуються, тому що кожному з них необхідно знати лише, як обмінюватися даними з шиною.
* Гнучкість. Приведення набору застосувань, складових складний процес, або схем зв'язку між застосуваннями у відповідність бизнес-требованиям, що змінюється, або вимогам користувача просто шляхом внесення змін до конфігурації або параметрів, керівники маршрутизацією.
* Слабке скріплення. Окрім інтерфейсу, що надається застосуванням, для зв'язку з шиною повідомлень, немає ніяких інших залежностей з самим застосуванням, що забезпечує можливість зміни, оновлення і заміни його іншим застосуванням, що надає такий же інтерфейс.
* Масштабованість. Можливість підключення до шини безлічі екземплярів одного застосування для забезпечення одночасної обробки безлічі запитів.
* Простота застосування. Не дивлячись на те, що реалізація шини повідомлень ускладнює інфраструктуру, кожному застосуванню доводиться підтримувати лише одне підключення до шини повідомлень, а не безліч підключень до інших застосувань.

Простим різновидом такої архітектури є конвеєр, засоби для організації якого є в операційній системі UNIX. Конвеєром є послідовність програм, в якій стандартне виведення кожної програми, окрім найостаннішої, пов'язане із стандартним введенням наступної програми цієї послідовності (див. мал. 6.2). Конвеєр обробляє деякий потік повідомлень. Кожне повідомлення цього потоку поступає на введення першій програмі, яка обробивши його передає перероблене повідомлення наступної програми, а сама починає обробку чергового повідомлення потоку. Таким же чином діє кожна програма конвеєра: отримавши повідомлення від попередньої програми і обробивши його, вона передає перероблене повідомлення наступної програми, а остання програма конвеєра виводить результат роботи всього конвеєра (результуюче повідомлення). Таким чином, в конвеєрі, що складається з n програм, може одночасно знаходитися в обробці до n повідомлень. Звичайно, внаслідок того, що різні програми конвеєра можуть витратити на обробку чергових повідомлень різні відрізки часу, необхідно забезпечити яким-небудь чином синхронізацію цих процесів (деякі процеси можуть знаходитися у стадії очікування або можливості передати перероблене повідомлення, або можливості отримати чергове повідомлення).

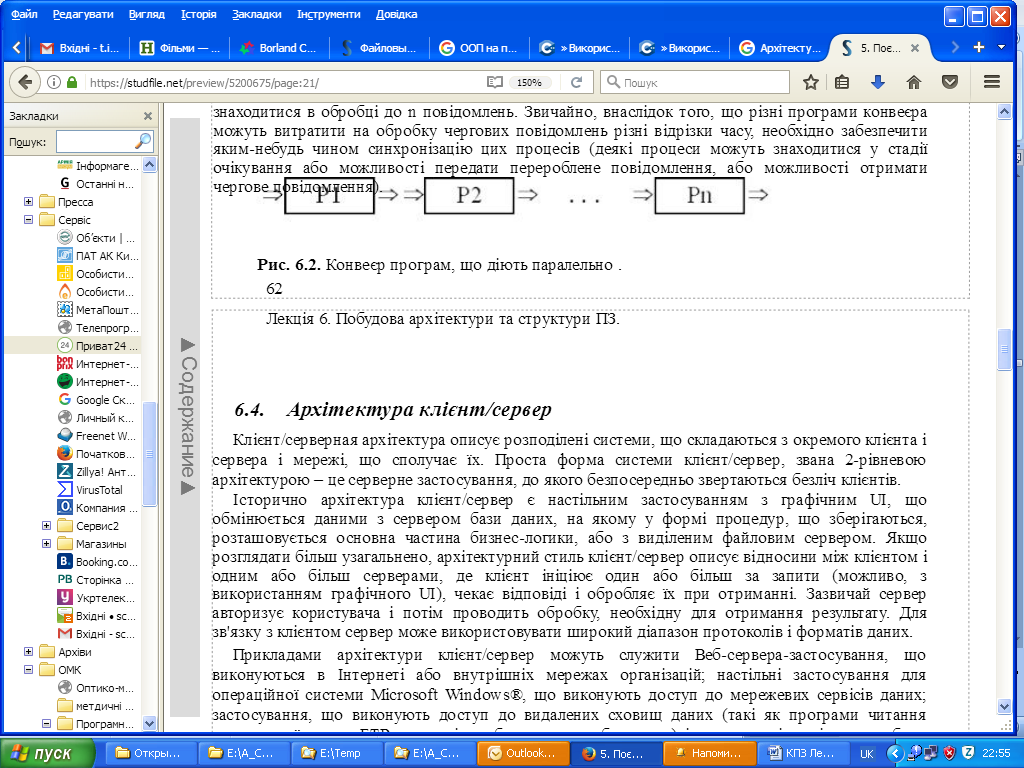


Рис. 2. Конвеєр програм, що діють паралельно .

## 6.4. Архітектура клієнт/сервер

Клієнт/серверная архітектура описує розподілені системи, що складаються з окремого клієнта і сервера і мережі, що сполучає їх. Проста форма системи клієнт/сервер, звана 2-рівневою архітектурою – це серверне застосування, до якого безпосередньо звертаються безліч клієнтів.

Історично архітектура клієнт/сервер є настільним застосуванням з графічним UI, що обмінюється даними з сервером бази даних, на якому у формі процедур, що зберігаються, розташовується основна частина бизнес-логики, або з виділеним файловим сервером. Якщо розглядати більш узагальнено, архітектурний стиль клієнт/сервер описує відносини між клієнтом і одним або більш серверами, де клієнт ініціює один або більш за запити (можливо, з використанням графічного UI), чекає відповіді і обробляє їх при отриманні. Зазвичай сервер авторизує користувача і потім проводить обробку, необхідну для отримання результату. Для зв'язку з клієнтом сервер може використовувати широкий діапазон протоколів і форматів даних.

Прикладами архітектури клієнт/сервер можуть служити Веб-сервера-застосування, що виконуються в Інтернеті або внутрішніх мережах організацій; настільні застосування для операційної системи Microsoft Windows®, що виконують доступ до мережевих сервісів даних; застосування, що виконують доступ до видалених сховищ даних (такі як програми читання електронної пошти, FTP-клиенты і засоби доступу до баз даних); інструменти і утиліти для роботи з видаленими системами (такі як засоби управління системою і засобу моніторингу мережі).

Основні переваги архітектурного стилю клієнт/сервер:

* Велика безпека. Всі дані зберігаються на сервері, який зазвичай забезпечує більший контроль безпеки, чим клієнтські комп'ютери.
* Централізований доступ до даних. Оскільки дані зберігаються тільки на сервері, адміністрування доступу до даних набагато простіше, ніж в інших архітектурах.
* Простота обслуговування. Ролі і відповідальність обчислювальної системи розподілені між декількома серверами, що взаємодіють один з одним по мережі. Завдяки цьому клієнт гарантовано залишається неінформованим і не схильним до впливу подій, що відбуваються з сервером (ремонт, оновлення або переміщення).

Рекомендується розглядати можливість застосування архітектури клієнт/сервер,

* якщо створюване застосування повинне розміщуватися на сервері і не повинно підтримувати безліч клієнтів;
* якщо створюються Веб- сервера-застосування, що надаються через Веб-сервер-браузер;
* якщо реалізуються бизнес-процессы, які використовуватимуться в рамках організації;
* або якщо створюються сервіси для використання іншими застосуваннями.

Архітектура клієнт/сервер добре підходить для централізації сховища даних, функції резервного копіювання і управління, або якщо ПП повинен підтримувати різні типи клієнтів і різні пристрої.

Традиційна 2-рівнева архітектура клієнт/сервер має багато недоліків:

1.тенденція тісного звязування даних і бізнес-логіки серверної частини ПЗ, що може мати негативний вплив на розширюваність і масштабованість системи вцілому

2.залежність від центрального сервера, що негативно позначається на надійності системи.

Для вирішення цих проблем архітектурний стиль клієнт/сервер був розвинений в більш універсальний 3-рівневий (або N-рівневий), в якому усунено деякі недоліки, властиві 2-рівневій архітектурі клієнт/сервер, і забезпечуються додаткові переваги.

***Контрольні запитання***.

1. Що таке архітектура ПЗ?

2. Що таке описова та розпорядча архітектура ПЗ?

3. Що таке компонент ПЗ?

4. Що таке архітектурна конфігурація або топологія?

5. Які принципи проектування впливають на архітектуру ПЗ?

6. Які класи архітектури Вам відомі?

***Рекомендована література***

1. Лавріщева К.М. Програмна інженерія. Електронний підручник: http://csc.knu.ua/uk/library/books/lavrishcheva-6.pdf
2. Лавріщева К.М. Програмна інженерія. – Підручник.–К.:Академперіодика, 2008.–415с.
3. И. Соммервиль. Инженерия программного обеспечения,  
    6 изд. – И.д. "Вильямс", 2002.
4. Якобсон А. Унифицированный процесс разработки программного обеспечения / А.Якобсон, Г.Буч, Д.Рамбо. –СПб.: Питер, 2002. – 496 с.

***Для самостійного вивчення***: Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

1. Кеш — це швидка проміжна (буферна) [пам'ять](https://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%27%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0_%D0%BF%D0%B0%D0%BC%27%D1%8F%D1%82%D1%8C) невеликої місткості, що розташована поміж процесором і основною пам'яттю. Операції обміну даними між процесором і кешем виконуються швидше, ніж звернення процесора безпосередньо до основної пам'яті, тим самим прискорюється робота процесора з пам'яттю. [↑](#footnote-ref-1)