

**ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ  
УНІВЕРСИТЕТ ІМЕНІ ІВАНА ПУЛЮЯ**

**Крос-платформне програмування**

Освітньо-кваліфікаційний рівень – «Бакалавр»

Галузь знань “ **Інформатика та обчислювальна техніка**”  
Напрямок підготовки – **6.050101 «Комп’ютерні науки»**

Конспект лекцій

**ТЕРНОПІЛЬ - 2012**

Розроблено кафедрою комп'ютерних наук ТНТУ відповідно до освітньо-професійної програми, освітньо-кваліфікаційної характеристики та навчального плану підготовки бакалаврів з галузі знань “Інформатика та обчислювальна техніка”, напряму підготовки 6.050101 “Комп'ютерні науки”

Укладачі: к.т.н., доц. Р.О. Козак, асист. Т.В. Михайлович

## ЛЕКЦІЯ 1. ВИЗНАЧЕННЯ ТА ВЛАСТИВОСТІ КОМПОНЕНТІВ

### План

- 1.1 Поняття крос-платформності, її типи
- 1.2 Визначення та властивості компонентів.
- 1.3 Компонентна модель .Net Framework. Типи компонентів
- 1.4 Динамічна бібліотека DLL як приклад компонента
- 1.5 Висновки

### 1.1 Поняття крос-платформності, її типи

#### Означення 1

*Крос-платформне програмне забезпечення* — програмне забезпечення, що працює більш ніж на одній апаратній платформі і операційній системі (ОС).

#### Означення 2

*Крос-платформне програмування* — технологія створення і інтеграції в єдину систему компонентів, які розроблені на різних платформах.

#### Рівні кросплатформності

Поняття кросплатформності може використовуватися на різних рівнях абстракції інформаційних систем:

#### 1. На рівні мови програмування

Крос-платформними можна назвати більшість сучасних мов програмування високого рівня.

Наприклад, C, C++ і Object Pascal — крос-платформні мови на рівні компіляції, тобто для цих мов є компілятори під різні платформи.

Java і C# — крос-платформні мови на рівні виконання, тобто їх виконувані файли можна запускати на різних платформах без попередньої перекомпіляції. Це забезпечується двох-етапною компіляцією через *проміжний код*. В Java для цього використовується *байт-код* і *віртуальна машина* (JRE), реалізація якої є для різних ОС, а в C# — через проміжний код на проміжній мові програмування (близькій до мови *асемблера*) і загальномовного середовища програмування (CLR – *Common Language Runtime*). Нагадаємо, що CLR – це динамічна складова .Net Framework.

Реалізація .Net Framework є для всіх версій Windows. Реалізація для платформи Linux – проект MONO.

Мови скриптів – PHP, ActionScript, Perl, Python, Tcl і Ruby — кросплатформні мови, що інтерпретуються, їх інтерпретатори існують для багатьох платформ.

#### 2. На рівні прикладних програм

Багато *прикладних програм* також є крос-платформними. Особливо ця якість виражена в програмах, спочатку розроблених для UNIX-подібних операційних систем. Важливою умовою їх переносності на інші платформи

є сумісність платформ з рекомендаціями POSIX, а також існування компілятора для платформи, на яку здійснюється перенесення.

***Приклади:***

- [Apache](#)
- [BinkD](#)
- [CVS](#)
- [Emacs](#)
- [GIMP](#)
- [GoldEd](#)
- [Inkscape](#)
- [Lotus Notes](#)
- [Mozilla Firefox](#), [Mozilla Thunderbird](#), [SeaMonkey](#)
- [MySQL](#)
- [OpenOffice.org](#)
- [Opera](#)
- [VIM](#)

### **3. На рівні операційної системи**

Сучасні *операційні системи* також часто є крос-платформними. Наприклад, операційні системи з відкритим кодом, такі як NetBSD, Linux, FreeBSD, AROS можуть працювати на декількох різних платформах, найчастіше це x86, ARM, MIPS, m68k, PowerPC, Alpha, AMD64, SPARC. Microsoft Windows може працювати як на платформі Intel x86, так і на Intel Itanium. Операційна система NetBSD найбільш переносимою, вона портована на більшість існуючих платформ.

***Емуляція***

Якщо програма не призначена для виконання (запуску) на певній платформі, але для цієї платформи існує *емулятор* платформи, базової для даної програми, то програма може бути виконана в середовищі емулятора.

Звичайне виконання програми в середовищі емулятора призводить до *зниження продуктивності* в порівнянні з аналогічними програмами, для яких платформа є базовою, оскільки значна частина ресурсів системи витрачається на виконання функцій емулятора.

### **1.2 Визначення та властивості компонентів**

Сучасні кросплатформні системи створюються з використанням *компонентно-орієнтованого підходу*.

*Компонентне програмування* є подальшим розвитком об'єктно-орієнтованого програмування (ООП) і концепції «повторного використання» (reuse).

Компонентне програмування покликане забезпечити простішу і швидшу процедуру інсталяції прикладного програмного забезпечення, а

також збільшити відсоток повторного використання коду, тобто підсилити основні переваги ООП.

Перш за все, сформулюємо головне для даного підходу визначення компонента.

Під *компонентом* далі матимемо на увазі незалежний модуль програмного коду, призначений для повторного використання і розгортання.

Такий компонент є структурною одиницею програмної системи, що має чітко визначений *інтерфейс*.

Компонент може бути незалежно поставлений або не поставлений, доданий до складу деякої системи або видалений з неї, у тому числі, може включатися до складу систем інших постачальників.

Таким чином, *компонент* — це виділена структурна одиниця розгортання з чітко визначеним інтерфейсом.

Всі залежності компонента від оточення мають бути описані в рамках цього інтерфейсу. Один компонент може також мати декілька інтерфейсів, граючи декілька різних ролей в системі. При описі інтерфейсу компонента важлива не лише сигнатура операцій, які можна виконувати з його допомогою. Стає важливим і те, які інші компоненти він може використовувати при роботі, а також яким обмеженням повинні задовольняти вхідні дані операцій і які властивості виконуються для результатів їх роботи. Ці обмеження є так званим *інтерфейсним контрактом* або *програмним контрактом компонента*.

#### **Властивості компонентів**

**1. Використання компонента (виклик його методів) можливе лише через його інтерфейси відповідно до контракту.**

Інтерфейс компонента включає набір операцій, які можна викликати в будь-якого компонента, який реалізує даний інтерфейс, і набір операцій, які цей компонент може викликати у відповідь в інших компонентах. *Інтерфейсний контракт* для кожної операції самого компонента (або використовуваного ним) визначає правила взаємодії компонентів в системі.

Для опису інтерфейсів призначені мови опису інтерфейсів IDL (Interface Definition Language).

Ці мови містять *операції*, які є викликами відкритих (public) методів класів, що входять до складу компонента, і *операнди* – відкриті (public) поля класів.

**2. Компонент повинен працювати в будь-якому середовищі, де є необхідні для його роботи інші компоненти.**

Це вимагає наявності певної інфраструктури, яка дозволяє компонентам знаходити один одного і взаємодіяти по певних правилах.

Набір правил визначення інтерфейсів компонентів і їх реалізацій, а також правил, за якими компоненти працюють в системі і взаємодіють один з одним, прийнято називати *компонентною моделлю* (component model).

У компонентну модель входять правила, що регламентують життєвий цикл компонента, тобто те, через які стани він проходить при своєму існуванні в рамках деякої системи (незавантажений, завантажений, пасивний, активний, знаходиться в кеші і ін.) і як виконуються переходи між цими станами).

Існують декілька компонентних моделей в різних ОС і від різних виробників. Правильно взаємодіяти один з одним можуть лише компоненти, побудовані в рамках *однієї моделі*, оскільки компонентна модель визначає «мову», на якій компоненти можуть спілкуватися один з одним.

Окрім компонентної моделі, для роботи компонентів необхідний деякий набір базових служб (basic services). Наприклад, компоненти повинні уміти знаходити один одного в середовищі, яке, можливо, розподілене на декілька машин. Компоненти повинні уміти передавати один одному дані, знову ж таки, можливо, за допомогою мережових взаємодій, але реалізації окремих компонентів самі по собі не повинні залежати від вигляду використовуваного зв'язку і від розташування їх партнерів по взаємодії. Набір таких базових, необхідних для функціонування більшості компонентів служб, разом з підтримуваною з їх допомогою компонентною моделлю називається компонентним середовищем (або компонентним каркасом, component framework). Приклади відомих компонентних середовищ — різні реалізації J2EE, .NET Framework, CORBA, COM, COM+, DCOM.

### ***3. Компоненти відрізняються від класів об'єктно-орієнтованих мов.***

Клас визначає не лише набір інтерфейсів, що реалізуються, але і саму їх реалізацію, оскільки він містить код визначуваних операцій. Контракт компонента, найчастіше, не фіксує реалізацію його інтерфейсів. Клас написаний на певній мові програмування. Компонент же не прив'язаний до певної мови, якщо, звичайно, його компонентна модель цього не вимагає, — компонентна модель є для компонентів тим же, чим для класів є мова програмування.

Зазвичай компонент є *крупнішою структурною одиницею*, ніж клас. Реалізація компонента часто складається з декількох тісно зв'язаних один з одним класів.

### ***4. Компоненти не залежать від мов програмування.***

Компоненти зберігаються і поширюються у бінарному вигляді і не залежать від мов програмування програмної системи.

Користувач і постачальник компонента можуть використовувати різні мови і бути територіально розподіленими.

### 1.3 Компонента модель .Net Framework. Типи компонентів

#### *Поняття платформи MS.NET*

Під платформою Microsoft.NET слід розуміти інтегровану систему (*інфраструктуру*) засобів розробки, розгортання і виконання складних (як правило, розподілених) програмних систем.

Основа .Net – це Microsoft .Net Framework – компонентний каркас, набір засобів і технологій для розробки і виконання програмних систем.

#### *Ключові характеристики MS.NET*

11. Сучасні засоби розробки – платформа MS.Net включає як готові компоненти для побудови ПЗ, так й інтегроване середовище розробки, яке забезпечує можливість *багатомовної розробки* ПЗ з використанням різних мов програмування (C#, C++, VB.Net). Як результат, розробник програм вже не обмежується вибором однієї якої-небудь мови програмування, а може варіювати засоби розробки з урахуванням власного досвіду і властивостей програм, що розробляються, навіть в межах однієї програмної системи.

2. Компонентне представлення ПЗ – MS.Net розвиває існуючі підходи до основного способу зниження складності ПЗ - *компонентному представленню програмних систем* - пропонуючи більш простий, зручний і надійний метод формування програмних компонентів.

Корпорацією Microsoft запропонована реалізація *компонентно-орієнтованого підходу* до проектування і реалізації застосунків, який є розвитком об'єктно-орієнтованого підходу. Згідно цього підходу, інтеграція об'єктів, виконується на основі *інтерфейсів*, що представляють ці об'єкти (або фрагменти програм) як незалежні компоненти. Такий підхід істотно полегшує написання і взаємодію програмних компонентів в гетерогенному середовищі проектування і реалізації.

Для інсталяції на комп'ютери користувачів програми створюються інсталяційні комплекти у формі *збірок*.

Кожний тип збірки характеризується унікальним ідентифікатором – *номером версії збірки*. Таким чином, кожний програмний проект формується у вигляді *збірки*, яка є самодостатнім компонентом для розгортання, тиражування і повторного використання.

Між збірками і просторами імен існує наступне співвідношення. Збірка може містити декілька просторів імен. В той же час, простір імен може займати декілька збірок.

Збірка може мати в своєму складі як один, так і декілька файлів, які об'єднуються у складі *маніфесту* або опису збірки, який на звичній нам природній мові аналогічний змісту книги. Маніфест містить метадані про компоненти збірки, ідентифікацію автора і версії, відомості про типи і

залежність, а також режим і політику використання збірки. Метадані типів маніфесту повною мірою описують всі типи, які описані в збірці.

В результаті компіляції програмного коду в середовищі обчислень .NET створюється або *збірка*, або так званий *модуль*. При цьому збірка існує у формі виконуваного *файлу* (з розширенням EXE), або файлу *динамічної бібліотеки* (з розширенням DLL). Природно, до складу збірки входить *маніфест*. Модуль є файлом з розширенням NETMODULE. Він не містить маніфесту.

#### **Приклад 1.** Маніфест збірки

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1"
manifestVersion="1.0">
  <assemblyIdentity version="1.0.0.0"
name="MyApplication.app"/>
  <trustInfo xmlns="urn:schemas-microsoft-
com:asm.v2">
    <security>
      <requestedPrivileges xmlns="urn:schemas-
microsoft-com:asm.v3">
        <requestedExecutionLevel level="asInvoker"
uiAccess="false"/>
      </requestedPrivileges>
    </security>
  </trustInfo>
</assembly>
```

### **1.4 Динамічна бібліотека DLL як приклад компонента**

*Динамічна бібліотека* — набір функцій, скомпонованих разом у вигляді бінарного файлу, який може бути динамічно завантажений в адресний простір процесу, що використовує ці функції. *Динамічне завантаження* (dynamic loading) — завантаження під час виконання процесу.

Оскільки динамічні бібліотеки є двійковими файлами, можна організувати спільну роботу бібліотек, розроблених із використанням різних мов програмування і програмних засобів, що спрощує створення застосунків на основі програмних компонентів (отже, динамічне компонування лежить в основі компонентного підходу до розробки програмного забезпечення).

Основна мета створення і використання DLL-бібліотек — забезпечення повторного використання коду. Тому в реальних системах різні компоненти можуть викликати однакові бібліотечні методи. Для цього DLL-бібліотека повинна знаходитися в окремому Рішенні (Solution).



Для підключення бібліотеки до проекту файл зі збіркою, що містить бібліотеку потрібно підключити з меню «Add Reference». Оскільки проект не включений в Рішення, то у вікні додавання посилань потрібно вказати шлях до проекту у файловій системі.

## 1.5 Висновки

Важливим наслідком реалізації компонентного підходу є зниження вартості проектування ПЗ.

До переваг компонентного програмування слід віднести і можливість удосконалення стратегії повторного використання коду.

В платформі Microsoft .NET реалізовано компонентно-орієнтований підхід до програмування. Цей підхід до проектування і реалізації програмних систем і комплексів є розвитком об'єктно-орієнтованого і практично придатніший для розробки великих і розподілених систем.

В компонентній моделі .Net компонентом є збірка, яка може бути у вигляді *виконуваного файлу* (з розширенням EXE), або файлу *динамічної бібліотеки* (з розширенням DLL). До складу збірки входить *маніфест*.

Збірка є самодостатньою одиницею для розгортання, тиражування та повторного використання.

Збірка має маніфест, який містить інформацію про збірку (метадані, які описують збірку).

Велику роль в сучасних системах відіграють бібліотеки повторного використання, зокрема DLL-бібліотеки.

DLL-бібліотеки містять методи, які викликаються динамічно при роботі програмної системи. Вони є окремими компонентами з розширенням .dll. Інтерфейсом бібліотеки є відкриті (public) методи і поля бібліотечних класів. DLL-бібліотека не містить точки входу (main), тому її методи можуть викликатися лише іншими компонентами. Разом з цим, в бібліотеці можуть бути і закриті методи класу, які викликаються методами в середині бібліотеки. DLL-бібліотеки – основа компонентної ідеології і повторного використання коду.

### Контрольні питання

1. Що таке інтерфейс компонента і яке його призначення?
2. Що таке компонент, чим він відрізняється від класу?
3. Що визначає інтерфейсний контракт? Що являє собою інтерфейс компонента?
4. Основні властивості компонентів?
5. Рівні кросплатформності і різниця між ними?
6. Для чого в .Net використовується компіляція через проміжний код?
7. Для чого використовується компонентний каркас?
8. Що таке компонентна модель і яке її призначення?

9. Що таке DLL-бібліотека? Чим вона відрізняється від консольного проекту?
10. Для чого потрібно XML-документування коду? Як воно допомагає при написанні виклику бібліотечних методів?
11. Чому бібліотечні методи повинні визначатися з модифікатором доступу public?
12. Що є інтерфейсним контрактом бібліотеки?

## ЛЕКЦІЯ 2. СПЕЦИФІКАЦІЯ ІНТЕРФЕЙСУ ЯК КОНТРАКТУ

### План

#### 2.1 Контрактне програмування

#### 2.2 Контракт в об'єктно-орієнтованому програмуванні.

### 2.1 Контрактне програмування

**Контрактне програмування** (*design by contract (DbC)*, *programming by contract*, *contract-based programming*) – це метод проектування програмного забезпечення. Він припускає, що проектувальник повинен визначити формальні, точні і верифіковані специфікації інтерфейсів для компонентів системи. При цьому, крім звичайного визначення абстрактних типів даних, також використовуються *передумови*, *післяумови* та *інваріанти*. Дані специфікації називаються "контрактами" відповідно до концептуальної метафорою умов і відповідальності в цивільно-правових договорах.

Основна ідея контрактного програмування – це модель взаємодії елементів програмної системи, яка ґрунтується на ідеї взаємних зобов'язань і переваг. Як і в бізнесі, *клієнт* і *постачальник* діють у відповідності з певним *контрактом*. Контракт деякого методу чи функції може включати в себе:

- конкретні зобов'язання, які будь клієнтський модуль повинен виконати перед викликом методу – *передумови*, які дають перевагу для постачальника – він може не перевіряти виконання передумов;
- конкретні властивості, які повинні бути присутніми після виконання методу – *післяумови*, які входять в зобов'язання постачальника;
- зобов'язання по виконанню конкретних властивостей – *інваріантів*, які повинні виконуватися при отриманні постачальником повідомлення, а також при виході з методу.

Багато мов програмування дозволяють враховувати такі зобов'язання. Контрактне програмування ставить ці вимоги критичними для коректності програм, тому вони повинні бути затверджені при проектуванні. Таким чином, контрактне програмування передбачає початок писання коду із написання формальних тверджень коректності (assertions).

### 2.2 Контракт в об'єктно-орієнтованому програмуванні

В об'єктно-орієнтованому програмуванні контракт методу зазвичай включає наступну інформацію:

- можливі типи вхідних даних та їх значення;
- типи повернутих даних та їх значення;
- умови виникнення виключень, їх типи та значення;
- присутність побічного ефекту методу;

- передумови, які можуть бути ослаблені (але не посилені) у підкласах;
- післяумови, які можуть бути посилені (але не ослаблені) у підкласах;
- інваріанти, які можуть бути посилені (але не ослаблені) у підкласах;
- (іноді) гарантії продуктивності, наприклад, тимчасова складність або складність по пам'яті.

При використанні контрактів сам код не зобов'язаний перевіряти їх виконання. Зазвичай в таких випадках в коді роблять *жорстке падіння* («fail hard»), таким чином полегшуючи налагодження виконання контрактів. У багатьох мовах, таких як [C](#), [C++](#), [Delphi](#), [PHP](#), така поведінка реалізується оператором `assert`. У реліз-варіанті коду ця поведінка може бути збережена, або перевірки можуть бути прибрані щоб підвищити продуктивність.

Юніт-тести перевіряють модуль ізольовано від інших, перевіряючи, що модуль задовольняє припущеннями контракту, а також свої контракти виконують використовувані ним модулі. Інтеграційні тести перевіряють, що модулі працюють коректно разом.

Контрактне програмування може підвищити рівень повторного використання коду, оскільки зобов'язання модуля чітко задокументовані. Взагалі, контракт модуля можна розглядати також як спосіб документації програмного забезпечення.

Семантика інтерфейсу компонента може бути представлена за допомогою контрактів, що визначають зовнішні обмеження і підтримують інваріант, який містить у собі правила встановлення взаємозв'язків властивостей компонента або умови його життєздатності. Крім того, для кожної операції компонента контракт може визначати обмеження, що повинні бути враховані клієнтом перед викликом операції (передумова), і постумови перевірки правильності функціонування компонента після завершення операції. Перед- і післяумова визначають специфікацію поведінки компонента і залежать від стану компонента, а також інтерфейсу і зв'язаним з ним набором інваріантів. Контракти й інтерфейс зв'язані між собою, але їхні сутності різні. Інтерфейс являє собою колекцію операцій або функціональних властивостей специфікації сервісів, що підтримує компонент. Контракт задає опис поведінки компонента, націлений на взаємодію з іншими компонентами і відбиває семантику функціональних властивостей компонента.

### Контрольні питання

1. Що визначає інтерфейсний контракт? Що являє собою інтерфейс компонента?
2. Що є інтерфейсним контрактом бібліотеки?
3. Чим є контракт методу в об'єктно-орієнтованому програмуванні?
4. Що таке контрактне програмування?
5. Яка основна ідея контрактного програмування?
6. Яку інформацію включає контракт в об'єктно-орієнтованому програмуванні?

7. Для чого призначені [Юніт-тести](#)?

### ЛЕКЦІЯ 3. МОДЕЛЬ ПОСИЛАНЬ. СТРАТЕГІЇ ІНТЕГРАЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

#### План

3.1 Узагальнена модель компонентної системи.

3.2 Особливості компонентної ідеології. Інтегровність компонентів.

#### 3.1 Узагальнена модель компонентної системи

Компонентна програмна система складається з незалежних компонентів. Узагальнена модель такої системи зображена на рисунку 1.1.

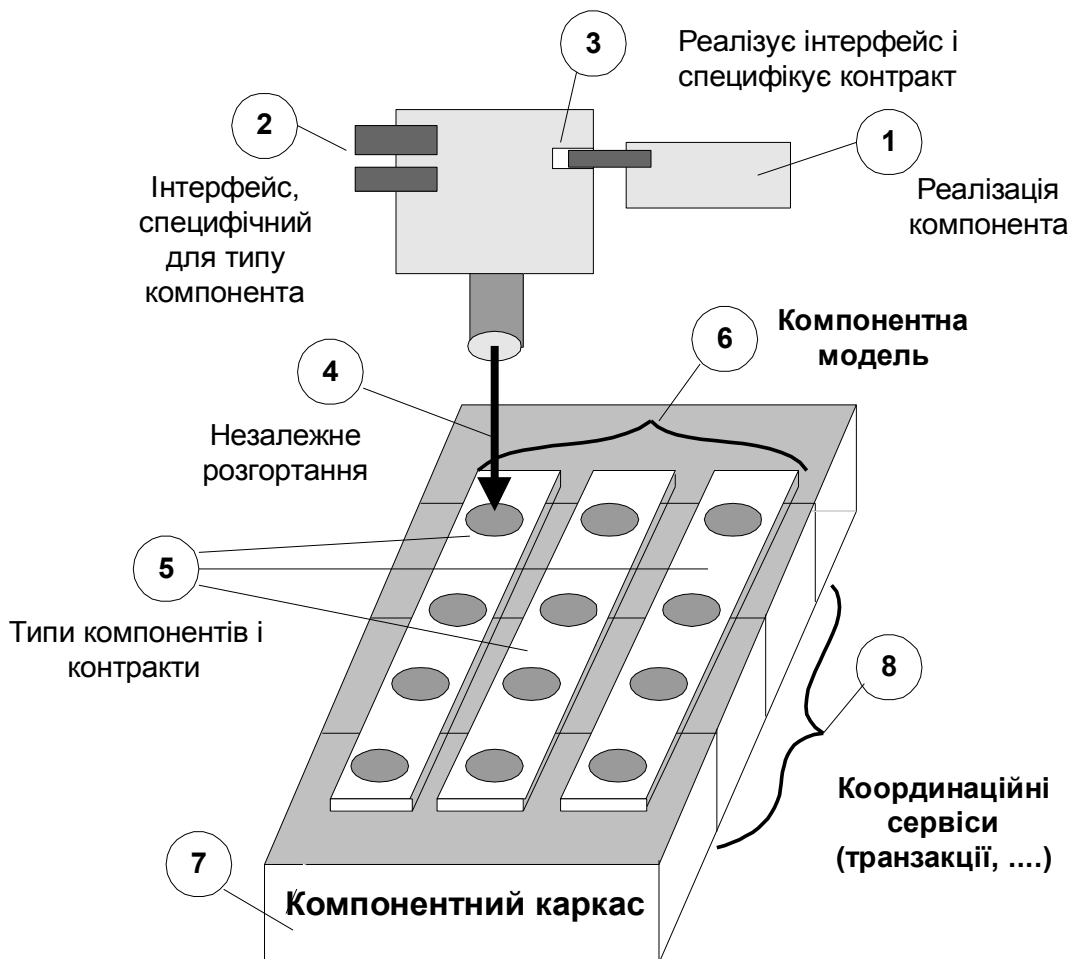


Рисунок 1.1 – Узагальнена модель компонентної системи

**Компонент** (1) – це програмна реалізація (виконуваний код) функцій об'єкту, призначена для виконання. Разом з функціональністю компонент реалізує один або декілька інтерфейсів (2) відповідно до певних зобов'язань, описаних у контракті (3). Контрактні зобов'язання гарантують, що незалежно розроблені компоненти можуть взаємодіяти один з одним і розгортатися в стандартних середовищах (4) (на етапі побудови системи або на етапі її виконання).

Компонентна програмна система ґрунтується на невеликій кількості *компонентів* різних *типів* (5), кожний з яких має спеціалізовану роль в системі і описується інтерфейсом (2). *Компонентна модель* (6) утворена множиною типів компонентів, їх інтерфейсів, а також специфікацією допустимих *шаблонів (паттернів) взаємодії* типів компонентів. *Компонентний каркас* (7) забезпечує множину *сервісів* (8) для підтримки функціонування компонентної моделі. У багатьох відношеннях компонентні каркаси подібні спеціалізованим операційним системам, хоча вони оперують на більш високих рівнях абстракції і мають більш обмежений діапазон механізмів координації взаємодії компонентів.

### **3.2 Особливості компонентної ідеології. Інтегровність компонентів**

Найбільш важливий аспект компонентної ідеології побудови – передбачуваність *композиції* взаємодіючих компонентів і каркасів. Можливі три основні види композицій в програмній системі:

- «компонент–компонент» (взаємодія по контрактах *прикладного* рівня, яка визначається під час розробки (*раннє зв'язування*) або під час виконання (*пізнє зв'язування*);
- «каркас–компонент» (взаємодія між компонентом і іншими компонентами каркаса по контрактах *системного* рівня із забезпеченням управління ресурсами);
- «каркас–каркас» (взаємодія між компонентами, які розгортаються в гетерогенних середовищах (каркасах) по контрактах інтероперабельного (interoperation) рівня).

*Компонент* виступає в двох ролях – як *реалізація* (що розробляється, розгортається і інтегрується в систему) і як *архітектурна абстракція* (що визначає правила проектування, встановлені стандартною моделлю для всіх компонентів).

Компоненти розробляються у вигляді деякої програмної абстракції, що включає:

*інформаційну частину* – опис призначення, дати виготовлення, умов застосування (ОС, платформа тощо) та можливостей повторного використання;

*зовнішню частину* – інтерфейси, які визначають взаємодію компоненту із зовнішнім середовищем і з платформами, на яких він буде працювати, і забезпечують такі загальні характеристики компоненту:

- інтероперабельність – здатність взаємодіяти з компонентами інших середовищ;
- переносність (мобільність) – здатність працювати на різних платформах;
- інтегрованість – здатність до об'єднання з іншими компонентами в складі програмної системи;

– не функціональні характеристики – безпека, надійність і захист компоненту і даних;

*внутрішню частину* – фрагмент програмного коду або абстрактну структуру - проект компоненту, його специфікацію і початковий код – які реалізують інтерфейси компоненту, його функціональність і схеми розгортання.

*Специфікація інтерфейсу* може виконуватися засобами API (Application Programming Interface) мови програмування або на мові специфікації інтерфейсу (не залежному від мови програмування) - Interface Definition Language (IDL).

Сучасні мови програмування, наприклад, Java, мають розширення, спеціально призначені для специфікації поведінки: iContract, JML (Java Modeling Language), Jass (Java with assertions), Biscotti, and JINSLA (Java INterface Specification LAnguage).

Таким чином, компонентні системи ґрунтуються на чітко визначених *стандартах і угодах* для розробників компонентів (встановлених у компонентній моделі) та *інфраструктурі* компонентів (компонентному каркасі), яка реалізує сервіси для моделі, спрощуючи розгортання окремих компонентів і застосунків.

*Компонентна модель* пропонує:

- стандарти по *типах компонентів* (наприклад, проекти, форми, COBRA-компоненти, RMI-компоненти, стандартні класи-оболонки, бази даних, JSP-компоненти, сервлети, XML-документи, DTD-документи і т.п., які визначені у відповідних мовах програмування);

- стандарти *схем взаємодії* (методи локалізації, протоколи комунікації, необхідні атрибути якості взаємодії – безпека, управління транзакціями тощо);

- угоди про *зв'язування* компонентів з ресурсами (сервісами, що надаються каркасом або іншим компонентом, розгорненим у цьому каркасі). Модель описує, які ресурси доступні компонентам, як і коли компоненти зв'язуються з цими ресурсами. Каркас, у свою чергу, розглядає компоненти як ресурси, що підлягають управлінню.

**Найбільш відомі компонентні моделі** – COM (Component Object Model) (DCOM – розподілена компонентна модель, COM+), CORBA, .Net Framework.

1. **Component Object Model (COM)** – початковий стандарт Microsoft для компонентів. Визначає протокол для конкретизації і використання компонент усередині процесу, між процесами або між комп'ютерами. Основа для ActiveX, OLE і багатьох інших технологій. Може використовуватися в Visual Basic, C++ і ін.
2. **Java Beans** – стандарт Sun Microsystems для компонентів (тільки для Java).

3. **CORBA** (стандарт OMG, має громіздкий IDL-інтерфейс, складність відображення однієї мови реалізації в іншу).
4. **Dot Net Framework** – сучасна компонента модель Microsoft для розробки складних розподілених програмних систем.

*Компонентний каркас* (подібно операційній системі, об'єкти дії якої – компоненти) керує ресурсами, розподіленими компонентами, і надає механізми для організації взаємодії компонентів. Каркас необов'язково існує окремо від компонентів під час роботи системи, його реалізація може бути об'єднана з реалізацією компоненту, хоча переважно перше, як, наприклад, каркас для підтримки компонентної моделі EJB (Enterprise JavaBeans).

Основна ідея *компонентної ідеології* – розповсюдження класів в *бінарному вигляді* (тобто не у вигляді початкового коду) і надання доступу до методів класу через чітко визначені *інтерфейси*, що дозволяє зняти проблему несумісності компіляторів і забезпечує зміну версій класів без перекомпіляції компонентів.

### **Контрольні питання**

1. Що таке модель посилань (узагальнена модель компонентної системи)?
2. В чому полягає інтегровність компонентної системи?
3. Основна ідея компонентної ідеології.
4. На чому ґрунтується компонентна програмна система ?



## ЛЕКЦІЇ 4,5. РОЗРОБКА ТА ЗБИРАННЯ КОМПОНЕНТІВ. ОБ'ЄКТИ ТА СЕРВІСИ, ЩО НИМИ НАДАЮТЬСЯ

### План

- 4.1 Шаблон створення компонентів для Windows Forms
- 4.2 Структура вікна Visual Studio при створенні застосунків
  - 4.2.1 Вікно проекту
  - 4.2.2 Меню і панель інструментів
  - 4.2.3 Properties Explorer – вікно властивостей об'єктів
  - 4.2.4 Вікно компонентів: Toolbox
  - 4.2.5 Властивості проекту
- 5.1 Типи форм (модальні і не модальні).
  - 5.1.1 Типи форм
  - 5.1.2 Властивості форми
  - 5.1.3 Методи форми
  - 5.1.4 Події форми
- 5.2 Розміщення на формі елементів керування (Controls)
  - 5.2.1 Ієрархія класів Control
  - 5.2.2 Заголовки (Label) і текстові поля (TextBox)
  - 5.2.3 Компонент NumericUpDown – регулятор чисел
  - 5.2.4 Контейнери елементів

### 4.1 Шаблони створення компонентів для Windows Forms

Windows-застосунок в термінології .Net це рішення (Solution). Для компілятора – це *збірка* (Assembly). Рішення може складатися з одного або кількох проектів. Кожний проект може складатися з кількох форм і інших файлів, рисунків, ресурсів, маніфесту (опису збірки).

Відкомпільована збірка є готовим до використання *компонентом*.

Для створення Windows-застосунку в середовищі Visual Studio потрібно вибрати тип шаблону – Windows Application, вказати назву проекту та шлях у файловій системі, де буде зберігатися проект.

Класи для розробки Windows застосунків розміщені в просторі імен Windows.Forms.

Для підключення цього простору при створенні проекту згідно шаблону автоматично добавляється директива імпорту:

```
using System.Windows.Forms;
```

Графічний інтерфейс користувача у Windows-застосунку – це сукупність форм, на яких розміщені елементи керування (Controls) та *готові компоненти*. Керування роботою програми виконується за допомогою меню, панелі інструментів та програмних кнопок.

**Форма** — це клас, який задається *властивостями*, що визначають його зовнішній вигляд, *методами*, які визначають його поведінку, і *подіями*, що визначають взаємодію з користувачем.

При створенні проекту автоматично створюється порожня форма. Це стандартний шаблон нової програми Windows Forms.

На відміну від консольного застосунку, код Windows форми в C# розміщений у двох файлах, *Form1.cs* і *Form1.designer.cs*.

Це так звані *неповні* (*partial*) класи. Неповні класи можуть бути розміщені у декількох файлах. При компіляції виконується формування єдиного класу.

У файлі *Form1.designer.cs* знаходиться код, який генерується дизайнером форми. Він містить конструктори форми і всіх елементів форми.

**Наприклад,**

```
namespace MyPrg
{
    partial class Form1
    {
        #region Windows Form Designer generated code
        private void InitializeComponent()
        {
            this.components = new
System.ComponentModel.Container();
            this.AutoScaleMode =
System.Windows.Forms.AutoScaleMode.Font;
            this.Text = "Form1";
        }
        #endregion
    }
}
```

У файлі *Form1.cs* міститься код ініціалізації класу і члени класу (поля, властивості, методи, обробники подій). Саме з цим класом працює програміст.

Крім цих двох файлів автоматично створюється статичний клас *Program* у файлі (*Program.cs*), який містить точку входу в програму.

```
using System;
using System.Windows.Forms;
namespace MyPrg
{
    static class Program
    {
        static void Main()
        {
```

```

        Application.Run(new Form1());
    }
}

```

Метод `Main` є точкою входу для застосунку і викликає метод `Application.Run`, який створює клас `Form1`.

Клас `Application` – головний клас нашого застосунку. Він запускає цикл обробки повідомлень для нашого застосунку.

Це прихований (Sealed) клас, його наслідування неможливе. Для керування роботою ми можемо використовувати методи цього класу, у нашому прикладі це метод `Run`.

**Примітка 1.** Якщо змінити назву форми з `Form1` на іншу у вікні властивостей, то потрібно змінити цю назву і в методі `Application.Run(new Form1());`

Крім того, до складу кожного застосунку входить файл з інформацією про збірку (`Assembly.info`).

Клас `System.Windows.Forms.Application`

Клас `Application` можна розглядати як "клас низького рівня", що дозволяє нам керувати поведінкою застосунку `Windows Forms`. Крім того, цей клас визначає набір подій на рівні всього застосунку, наприклад його закриття або простій центрального процесора.

Таблиця 4.1 – Найбільш важливі методи класу `Application` (всі вони є статичними)

Метод класу	Призначення
<code>AddMessageFilter()</code>	Ці методи дозволяють перехоплювати повідомлення <code>RemoveMessageFilter()</code> і виконувати з цими повідомленнями необхідні попередні дії. Для того щоб додати фільтр повідомлень, необхідно вказати клас, що реалізує інтерфейс <code>IMessageFilter</code>
<code>DoEvents()</code>	Забезпечує здатність застосунку обробляти повідомлення з черги повідомлень під час виконання якої-небудь тривалої операції. Можна сказати, що <code>DoEvents()</code> – це "швидкий" заміник нормальної багатопоточності
<code>Exit()</code>	Завершує роботу застосунку
<code>ExitThread()</code>	Припиняє обробку повідомлень для даного потоку і закриває всі вікна, власником яких є цей потік

Клас `Application` визначає багато статичних властивостей, більшість з яких доступні тільки для читання.

Таблиця 4.2 – Основні властивості класу `Application`

Властивість	Призначення
CommonAppDataRegistry	Повертає параметр системного реєстру, який зберігає загальну для всіх користувачів інформацію про застосунок
CompanyName	Повертає ім'я компанії
CurrentCulture	Дозволяє задати або отримати інформацію про поточну мову, для роботи з якою призначений поточний потік
CurrentInputLanguage	Дозволяє задати або отримати інформацію про мову для введення інформації, одержуваної поточним потоком
ProductName	Для отримання імені програмного продукту, яке асоційоване з даним застосунком
ProductVersion	Дозволяє отримати номер версії програмного продукту
StartupPath	Дозволяє визначити ім'я виконуваного файлу для працюючого застосунку і шлях до нього в операційній системі

## 4.2 Структура вікна Visual Studio при створенні застосунків

### 4.2.1 Вікно проекту

Вікно *Solution Explorer* – відображає дерево проекту і дозволяє управляти компонентами проекту. Наприклад, для того щоб додати в нього нову форму, просто виберіть в контекстному меню, що відкривається по клацанню правої кнопки миші, пункт *Add/Add Windows Form*.

Крім контекстного меню проекту існує ще ряд контекстних меню, що дозволяють управляти окремими елементами проекту. Так, щоб перемкнутися з вікна дизайнера у вікно коду проекту, виберіть в контекстному меню для *Form1* пункт *View Code*. Відповідно, щоб перемкнутися назад — *View Designer*. Контекстне меню є в кожного елемента дерева проекту. Використання контекстного меню — це швидкий інтерактивний спосіб навігації по проекту.

Приховування і відображення вікон – контекстне меню, команда *Auto Hide*.

### 4.2.2 Меню і панель інструментів

Головне меню має контекстну залежність від поточного стану середовища, тобто містить різні пункти залежно від того, чим ви зараз займаєтеся і в якому вікні знаходитесь. Основні команди меню відображаються і в панелі інструментів.

### 4.2.3 Properties Explorer – вікно властивостей об'єктів

Це вікно дозволяє працювати з властивостями форм і їх компонентів. Properties Explorer містить список всіх властивостей активного компоненту.

Другою важливою задачею, яку виконує *Properties Explorer*, є управління подіями. Для того щоб перемкнутися на закладку подій, натисніть кнопку із зображенням блискавки у верхній частині вікна.

Вікно подій дозволяє обробляти реакцію форми або компоненту на різні дії користувача або операційної системи, наприклад створити обробник подій від миші або клавіатури. В лівій частині вікна міститься список всіх доступних подій, а в правій — імен методів, які обробляють події. Типово, список методів порожній. Ви можете додати новий обробник, вписавши ім'я методу у відповідну комірку, або створити обробник з типовим іменем.

### 4.2.4 Вікно компонентів Toolbox

Вікно відображає найбільш часто використовувані .NET компоненти для створення застосунків Windows. Toolbox має декілька закладок: *Data*, *Components*, *Windows Forms*, *Clipboard Ring* і *General*. Всі, окрім *Clipboard Ring* і *General*, містять компоненти, які можна перетягнути мишею на форму. Закладка *Windows Forms* включає візуальні елементи управління, такі як кнопки, списки, дерева. Закладка *Data* присвячена базам даних. Закладка *Components* містить невізуальні компоненти, найбільш представницьким серед яких є *Timer*.

*Clipboard Ring* відображає вміст буфера обміну за останні N операцій копіювання або вирізування. Для вставки вмісту, який був скопійований в буфер обміну декілька операцій назад, просто клацніть двічі лівою кнопкою миші по необхідному рядку.

### 4.2.5 Властивості проекту

Кожний проект має набір властивостей. Середовище Visual Studio .NET дозволяє змінювати ці настройки візуально. Виділіть в дереві *Solution Explorer* кореневий елемент з назвою проекту. Натисніть пункт меню *View/Property Pages*. З'явиться вікно властивостей проекту «*Common Properties/General*» (таблиця 4.3).

Таблиця 4.3 – Основні властивості проекту

Властивість	Призначення
<i>Assembly Name</i>	Ім'я збірки
<i>Output Type</i>	Тут можна вибрати Windows Application
<i>Console Application</i> або <i>Class Library</i> .	Типово для Windows Forms встановлюється тип Windows Application
<i>Default Namespace</i>	Використовуваний типово в проекті простір імен.
<i>Startup Object</i>	Ім'я класу, що містить метод Main, який буде викликатися при запуску застосування.

<i>Application Icon</i>	Шлях до файла з піктограмою для застосування
<i>Project File</i>	Ім'я файла, що містить інформацію про проект.
<i>Project Folder</i>	Шлях до папки, що містить файл проекту.
<i>Output File</i>	Ім'я вихідного файла. Файл з таким ім'ям буде формуватися при побудові вашого застосування.

Крім того, вам доцільно знати про властивості на закладці *Configuration Properties/Build* (таблиця 4.4).

Таблиця 4.4 – Основні властивості конфігурування збірки (Рішення)

<b>Властивість</b>	<b>Призначення</b>
<i>Conditional Compilation Constants</i>	Визначені під час компіляції проекту константи. Вони допомагають розробнику управляти ходом компіляції проекту.
<i>Optimize code</i>	Включення цієї властивості в <b>true</b> допомагає, в деяких випадках, збільшити продуктивність програми.
<i>Check for Arithmetic Overflow/Underflow</i>	Дозволяє контролювати вихід результату за межі допустимих значень.
<i>Allow unsafe code blocks</i>	Дозволяє використовувати в проекті ключове слово <b>unsafe</b> .
<i>Warning Level</i>	Визначає рівень попереджень, що відображаються при компіляції програми.
<i>Treat Warnings As Errors</i>	Дозволяє сприймати всі попередження як помилки.
<i>Output Path</i>	Шлях, де буде сформований вихідний файл.
<i>XML Documentation File</i>	Ім'я файла, в який буде записуватися документація з коментарів програми. Для формування документації необхідно використовувати меню <i>Tools/Build Comment Web Pages</i> .
<i>Generate Debugging Information</i>	Генерувати інформацію відладки. Ця опція повинна бути включена при відладці застосування.

Краще залишити типові значення.

### **Дизайнер форм**

Дизайнер призначений для зручного і інтуїтивного створення інтерфейсу програми. До основних елементів дизайнера форм можна віднести:

- Properties Window (пункт меню *View /Properties Window*);
- Layout Toolbar (пункт меню *View/Toolbars/Layout*);
- Toolbox (пункт меню *View/Toolbox*).

## 5.1. Типи форм (модальні і не модальні)

### 5.1.1. Типи форм

У Windows є 2 типи форм: модальні (*Modal*) і не модальні. Модальність визначає поведінку фокусу вводу форми і тип інтерфейсу: однодокументний (*SDI*) чи багатодокументний (*MDI*).

Модальна форма не дозволяє переключати фокус вводу на іншу форму без свого закриття. Вона є реалізацією SDI. Прикладами модальних форм є всі стандартні блоки діалогу Windows (у тому числі повідомлення про помилки). Модальні форми мають фіксований розмір і містять обмежений склад команд системного меню.

Не модальна форма дозволяє переключатися на іншу форму. Вона реалізує MDI (*multiple document interface*). Ці форми можуть містити інші форми, які в цьому випадку називаються MDI child forms. MDI форма створюється після встановлення в **true** властивості `IsMdiContainer`.

Прикладом є вікно Word або Excel. Як правило, головна форма в застосунку є не модальною. Не модальні форми мають системне меню і кнопки згортки-відновлення. Вони можуть змінювати свій розмір.

Кожна форма як екземпляр класу `System.Windows.Forms` має властивості, методи та події.

### 5.1.2. Властивості форми

Властивості визначають розмір і поведінку форми, фон, заголовок форми, особливості керування формою.

Статичні властивості можна встановити двома способами:

1) написати відповідні конструктори з ініціалізацією в файлі `Form1.designer.cs`;


2) встановити значення властивостей у вікні властивостей (*Properties*) в середовищі Visual Studio.

Динамічні властивості можна задавати під час виконання як реакцію на події.

Основні властивості форми (класу `Form`) перелічені в таблиці 5.1.

Таблиця 5.1 – Основні властивості форми (класу `Form`)

Властивість	Призначення	Типове значення
Name	Назва форми в проєкті.	Form1, Form2...
AcceptButton	Встановлюється значення кнопки, яка буде спрацьовувати при натисненні клавіші Enter. Для того щоб ця властивість була активною, необхідна наявність принаймні однієї кнопки, розташованої на формі	None
BackColor	Колір форми	Control
BackgroundIma	Фоновий рисунок	None

ge		
CancelButton	Встановлюється значення кнопки, яка буде спрацьовувати при натисненні клавіші Esc.	None
ControlBox	Встановлюється наявність або відсутність трьох стандартних кнопок у верхньому правому кутку форми: "Згорнути", "Розгорнути" і "Закрити"	
Cursor	Визначається вид курсора при його положенні на формі	Default
DrawGrid	Встановлюється наявність або відсутність сітки з точок, яка допомагає форматувати елементи форми.	<b>True</b>
Font	Формат шрифту	"Microsoft Sans Serif"; "8,25pt"
FormBorderStyle	Визначення виду границь форми.	Sizable
Icon	Зображення іконки, що розташовується в заголовку форми. Підтримуються формати .ico.	 (Icon)
MaximizeBox	Визначається активність стандартної кнопки "Розгорнути" у верхньому правому кутку форми.	<b>True</b>
MaximumSize	Максимальний розмір ширини і висоти форми, що задається в пікселях. Форма буде приймати вказаний розмір при натисненні на стандартну кнопку "Розгорнути"	0; 0 (На весь екран)
MinimizeBox	Визначається активність стандартної кнопки "Згорнути" у верхньому правому кутку форми	<b>True</b>
MinimumSize	Мінімальний розмір ширини і висоти форми, що задається в пікселях. Форма буде приймати вказаний розмір при зміні її границь користувачем (якщо властивість FormBorderStyle має типові значення Sizable)	0; 0
Size	Ширина и висота форми	300; 300
StartPosition	Визначення розташування форми при запуску застосування.	WindowsDefaultLocation
Text	Заголовок форми.	Form1, Form 2
WindowState	Визначення положення форми при	Normal



	запуску. Можливі значення: Normal — форма запускається з розмірами, вказаними у властивості Size; Minimized — форма запускається згорнутою на панелі задач; Maximized — форма розгортається на весь екран.	
--	---	--

### 5.1.3. Методи форми

Методи забезпечують керування формою.

Таблиця 5.2 – Основні методи класу Form

Ім'я методу	Призначення
Form.ShowDialog()	Забезпечує представлення форми як модального <i>dialog box</i> .
Form.Show()	Показує форму як немодальний <i>dialog box</i> .
Form.SetDesktopLocation()	Використовується для позиціонування форми на поверхні <i>desktop</i> .
Form.Activate()	Активізує приховану форму.
Form.Hide()	Приховує форму.
Form.Close()	Закриває форму.

### 5.1.4. Події форми

Події форми пов'язані із створенням, знищенням, режимами роботи форми.

Таблиця 5.3 – Основні події класу Form

Подія	Призначення
Load	Генерується один раз, безпосередньо після першого виклику методу Form.Show() або Form.ShowDialog(). Ця подія може використовуватися для первинної ініціалізації змінних і для підготовки форми до роботи (для кожної форми).
OnLoad	Призначення максимальних і мінімальних розмірів форми
Activated	Подія генерується при активізації форми. В обробнику події вставляють методи Form.Show(), Form.ShowDialog(), Form.Activate()
VisibleChanged	Генерується при зміні властивості Visible форми коли вона стає видимою або невидимою. Події сприяють методи Form.Show(), Form.ShowDialog(), Form.Hide(), Form.Close().
Deactivated	Виникає при втраті фокусу формою в результаті

	взаємодії з інтерфейсом користувача або в результаті виклику методів <code>Form.Hide()</code> або <code>Form.Close()</code> – але тільки для активної форми. Якщо закривати неактивну форму, подія не відбудеться.
Closing	Виникає безпосередньо перед закриттям форми. У цей момент процес закриття форми може бути припинений і взагалі відмінений, чому сприяє розміщений в тілі обробника події наступний програмний код: <code>e.Cancel = true;</code> де <code>e</code> - подія типу <code>CancelEventArgs</code> .
Closed	Вже після закриття форми. В обробнику цієї події розміщується будь-який код для “очищення” після закриття форми.

## 5.2 Розміщення на формі елементів керування (Controls)

### 5.2.1 Ієрархія класів Control

Форма є контейнером, в якому розміщуються елементи керування (*Controls*) та компоненти.

Елементи керування — це компоненти, що забезпечують взаємодію між користувачем і програмою. Ці елементи розміщені в вікні *Toolbox*. Базовим класом є клас `Control`.

У вікні *Toolbox* елементи згруповані у групи за їх призначенням. Основні групи елементів:

#### Група командних об'єктів

Елементи `Button`, `LinkLabel`, `ToolBar` реагують на натискання кнопки миші і негайно запускають яку-небудь дію.

#### Група текстових об'єктів

Більшість застосунків надають можливість користувачу вводити текст і, у свою чергу, виводять різну інформацію у вигляді текстових записів. Елементи `TextBox`, `RichTextBox` приймають текст, а елементи `Label`, `StatusBar` виводять його. Для обробки введенного користувачем тексту, як правило, слід натискувати на один або декілька елементів з групи командних об'єктів.

#### Група перемикачів

До цієї групи входять об'єкти класів: `ComboBox`, `ListBox`, `ListView`, `TreeView`, `NumericUpDown` і інші.

#### Група контейнерів

Елементи-контейнери дозволяють групувати елементи. Як правило, елементи цієї групи розташовані на формі, служать підкладкою кнопкам, текстовим полям, спискам — тому вони і називаються контейнерами. Елементи `Panel`, `GroupBox`, `TabControl`, крім всього іншого, розділяють можливості застосунку на логічні групи, забезпечуючи зручність роботи.

### Група графічних елементів

Для розміщення і відображення на формі графічних елементів (піктограм, зображень, заставок) використовуються елементи `ImageList`, `PictureBox`.

`PictureBox` – об'єкт-контейнер для вставки зображень.

### Діалогові вікна

Виконуючи різні операції з документом — відкриття, збереження, друк, попередній перегляд, — ми стикаємося з відповідними діалоговими вікнами. Класи `OpenFileDialog`, `SaveFileDialog`, `ColorDialog`, `PrintDialog` містять вже готові операції для роботи з цими елементами.

### Група меню

Містить елементи для створення різних типів меню: звичайного панелі інструментів, контекстного меню.

На рисунку 5.1 зображено ієрархію класів `Control`.

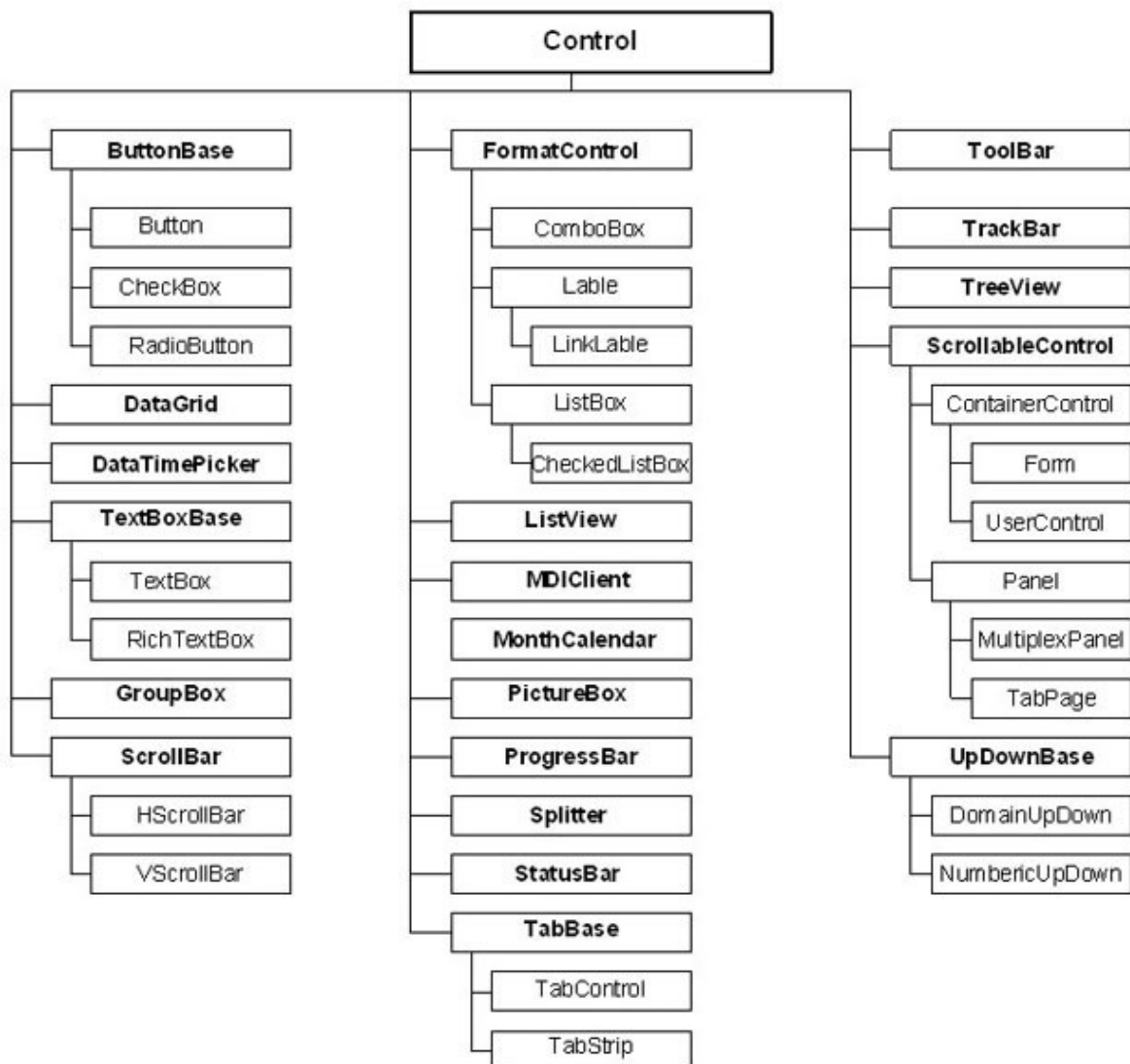


Рисунок 5.1 – Ієрархія класів `Control`

### 5.2.2 Заголовки (`Label`) і текстові поля (`TextBox`)

Клас `Label` (*мітка*) дозволяє виводити на форму текстову інформацію.

Клас `TextBox` походить безпосередньо від класу `TextBoxBase`, забезпечує загальними можливостями як `TextBox`, так і `RichTextBox`. Властивості, визначені в `TextBoxBase`. Основні властивості:

Таблиця 5.4 – Основні властивості класу `TextBoxBase`

Властивість	Призначення
<code>Name</code>	Назва поля ( <i>ідентифікатор</i> )
<code>Text</code>	Текст
<code>AutoSize</code>	Визначає, чи буде елемент управління автоматично змінювати розмір при зміні шрифту на ньому
<code>BackColor</code> , <code>ForeColor</code>	Дозволяють отримати або встановити значення кольору фону і переднього плану
<code>HideSelection</code>	Дозволяє отримати або встановити значення, визначальне, чи буде текст в <code>TextBox</code> залишатися виділеним після того, як цей елемент управління буде виведений з фокусу
<code>MaxLength</code>	Визначає максимальну кількість символів, яка можна буде ввести в <code>TextBox</code>
<code>Modified</code>	Дозволяє отримати або встановити значення, що визначає, чи був текст в <code>TextBox</code> змінений користувачем
<code>Multiline</code>	Указує, чи може <code>TextBox</code> містити декілька рядків тексту
<code>ReadOnly</code>	Позначає <code>TextBox</code> як доступний тільки для читання"
<code>SelectedText</code> , <code>SelectionLength</code>	Містять виділений текст (або певну кількість символів) в <code>TextBox</code>
<code>SelectionStart</code>	Дозволяє отримати початок виділеного тексту в <code>TextBox</code>
<code>WordWrap</code>	Визначає, чи буде текст в <code>TextBox</code> автоматично переноситися на новий рядок досягши граничної довжини рядка

В `TextBoxBase` визначено методи для роботи з буфером обміну (*Cut*, *Copy* і *Paste*), відміною введення (*Undo*) і іншими можливостями редагування (*Clear*, *AppendText* і т. п.).

З усіх подій, визначених в `TextBoxBase`, найбільший інтерес представляє подія `TextChanged`. Ця подія відбувається при зміні тексту в об'єкті класу, похідному від `TextBoxBase`. Обробник події можна використовувати для перевірки допустимості символів, що вводяться користувачем (наприклад, припустимо, що користувач повинен вводити в полі тільки цифри або, навпаки, тільки букви).

**Приклад 1.** Обробник події KeyPress для поля txtDisc, в яке МОЖНО вводити тільки літери:

```
private void txtDisc_KeyPress(object sender,
KeyPressEventArgs e)
{
    if (char.IsDigit(e.KeyChar))
    {
        e.Handled = true;
        MessageBox.Show("Поле не може містити
цифри");
    }
}
```

Призначення текстовому полю обробника події виконується у вікні *Property (Events)* для події KeyPress.

**Приклад 2.** Якщо поле може містити тільки цифри, то обробник може бути таким:

```
private void txtPIN_KeyPress(object sender,
KeyPressEventArgs e)
{
    if (!char.IsDigit(e.KeyChar) )
    {
        e.Handled = true;
        lbloutput.Text ="Поле PIN повинно
бути числовим";
    }
}
```

### Примітка

Подія KeyPress блокує частину клавіатури.

Крім властивостей, визначених в TextBoxBase, кожний похідний клас має власні властивості. В таблиці 5.5 перелічені основні властивості класу TextBox.

Таблиця 5.5 – Основні властивості класу TextBox

Властивість	Призначення
AcceptsReturn	Дозволяє визначити, що буде, коли користувач при введенні тексту натисне на <i>Enter</i> . Варіанту два: або в TextBox починається новий рядок тексту, або активізується типова кнопка на формі
CharacterCasing	Дозволяє вибрати символ, який використовується для відображення даних, що вводяться користувачем (в полі для введення пароля)
ScrollBars	Дозволяє отримати або встановити значення, яке

	визначає, чи будуть в TextBox з декількома рядками смуги прокрутки
TextAlign	Дозволяє визначити вирівнювання тексту в TextBox

### 5.2.3 NumericUpDown - регулятор чисел

Компонент `NumericUpDown` – варіант класу `ComboBox`, дозволяє без допомоги клавіатури вводити тільки числові значення в поле. Взагалі, цей елемент має три способи для введення даних: клацання мишкою на покажчики вгору-вниз, використання кнопок вгору-вниз на клавіатурі або введення даних в полі введення.

Вибране значення визначається властивістю `UpDown1.Value`. Повертає десяткове число, тому потрібно конвертувати його в тип `int`.

#### Приклад

```
int Value = (int)numericUpDown1.Value;
```

### 5.2.4 Контейнери елементів

Ціла група елементів-контейнерів в *C#* призначена для групування елементів. Основні контейнери:

`GroupBox` – логічне об'єднання елементів (прапорців, перемикачів).

`Panel` – Підтримує смуги прокрутки. Елементи `Panel` використовуються для економії простору на формі. Наприклад, якщо елементи, які плануємо розмістити на формі, на ній не вміщаються, то можна помістити їх всередину `Panel` і встановити для властивості `AutoScroll` об'єкту `Panel` значення `true`. В результаті користувач отримає можливість доступу до елементів управління, що не вміщаються, за допомогою смуг прокрутки.

### Контрольні питання і завдання

1. З яких файлів складається клас `Form`? Чому цей клас розділений на два файли?
2. В якому файлі знаходиться точка входу в збірку?
3. Чим модальна форма відрізняється від немодальної?
4. Що таке *неповний* клас?
5. Який метод і якого класу запускає програму на виконання? Як називається головний клас збірки?
6. У якому вікні VS розташовані елементи керування і стандартні компоненти?
7. За допомогою якого вікна можна налаштувати властивості форми і її елементів?
8. Як встановити на формі властивості для програмних кнопок, які будуть спрацьовувати при натисненні клавіш *Enter* і *Esc*? Які властивості при цьому слід встановити?
9. Яка різниця в елементах керування `Label` і `TextBox`?

10. Який елемент є базовим в ієрархії елементів керування?
11. Чим `NumericUpDown` відрізняється від `ComboBox`?

## ЛЕКЦІЯ 6. МАРШАЛІНГ

### План

6.1 Маршалинг. Етапи виклику методів.

6.2 СОМ-об'єкти.

6.3 Види маршалингу.

### 6.1 Маршалинг. Етапи виклику методів

**Маршалинг** (від англ. *to marshal* — *упорядковувати*) — процес перетворення представлення об'єкту в пам'яті у формат даних, придатний для зберігання або передачі. Зазвичай застосовується коли дані необхідно передавати між різними частинами однієї програми або від однієї програми до іншої.

Протилежний процес називається *демаршалингом*.

Місцезнаходження сервера прозоро для клієнта, але ми не торкалися питання про те, як ця прозорість забезпечується. Адже, по суті справи, клієнт викликає методи, які можуть розташовуватися в адресному просторі іншого процесу або навіть на віддаленому комп'ютері. Насправді методи, природно, викликаються в адресному просторі сервера. На клієнтській стороні збирається вся інформація, необхідна для виклику методу, потім ці дані передаються серверу, розшифровуються, викликається потрібний метод з потрібними параметрами, а результат передається клієнтові в зворотному порядку. Таким чином, завдання виклику методу через кордони процесу розбивається на три частини:

- Збір на стороні клієнта всіх необхідних для виклику даних і серіалізація їх в єдиний потік. Цей етап називається *маршалингом* (*marshaling*).
- Передача сформованого потоку через границі процесів або по мережі.
- Витяг на стороні сервера даних з потоку і формування на їх основі структур, які потім будуть передані методу в якості параметрів. Цей етап називається зворотним маршалингом (*unmarshaling*).

Нерідко під словом "*маршалинг*" об'єднують всі три етапи процесу, хоча це не зовсім правильно.

Детальне знайомство з етапами виклику ми почнемо з другого з них. Це єдиний етап, на якому не враховується специфіка викликається методу. Всі дії системи на даному етапі зводяться до вирішення простої транспортної задачі: як передати потік заданої довжини з адресного простору клієнта в адресний простір сервера. Це завдання вирішується спеціальними системними об'єктами, які називаються об'єктами каналу. Об'єкт каналу на стороні клієнта отримує потік даних і передає його об'єкту каналу на стороні сервера. Передача здійснюється за допомогою протоколу ORPC (*Object Remote Procedure Call* - виклик віддалених процедур об'єкта).



## 6.2 COM-об'єкти

Тепер повернемося до першого і третього етапів – прямого і зворотного маршалажу. Цим займаються два спеціальних об'єкта – заступник (*proxy*) і заглушка (*stub*). Заступник завантажується в адресний простір клієнта. Спрощено його можна розглядати як спеціальний COM-об'єкт, який реалізує ті ж інтерфейси, що і той об'єкт, який клієнт використовує. Фактично, клієнт працює з інтерфейсами заступника, а не самого об'єкта. Коли клієнт думає, що він викликає метод COM-об'єкта, насправді він викликає відповідний метод заступника. Заступник, отримавши виклик, формує потік, який слід відіслати серверу, і передає його своєму об'єкту каналу. До цього моменту дані передаються в межах одного адресного простору. Далі клієнтський об'єкт каналу, використовуючи ORPC, передає потік серверного об'єкту, а той – заглушці (і серверний об'єкт, і заглушка знаходяться в адресному просторі сервера). Заглушка витягує дані з потоку, відновлюючи їх структуру, і викликає відповідний метод сервера. Після завершення цього методу результати його роботи передаються клієнту по тому ж ланцюжку, але в зворотному порядку. Таким чином, для вхідних параметрів метод прямого маршалажу виконується заступником, а зворотний – заглушкою, а для вихідних параметрів – навпаки.

Використання заглушки і заступника призводить до того, що клієнт завжди викликає методи заступника в своєму адресному просторі, а сервер отримує виклики від заглушки також в своєму адресному просторі. Процес передачі даних з одного адресного простору в інше невидимий ні для сервера, ні для клієнта.

Упаковка даних в потік і їх подальше вилучення – не таке просте завдання, як це може здатися на перший погляд. Основна проблема полягає в передачі параметрів. З параметрами простих типів (**Integer**, **Real**, **Boolean** і т.п.) все просто - вони в потрібному порядку заносяться в потік і так само з нього витягуються. А ось з покажчиками так просто не вийде: безглуздо передавати в потік саме значення покажчика - в іншому адресному просторі воно не матиме ніякого сенсу. Потрібно вставити в потік ту область пам'яті, на яку посилається вказівник. Для цього треба знати розмір цієї області. Крім того, покажчик може посилатися на структуру, яка, в свою чергу, теж може містити покажчики, які також потрібно передавати не за значенням, а копіювати в потік потрібну область пам'яті. Деякі специфічні типи (як, наприклад, BSTR) можуть зберігати дані і в області, що знаходиться по негативному зміщенню від покажчика. Щоб коректно упаковати всі дані в потік, а потім витягнути їх і відновити їх структуру, і заглушка, і заступник повинні володіти вичерпною інформацією про всіх типах даних, використовуваних параметрами. Крім того, вони повинні знати, які параметри є вхідними і передаються від клієнта серверу, а які - вихідними (від сервера клієнту). Очевидно, що

побудова універсальних заступника і заглушки, рівною мірою придатних для всіх інтерфейсів, неможливо.

### 6.3 Види маршалингу

В COM / DCOM існує три види маршалингу, що розрізняються тим, хто несе відповідальність за упаковку і розпаковування даних:

*Стандартний маршалинг.* У цьому випадку заступник і заглушка створюються спеціальної динамічно компонованих бібліотекою, яка так і називається - *proxy / stub dll*. Ця бібліотека повинна бути встановлена і зареєстрована як на серверному, так і на клієнтському комп'ютері. Якщо сервер розрахований на стандартний маршалинг, його розробник повинен також створити *proxy / stub dll* і поширювати її разом зі своїм сервером. Деякі середовища розробки дозволяють автоматизувати побудову *proxy / stub dll* на основі формального опису інтерфейсів. Delphi в число таких засобів, на жаль, не входить. Тим не менш, на Delphi можна розробити сервер, орієнтований на стандартний маршалинг, хоча *proxy / stub dll* доведеться створювати іншими засобами. При стандартному маршалингу розробник має великий вибір типів для параметрів і може визначати свої структури.

*Універсальний маршалинг.* У цьому випадку заступник і заглушка реалізуються стандартної системної бібліотекою *oleaut32.dll*. Для того, щоб ця бібліотека могла врахувати особливості конкретних інтерфейсів, на клієнтському і серверному комп'ютерах повинна бути зареєстрована так звана бібліотека типів, яку надає розробник сервера. Бібліотека типів (*type library*) звичайно являє собою файл з розширенням *tlb*, який зберігає опис інтерфейсів. Бібліотека типів може бути також включена в сам сервер (як у внутрішній, так і в зовнішній), і тоді сервер стає самодостатнім, ніяких додаткових файлів для роботи з ним не потрібно. При використанні універсального маршалингу дозволено використовувати тільки VARIANT-сумісні типи.

*Користувальницький маршалинг.* У цьому випадку об'єкт сам займається упаковкою даних, реалізуючи стандартний інтерфейс *IMarshal*. Розробник сервера при цьому повинен надати оригінальну бібліотеку для створення заступника в адресному просторі клієнта, і цей заступник також повинен реалізовувати інтерфейс *IMarshal*. У цьому випадку розробник сервера отримує можливість не тільки керувати процесами прямого і зворотного маршалингу, але і вибрати свій спосіб передачі даних, відмовившись від послуг стандартних об'єктів каналу. Це дозволяє, наприклад, використовувати нестандартний транспортний протокол або навіть нестандартне фізичне з'єднання при взаємодії двох комп'ютерів. Ми не будемо більше тут зупинятися на користувальницькому маршалингові, так як він досить складний, а використовується рідко. Відзначимо тільки, що далі ми зіткнемося зі

стандартними інтерфейсами, які через особливості свого застосування розраховані на користувальницький маршalling.

Слід зазначити, що існує деяка неоднозначність в понятті "*стандартний маршalling*". Справа в тому, що більшість стандартних інтерфейсів маршалуються через *oleaut32.dll*, тому дана бібліотека зазвичай називається *стандартним маршалером*. Відповідно, маршalling за допомогою цієї бібліотеки також іноді називають стандартним, створюючи плутанину з маршallingа через оригінальну *proxy / stub dll*. Зокрема, в довідці Delphi маршallingа через бібліотеку типів і *oleaut32.dll* називається стандартним (мабуть, з маркетингових міркувань, тому що інакше довелося б писати, що Delphi не підтримує в повному обсязі стандартний маршallingа, а це звучить не дуже добре: стандартний – і раптом не підтримує). Ми тут під стандартним маршallingа завжди будемо мати на увазі маршallingа через *proxy / stub dll*.

При реєстрації інтерфейсу в реєстрі вказується спосіб його маршallingа і *proxy / stub dll* або бібліотека типів, які будуть при цьому використовуватися (виняток становить *користувальницький маршalling* - при його використанні інтерфейс взагалі не потребує реєстрації в реєстрі). Тип маршallingа є невід'ємною частиною інтерфейсу, тобто у всіх ситуаціях для маршallingа кожного конкретного інтерфейсу використовується один і той же метод і та сама бібліотека типів або *proxy / stub dll*.

Раніше ми говорили, що заступник можна спрощено розглядати як СОМ-об'єкт, що імпортує всі ті ж інтерфейси, що і віддалений об'єкт, який він заміщає. Цей об'єкт називається менеджером заступників (*proxy manager*). Він керує заступниками інтерфейсів. Окремий заступник - це СОМ-об'єкт, який реалізує два інтерфейси: *IRpcProxyBuffer* і той інтерфейс, для маршallingа якого використовується даний заступник (в деяких випадках один заступник інтерфейсу може відповідати за маршallingа відразу декількох інтерфейсів - в цьому випадку він реалізує їх все плюс *IRpcProxyBuffer*). Менеджер заступників самостійно реалізує інтерфейси *IUnknown* і *IMarshal* і, крім того, агрегує інтерфейси, що маршалуються заступниками інтерфейсів (агрегацію ми розглянемо трохи пізніше). *IRpcProxyBuffer* використовується тільки для взаємодії між менеджером заступників та заступниками інтерфейсів, клієнт не отримує доступ до цих інтерфейсів.

Таким чином, клієнт "бачить" в своєму адресному просторі СОМ-об'єкт, який реалізує всі інтерфейси необхідного йому СОМ-об'єкта (і, крім них, ще *IMarshal*, який використовується тільки системою, але не самим клієнтом). Цей СОМ-об'єкт сам по собі складається з компонентів, які в загальному випадку реалізуються різними бібліотеками. Дійсно, СОМ-об'єкт сервера може реалізовувати різні інтерфейси, які маршалуються різними способами або бібліотеками. Наприклад, інтерфейс *IUnknown* завжди маршується бібліотекою *oleaut32.dll* незалежно від того, як маршалуються інші інтерфейси об'єкта. Відповідно, може знадобитися

завантажити в адресний простір клієнта кілька різних бібліотек, кожна з яких створить свого заступника для маршала різних інтерфейсів одного об'єкта. Тим не менш, ця складна структура прихована від клієнта, йому заступник представляється єдиним об'єктом.

Хоча заступник вже зараз виглядає складним, надалі ми побачимо, що він ще більш складний, ніж ми його тут описали, тому виконує ряд додаткових функцій: буферизацію підрахунку посилянь, асинхронні виклики і т.п. На щастя, програмісту досить лише мати про все це загальне уявлення, деталі знати не обов'язково.

Об'єкт каналу – це теж СОМ-об'єкт, він реалізує інтерфейс `IRpcChannelBuffer`. Кожен заступник інтерфейсу отримує покажчик на `IRpcChannelBuffer`, через який він взаємодіє з об'єктом каналу. На вимогу заступника об'єкт каналу створює буфер, який заступник потім заповнює даними, і відправляє його об'єкту каналу на стороні сервера.

Чи використовують різні заступники інтерфейсу, об'єднані одним менеджером, один об'єкт каналу, або у кожного він свій – інформація на цей рахунок суперечлива. Пряма відповідь на це питання в MSDN відсутня, а непрямі вказівки в різних статтях дозволяють зробити як той, так і інший висновок. Однак знати це не обов'язково навіть в тому випадку, якщо стоїть завдання самостійно написати *proxy / stub dll* – система сама надає заступнику інтерфейсу покажчик на інтерфейс `IRpcChannelBuffer` об'єкта каналу і сама дбає про те, скільки об'єктів каналу створити.

Заглушка інтерфейсу також являє собою СОМ-об'єкт. Вона реалізує інтерфейс `IRpcStubBuffer`. Об'єкт каналу серверної сторони, отримавши виклик, використовує цей інтерфейс для передачі *заглушці* отриманих даних. Заглушки різних інтерфейсів працюють незалежно один від одного, тому ніякого менеджера заглушок, об'єднує їх, немає. Але в документації СОМ / DCOM рекомендується думати, що такий менеджер як би існує концептуально, тому що реально розрізнені заглушки імітують єдиний клієнт.

Схему передачі викликів через кордони адресного простору за допомогою заглушок і заступників ілюструє рисунок 6.1. У ньому передбачається, що сервер реалізує інтерфейси `IInterface1` і `IInterface2`, кожен з яких має власного заступника. Використане на малюнку позначення інтерфейсів у вигляді відрізків з колами на кінцях є загальноприйнятим і рекомендується до використання у всіх графічних зображеннях СОМ-об'єктів.

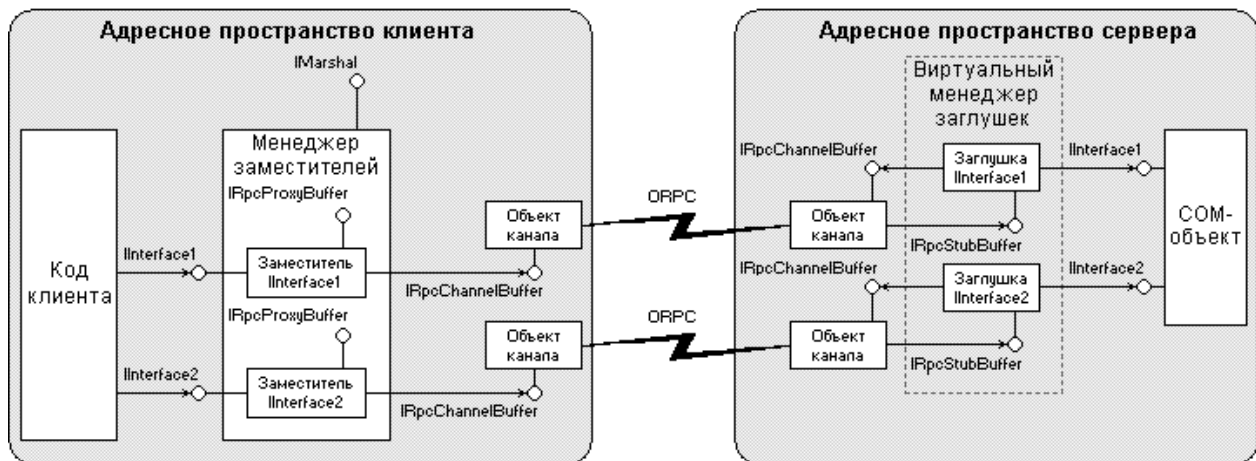


Рисунок 6.1 – Маршалинг

Коли клієнт намагається отримати покажчик на інтерфейс зовнішнього COM-об'єкта в перший раз, система запитує у COM-об'єкта сервера інтерфейс `IMarshal` (цей запит виконується звичайним чином, через `IUnknown.QueryInterface`). Якщо такий інтерфейс у COM-об'єкта є, система через його функції отримує інформацію про те, який COM-об'єкт буде виконувати маршалинг на стороні клієнта і завантажує відповідну *dll* в адресний простір клієнта. Якщо інтерфейс `IMarshal` у COM-об'єкта відсутня, або для обраного інтерфейсу COM-об'єкт не реалізує користувальницький маршалинг, система читає інформацію про запрошення інтерфейсу з реєстру. Якщо там вказано, що слід використовувати стандартний маршалинг, відповідна *proxy / stub dll* завантажується в адресний простір клієнта і сервера. Якщо використовується універсальний маршалинг, з реєстру витягується інформація про місцезнаходження бібліотеки типів, і в адресний простір клієнта і сервера завантажується *oleaut32.dll*, яка будує заглушку і заступник на підставі даних з цієї бібліотеки. Для клієнта цей процес абсолютно прозорий: система виконує всі ці дії без його участі. Сервер, за винятком випадків користувальницького маршалинга, так само сам по собі не бере участь у процесі створення заступника і заглушки, але він повинен супроводжуватися або бібліотекою типів, або *proxy / stub dll*, щоб система могла виконати вищеперелічені дії.

Слід зауважити, що маршалинг іноді потрібний і при використанні внутрішніх COM-серверів, коли клієнт і сервер знаходяться в одному адресному просторі. Далі ми будемо розбирати ці випадки детально, а поки відзначимо, що COM-сервер, реалізований у вигляді *dll*, також повинен супроводжуватися бібліотекою типів, *proxy / stub dll* або реалізовувати `IMarshal`, інакше область його застосування буде сильно обмежена.

Розглянемо переваги і недоліки *стандартного* і *універсального* маршалинга. Головне достоїнство *стандартного маршалинга* - велика свобода у виборі типів параметрів. Програміст може створювати свої структури і передавати масиви будь-якої розмірності по посиланню.

Структури можуть містити покажчики на дані простих типів, на інші структури і на масиви, масиви можуть складатися з структур. При використанні універсального маршалаingu доступні тільки громіздкі безпечні масиви, а структури доступні тільки після спеціальних хитрощів, про які ми будемо говорити пізніше.

Коли розробник сервера реалізує інтерфейси, розроблені кимось іншим, він не може вибирати тип маршалаingu - він уже обраний розробником інтерфейсів. Надати бібліотеку типів або *proxy / stub dll* в цьому випадку повинен також розробник інтерфейсів. Проблема вибору способу маршалаingu виникає перед розробником сервера тільки тоді, коли він сам розробляє інтерфейси. При програмуванні на Delphi більш природним є вибір універсального маршалаingu, тому що бібліотека типів може бути побудована середовищем. Що ж стосується *proxy / stub dll*, то її доведеться писати повністю вручну або використовувати засоби, що не входять в поставку Delphi.

Ми вже відзначали, що бібліотека типів може бути інтегрована в сам сервер, і тоді з ним не потрібно поширювати ніяких додаткових файлів - все знаходиться всередині нього. Це може бути зручно, коли сервер реалізує унікальні інтерфейси, які не зустрічаються більше в одному сервері. Якщо ж набір інтерфейсів такий, що його реалізують різні сервери (така ситуація зустрічається, наприклад, коли у вигляді СОМ-серверів реалізуються *plug-in*'и: кожен *plug-in*, щоб бути сумісним з основним застосунком, повинен підтримувати фіксований набір інтерфейсів), то бібліотеку типів краще робити незалежною. Інакше вийде, що кожен сервер буде реєструвати сам себе в якості бібліотеки типів, а так як для кожного інтерфейсу в системі може бути зареєстрована лише одна бібліотека типів, всі сервери виявляться залежними від того, який був встановлений останнім, і перестануть працювати, якщо він буде деінсталюваний.

### **Контролі питання:**

1. Що таке маршалаingu?
2. Які є види маршалаingu? Розкрийте їх зміст.
3. Що вказується при реєстрації інтерфейсу в реєстрі?
4. Що таке заглушка інтерфейсу і що вона робить?
5. Яке головне достоїнство стандартного інтерфейсу?

## ЛЕКЦІЯ 7. COM, DCOM, .NET

### План

- 7.1 Основні поняття COM, DCOM, .NET.
- 7.2 Об'єктна модель розподілених компонентів.
- 7.3 Трирівнева архітектура бізнес-застосунків.

### 7.1 Основні поняття COM, DCOM, .NET

COM (*Component Object Model*) розшифровується просто – компонентна об'єктна модель. DCOM (*Distributed Component Object Model*) – це, відповідно, розподілена компонентна об'єктна модель.

Microsoft розробляє і підтримує DCOM виключно для створення серверів застосувань і управління розподіленими програмними об'єктами. На противагу Microsoft, два інших комп'ютерних гіганта – Sun і IBM – спираються на міжнародний стандарт CORBA. За бажання можна спільно використати обидві ці технології.

### 7.2 Об'єктна модель розподілених компонентів

Об'єктна модель розподілених компонентів ([англ. Distributed Component Object Model](#), скорочено DCOM) — власна технологія Microsoft для організації взаємодії між [компонентами програмного забезпечення](#), розподіленого між комп'ютерами в [мережі](#). DCOM, що спочатку мала назву «*Network OLE*», розширює інтерфейс [Microsoft COM](#) і забезпечує нижні рівні зв'язку з інфраструктурою сервера Microsoft COM+. Наразі, ця технологія застаріла та замінена [Microsoft .NET](#).

Доповнення «D» до [COM](#) відбулося завдяки використанню DCE/RPC — більш розширеної версії Microsoft, відомої як MSRPC.

.NET — прикладний програмний інтерфейс (API) від Microsoft для взаємодії між процесами, який був випущений у 2002 разом 1.0 версією *.NET Framework*. Це одна із серії технологій Microsoft, серія розпочалася у 1990 році випуском першої версії *Object Linking and Embedding* (OLE) для 16-бітної Windows. Проміжними кроками у розвитку цих технологій були *Component Object Model* (COM), яка вийшла в 1993 та оновлена в 1995 як COM-95, *Distributed Component Object Model* (DCOM), яка вийшла в 1997 (перейменована у ActiveX), і COM+ з її *Microsoft Transaction Server* (MTS), яка вийшла в 2000. Зараз місце цих технологій займає *Windows Communication Foundation* (WCF), яка є частиною .NET Framework 3.0. Також .NET Framework 3.0 входить у склад операційної системи Windows Vista.

Так само як члени родини технологій від Microsoft і схожі технології, такі як *Common Object Request Broker Architecture* (CORBA) та *remote method invocation* (RMI) у Java, .NET — це комплекс, у прямому сенсі цього слова. За допомогою операційної системи та мережевих агентів

клієнтський процес відсилає повідомлення серверному процесу та отримує відповіді

Один із ключових факторів у вирішенні цих проблем— використання DCE/RPC як основного механізму RPC поза DCOM. DCE/RPC чітко визначає правила щодо маршалінгу і відповідальності за вивільнення пам'яті. Крім того, технологія DCOM, підтримку якої має будь-яка ОС сімейства Windows, поєднує переваги технологій доступу до даних з клієнт-серверною технологією.

DCOM просто розширює можливості COM в частині реєстрації віддалених об'єктів, тому його іноді називають "COM з *подовженим дротом*".

OLE-механізми засновані на компонентній об'єктній моделі (COM, *Component Object Model*), яка дозволяє створювати об'єктно-орієнтовані застосунки на одній машині у рамках операційної системи Windows, що функціонує на ній, а ActiveX використовує DCOM (*Distributed Component Object Model*) – розподілену компонентну об'єктну модель, яку реалізують бібліотеки ActiveX, що за об'ємом є значно меншими, ніж бібліотеки OLE, проте швидші. Іншими словами, бібліотеки OLE переписані так, щоб забезпечувати функціональність, достатню для написання мережових застосунків. Збереглася і сумісність – будь-який програмний компонент OLE працюватиме з бібліотеками ActiveX. Моделі COM і DCOM спираються на базові мережові протоколи, такі як TCP/IP, і входять до складу операційної системи.

*Прикладні інтерфейси COM API* – це усе ті ж, усім відомі OLE-інтерфейси, що ведуть своє походження від DDE (*Dynamic Data Exchange*), – динамічного обміну даними, що дозволяє єдиним способом в Windows здійснювати обмін інформацією між процесами.

По суті своїй COM є простим об'єктним розширенням OLE-технології, коли в локальній моделі засобів автоматизації OLE команди від клієнта проходять через модуль-ініціалізатор (*proxy*), а повідомлення – в модуль-перехідник (*stub*), де воно розбирається, аналізується і звідки посилається команда на сервер, як показано на рисунку 7.1.

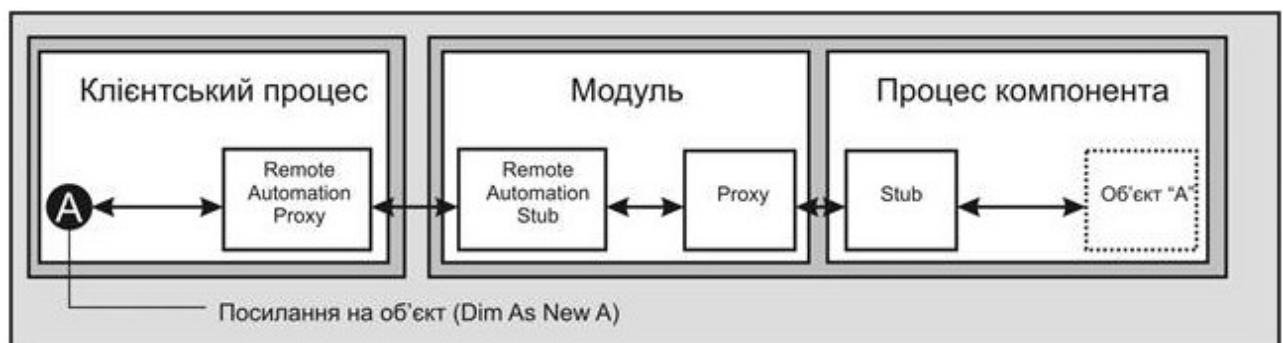


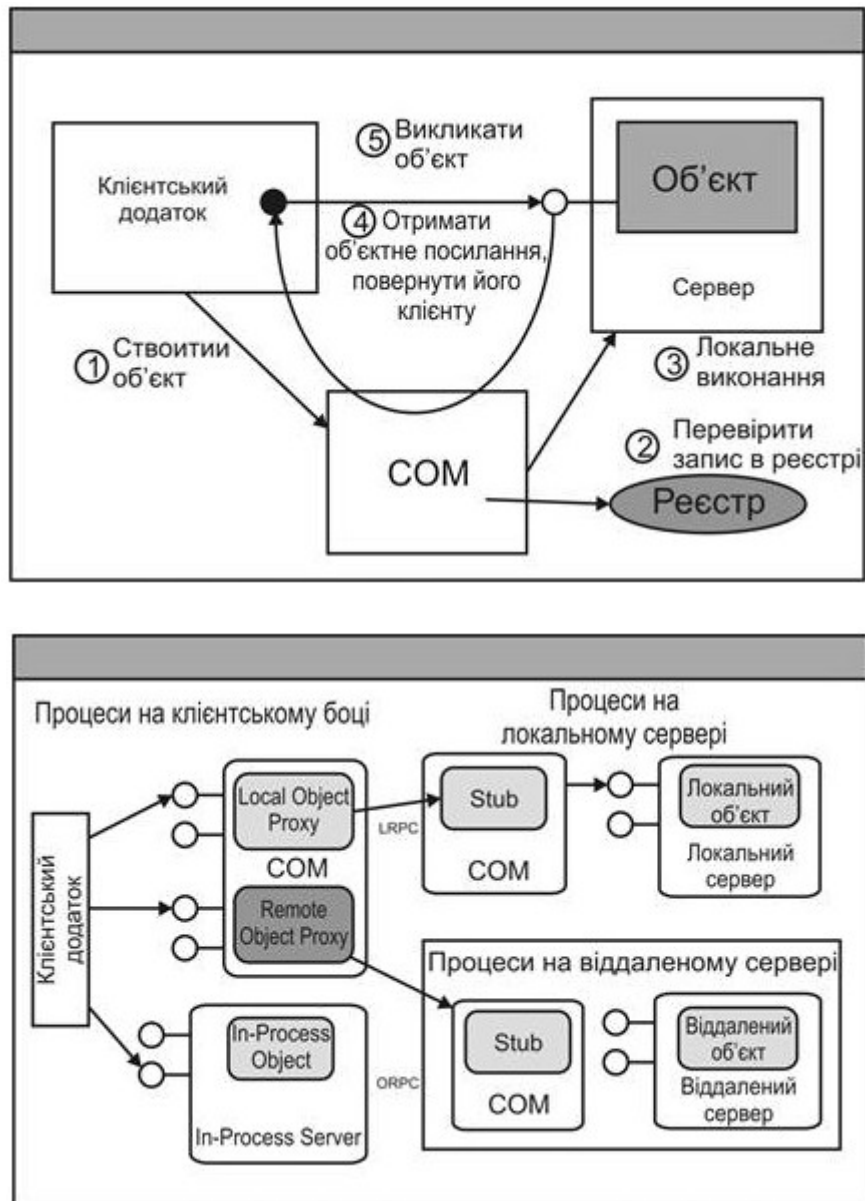
Рисунок 7.1 – Схема роботи OLE Automation

Наочніше ця ж схема, але вже в специфікації COM, представлена на рисунку 7.2.



Модель COM значно простіша з точки зору програмістів і її набагато зручніше використати в невеликих локальних мережах.

У дистанційній моделі, яка тепер замість COM дістала назву DCOM, ініціалізатор і перехідник стали абсолютно іншими, тепер в них використовуються процедури дистанційного виклику (RPC) для передачі запитів клієнта по мережі до модуля Remote Server Process, що функціонує на сервері. Клієнтська частина DCOM, аналогічна колишньому механізму Automation Manager, передає дані клієнта вказаному OLE-серверу і дані у відповідь по мережі до програми-клієнта, як показано на рисунку 7.3.



Рисунки 7.2, 7.3 – Схема взаємодії для локального і віддаленого варіантів

Як можна бачити з приведених рисунків, чудовою властивістю DCOM є його прозорість як для користувача, так і для програміста. Можна створити і запустити на своїй машині сервер OLE, а потім пристосувати його для роботи в дистанційному режимі простим перенесенням OLE-сервера на віддалений ПК, на якому працює модуль адміністратор

автоматизованого управління DCOM. Потім треба просто зареєструвати нове місце розташування OLE-сервера за допомогою звичайної утиліти (наприклад, з арсеналу засобів Visual Basic). Програмістові не доводиться міняти жодного рядка тексту програми, щоб використати розроблений OLE-сервер віддалено. Зараз майже у будь-якій RAD-системі є шаблони для створення OLE-серверів, методи і об'єкти яких доступні при роботі в дистанційному режимі. І, оскільки реєстрацію віддаленого сервера легко включити в процедуру установки програми-клієнта, не доводиться робити яких-небудь додаткових дій, щоб скористатися новими можливостями.

Робота серверів OLE в дистанційному режимі має ряд істотних переваг, у тому числі спрощене адміністрування і обслуговування, можливість його багатократного використання будь-якою *програмою-клієнтом* OLE, багаторівнева система захисту доступу і достовірності даних. Але найголовніше перевага – це здатність віддалених OLE-серверів виконувати роль серверів застосувань, тобто своєрідних виконавців алгоритмів (у новій термінології – *бізнес-правил*) для доступу до даних в розподілених прикладних системах.

Реалізація цієї можливості є ключовим моментом в створенні трирівневої архітектури бізнес-застосунків Microsoft, в якій частина призначеного для користувача інтерфейсу розміщена на робочій станції клієнта, бізнес-логіка – на сервері середнього рівня, а надпотужні засоби обробки даних – на SQL-сервері бази даних (рисунок 7.4).

### 7.3 Трирівнева архітектура бізнес-застосунків

Наприклад, трирівнева модель абсолютно не потрібна (і не обов'язково намагатися ділити програму на частини), якщо вона не вирішує загальну проблему ефективності.



Рисунок 7.4 – Трирівнева архітектура бізнес-застосувань

Проте у фірмовій документації Microsoft, присвяченій OLE-об'єктам, стверджується: "Найчастіше розміщення сервер-об'єктів доступу до даних на тій машині, де знаходиться база даних, може істотно скоротити завантаження мережі і збільшити загальну швидкість передачі даних". Є приклади реалізації трирівневих клієнт-серверних застосувань, де результати перевершують усі очікування. Продуктивність мережі практично перестає впливати на час обробки транзакцій, і своєрідне розпилювання монолітних програмних блоків на мобільні OLE-сегменти приводить до значного підвищення ефективності роботи застосування в цілому. Більше того, трафік в мережі істотно зменшується і стає більше передбачуваним.

Найприйнятніший, і найбільш простий, ефективний підхід на думку більшості творців територіально-розподілених застосувань – це створення трирівневої архітектури. У цій моделі є користувач-клієнт і сервер бази даних, а також середній рівень, який діє як "діловий" або *інтелектуальний* сервер застосувань. Можна потім спростити програму користувача, переміщаючи ділову логіку системи на сервер застосувань, який міг би знаходитися на тому ж самому комп'ютері, що і сервер бази даних (чи на іншому комп'ютері). Часто це дає ефект прискорення розробки і пониження дистрибутивних витрат для обслуговування "товстих" клієнтів. Сервер застосувань здійснює запити до сервера бази даних за вимогами клієнтів і реалізує бізнес-логіку, що не лише спрощує розробку і супровід клієнтів, але також означає, що слід модифікувати тільки одну програму при зміні ділової логіки реалізації бізнес-процесів.

Щоб краще зрозуміти можливості розподіленої архітектури, розглянемо приклад з реального життя. Припустимо, є велика транспортна компанія M-Trans, яка забезпечує своїх клієнтів спеціальною програмою для відстежування шляху проходження вантажу. Коли ви відправляєте вантаж з Москви в Делі, вам дається спеціальний транспортний номер. Потім, якщо ви захочете дізнатися, що сталося з вашим вантажем на шляху слідування, ви просто завантажуєте програму (назвемо її MTrack) і вводите свій транспортний номер. Діалоговий модуль зв'язується з центральною універсальною ЕОМ компанії і відшукує інформацію відносно поточного місця розташування і стану вашого вантажу. Наприклад, ви могли б дізнатися, що вантаж знаходиться нині у вузловому аеропорту відвантаження або що він був вже отриманий замовником. Електронний підпис замовника, природно, повинен свідчити про цей відрядний факт.

### **Контрольні питання**

1. Дайте визначення поняттям COM, DCOM, NET.
2. Охарактеризуйте об'єктну модель розподілених компонентів.
3. Що таке OLE-інтерфейси?
4. Переваги і недоліки OLE-інтерфейсу.
5. Охарактеризуйте відмінності COM і DCOM.
6. Опишіть трирівневу модель.

7. Наведіть приклад можливості розподіленої архітектури.

## ЛЕКЦІЯ 8. CORBA

### План

8.1 Поняття CORBA.

8.2 Архітектура CORBA.

8.3 Розподілені дані, застоснки, обчислення, користувачі.

### 8.1 Поняття CORBA

CORBA (*Common Object Request Broker Architecture*) – це технологія для світу розподілених інформаційних об'єктів, що тісно взаємодіють між собою у рамках програми, що управляє ними, яка віртуально з цих об'єктів і складається. Так от, CORBA – це архітектура, яка з максимальними зручностями забезпечує створення і функціонування програм, званих CORBA-застосунками. Останнім часом численні RAD-системи, наприклад Delphi, включають у свій арсенал підтримку CORBA.

Втім, традиційні RAD-системи, подібні Visual Basic, не потребують CORBA, тому що орієнтуються на наявний в Microsoft власний брокер об'єктних запитів, існуючий під іменем DCOM. Надалі виробникам програмних продуктів для програмістів стане все важче і важче підтримувати обидві технології. Delphi-програмісти можуть заспокоювати себе тим, що є можливість працювати і з CORBA, і з DCOM.

Звичайно ж, CORBA – більше масштабований, відкритіший і грандіозніший проект, ніж DCOM. CORBA розглядає вже існуючі застосування, як деякі метафори, які легко можна адаптувати і використати, правильно і одного разу описавши їх інтерфейси за допомогою спеціальної мови описів IDL (*Interface Definition Language* – мова опису запитів), для якого існують компілятори у більшість сучасних мов програмування. CORBA є прообразом нашого об'єктного майбутнього. У комп'ютерному світі абсолютно нормальним вважається факт, що програмісти усіх країн по мільйону разів переписують наново один і той же алгоритм, якщо він вимагає внесення деяких невеликих змін.

Що означають об'єктні технології в програмуванні, і не лише в програмуванні, а ширше, – в усій комп'ютерній творчості в епоху безроздільного панування Мережі? Ми вимушені визнати, що сучасне виробництво начисто позбавлене індивідуалізму і визнає тільки те, що може бути розмножене. Найкращим варіантом вважається той, який ви не створюєте, а лише настроюєте, компонуючи його з різного набору об'єктів, описуючи їх властивості і задаючи різні методи, що виконують той або інший вид роботи.

Сила CORBA в тому, що це система, що самоописується і самодокументується. Вона була розроблена для того, щоб дозволити створювати інтелектуальні компоненти, які можуть запросити інформацію один про одного і потім її ефективно використати. Число взаємодіючих об'єктів при цьому не обмежене, способи існування і місцезнаходження

об'єктів жорстко не визначені. Будь-який об'єкт може бути затребуваний іншим об'єктом або програмою в довільній точці Мережі, і це дуже сильно нагадує телепортацію.

## 8.2 Архітектура CORBA

Чотири головні елементи CORBA архітектури показані на рисунку 8.1.

**Брокер об'єктних запитів** (ORB – *Object Request Broker*) визначає механізми взаємодії об'єктів в різномірній мережі. Посередник об'єктних запитів, **ORB**, або просто брокер, – це логічне ядро системи. Саме він дозволяє об'єктам посилати запити і отримувати відповіді від інших об'єктів в мережі. ORB – це проміжне програмне забезпечення, яке встановлює відношення між об'єктами в розподіленому середовищі. Брокер по посиланню знаходить на сервері, що містить об'єкт-адресат (*object implementation* – реалізація об'єкту); активізує його, доставляє до нього запит; передає параметри і викликає відповідний метод, після чого повертає результат клієнтові. Ролі клієнта і сервера при цьому можуть мінятися. Взаємодія може бути встановлена між безліччю об'єктів. OMG розробив спеціальну мову верхнього рівня для опису ORB і інших компонентів CORBA – так званий IDL (*Interface Definition Language* – мова опису інтерфейсів). У ній немає присвоєнь, звичайних мовних конструкцій, типу **if** чи **while**, функцій і логічних переходів і т. д. IDL – це мова декларацій, вона описує батьківські класи, події і методи. Кожен інтерфейс визначає операції, які можуть бути викликані клієнтами, але не те – яким чином ці операції виконуються зовні IDL і CORBA. У цьому і привабливість – можна змусити працювати старі програми, якщо правильно описати їх інтерфейси і використати компілятор з IDL, який має відображення в конкретну мову програмування. Важливо зрозуміти, що є два типи інтерфейсів: *динамічні* і *статичні*. Перші визначаються на стадії розробки застосування і компілюються разом з ним, другі конструюються у момент виконання, на працюючому застосунку.

- **Сервіси об'єктів** (*Object Services*) є основними системними сервісами, які розробники використовують для створення застосувань.

- **Універсальні засоби** (*Common Facilities*) є також системними сервісами, але більш високого рівня, орієнтованими на підтримку призначених для користувача застосувань, таких як електронна пошта, засоби друку і т. д.

- **Прикладні об'єкти** (*Application Objects*) використовуються в конкретних прикладних завданнях.

Так, все-таки, навіщо потрібна CORBA?



Рисунок 8.1 – Головні елементи CORBA

### 8.3 Розподілені дані, застосунки, обчислення, користувачі

#### *Розподілені застосунки*

CORBA-архітектура забезпечує каркас для розробки і виконання розподілених застосунків. Але виникає нове питання – а навіщо взагалі потрібний цей розподіл? Як тільки ви зіткнетеся з ним, то побачите перед собою величезну купу нових проблем. Проте іноді немає іншого вибору: деякі застосунки просто вимагають бути розподіленими з наступних причин:

- дані, використовувані застосунком, – розподілені;
- обчислення – розподілені;
- користувачі застосунку – розподілені.

#### *Розподілені дані*

Деякі застосунки повинні виконуватися на безлічі комп'ютерів, тому що дані, до яких вони повинні звернутися, існують також на безлічі комп'ютерів. Власник може дозволити дистанційний доступ до даних, але зберігаються вони як локальні. Такі історичні (чи інші) причини, що визначають розподілену організацію даних.

#### *Розподілені обчислення*

Деякі застосування виконуються на безлічі комп'ютерів, щоб скористатися перевагою паралельного обчислення для вирішення специфічних завдань, наприклад, пов'язаних з дешифруванням. Інші застосування також можуть виконуватися на безлічі комп'ютерів, але щоб використати переваги деяких специфічних систем для реалізації різних частин свого алгоритму. Розподілені застосування завжди виграють при використанні необмеженої масштабованості і неоднорідності розподіленої системи.

#### *Розподілені користувачі*

Деякі застосування розподілені по безлічі комп'ютерів, тому що користувачі зв'язуються і взаємодіють один з одним через це застосування. Кожен користувач виконує фрагмент розподіленого застосування на його власному або чужому комп'ютері і активно використовує загальнодоступні об'єкти, які зазвичай виконуються на одному або більшій кількості

серверів. Типова архітектура для цього виду застосувань ілюструється на рисунку 8.2.

Як вже говорилося, CORBA є ідеальною архітектурою для створення таких застосувань. Актуальність її в наші дні активно затребувана розвитком Інтернету і систем електронної комерції, де багато функцій має бути розподілені по мережі і де є удосталь безліч вже працюючих об'єктних модулів, існуючих у вигляді ActiveX, або JavaBean-компонентів.

Брокер об'єктних запитів, ORB, ставить запит об'єкту і повертає будь-який результат клієнтові (рисунок 8.3).

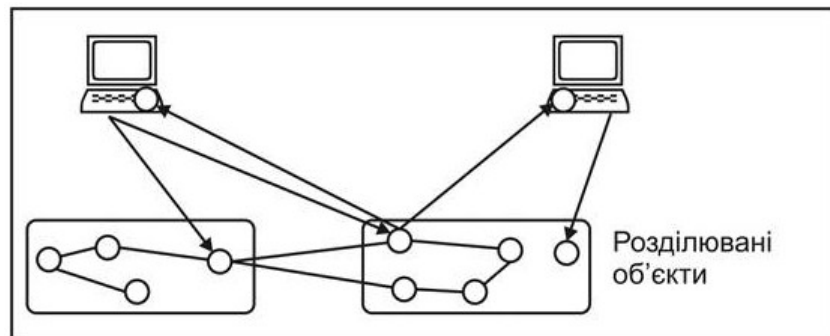


Рисунок 8.2 – Розподілені користувачі

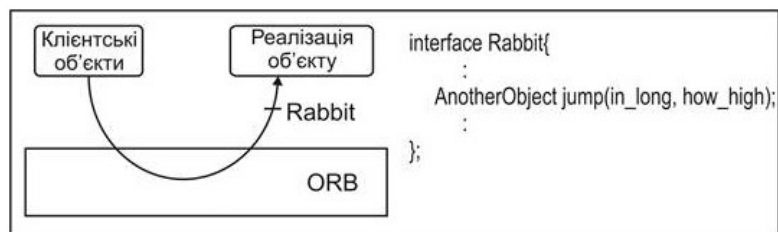


Рисунок 8.3 – Механізм взаємодії: клієнт запитує об'єкт через посередника ORB

### ***Від клієнта до сервера***

Оскільки CORBA є стандартом для створення розподілених застосувань, коли комп'ютери через віддалений доступ викликають програмні методи і об'єкти один одного, в ній чітко прописаний рівень міжмережових взаємодій поза всякою залежністю від місцезнаходження, мови і архітектури.

Можна чітко сформулювати ключові особливості CORBA-стандарту:

- CORBA-об'єкти можуть знаходитись в будь-якому місці мережі;
- CORBA-об'єкти можуть взаємодіяти з іншими CORBA-об'єктами на будь-яких платформах;
- CORBA-об'єкти можуть бути написані на будь-якій мові програмування, для якого є інтерфейс, що реалізовується IDL-компілятором (такі є для Java, C++, C, SmallTalk, COBOL і Ada).



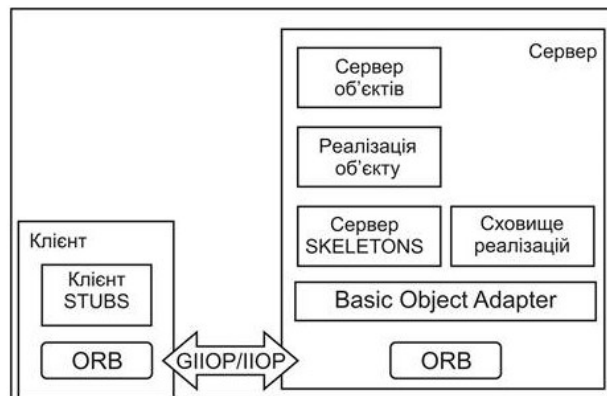


Рисунок 8.4 – Архітектура CORBA-сумісного застосування, що викликає безліч об'єктів з безлічі комп'ютерів

Є безліч CORBA-сервісів. Найпопулярніші з них наведені в таблиці 8.1.

Таблиця 8.1 – Найбільш популярні сервіси CORBA

Сервіс	Опис
Життєвий цикл об'єкта ( <i>Object life cycle</i> )	Визначає, яким чином CORBA-об'єкти будуть створені, видалені, переміщені або скопійовані
Іменування ( <i>Naming</i> )	Визначає спосіб символічного позначення CORBA-об'єктів
Події ( <i>Events</i> )	Обробка подій
Відношення ( <i>Relationships</i> )	Визначають стосунки між об'єктами (головний, підлеглий, зв'язковий і т. д.)
Перетворення об'єктів з однієї форми в іншу ( <i>Externalization</i> )	Координує перетворення об'єктів із зовнішніх форм у внутрішні і навпаки
Транзакції ( <i>Transactions</i> )	Координують доступ до CORBA-об'єктів
Контроль доступу ( <i>Concurrency Control</i> )	Забезпечує обслуговування блокування об'єктів CORBA
Властивість ( <i>Property</i> )	Підтримує асоціацію пари : ім'я-значення для об'єктів CORBA
Продавець ( <i>Trader</i> )	Підтримує пошук CORBA-об'єктів, заснований на афішованих властивостях об'єкту
Запит ( <i>Query</i> )	Підтримка запитів до об'єктів

### ***CORBA-продукти***

Не забувайте, що CORBA – це усього лише специфікація. Що стосується програмних виробів, що реалізують архітектуру CORBA, то їх є множина. Окремі виробники забезпечують свої брокери CORBA і IDL для відображення в різні мови програмування. Усі вони зобов'язані підтримувати Java.

Таблиця 8.2 – Найбільш відомі реалізації CORBA

ORB	Опис
Java ORB	Java 2 ORB поставляється у складі Sun's Java SDK. Там відсутні деякі особливості повної специфікації
VisiBroker for Java	Популярний Java ORB з Inprise Corporation. VisiBroker вбудований у багато інших продуктів. Приміром, саме він був вбудований у браузер Netscape Communicator.
OrbixWeb	Добротний Java ORB з Iona Technologies (для Unix)
WebSphere	Потужний сервер застосувань зі вбудованим ORB від IBM
Free or shareware ORBs	CORBA-реалізації для різних мов програмування доступні для завантаження по мережі Інтернет з різних джерел

**Контрольні питання**

1. Дайте визначення поняттю CORBA.
2. Назвіть головні елементи CORBA.
3. Універсальні засоби.
4. Сервіси об'єктів.
5. Прикладні об'єкти.
6. Що таке брокер об'єктних запитів?
7. Що таке розподілені дані застосунків?
8. Перелічіть найбільш відомі реалізації CORBA.

## ЛЕКЦІЯ 9. JAVA BEANS

### План

- 9.1 Поняття JavaBeans.
- 9.2 Правила опису JavaBean.
- 9.3 Основні поняття моделі JavaBeans.
- 9.4 Життєвий цикл компонентів JavaBeans.

### 9.1 Поняття JavaBeans

**JavaBeans** — класи написані мовою [Java](#), що відповідають набору правил. Вони використовуються для об'єднання кількох об'єктів в один (*bean*) для зручної передачі даних.

Специфікація [Sun Microsystems](#) визначає JavaBeans, як «універсальні програмні компоненти, якими можна керувати за допомогою графічного інтерфейсу» (*«reusable software components that can be manipulated visually in a builder tool»*).

JavaBeans забезпечують основу для багаторазово використовуваних і модульних [компонентів ПЗ](#). Компоненти JavaBeans можуть приймати різні форми, але найбільш широко вони використовуються в елементах графічного [інтерфейсу](#) користувача. Одна з цілей створення JavaBeans — взаємодія із схожими компонентними [структурами](#). Наприклад, [Windows-програма](#), за наявності відповідного *мосту* або *об'єкта-обгортки*, може використовувати компонент JavaBeans так, ніби він є компонентом COM або ActiveX.

### 9.2 Правила опису JavaBean

Щоб [клас](#) міг працювати, як *bean*, він повинен відповідати певним визначеним угодам про імена методів, конструктор і поведінку. Ці угоди дають можливість створення інструментів, які можуть використовувати, замінювати і з'єднувати JavaBeans.

#### **Правила опису:**

- Клас повинен мати публічний [конструктор](#) без параметрів. Такий конструктор дозволяє інструментам створювати об'єкт без додаткових складностей з параметрами.
- Властивості класу повинні бути доступні через методи `get`, `set`, `is`, які відповідають стандартним згодам про імена. Це дозволить інструментам автоматично визначати і оновлювати вміст *bean*'ів.
- Клас повинен бути [серіалізовним](#). Це дає можливість надійно зберігати та відновлювати стан *bean* незалежним від платформи і віртуальної машини способом.
- Він не повинен містити ніяких методів обробки подій.

### 9.3 Основні поняття моделі JavaBeans

Середовище JavaBeans є надбудовою над стандартною Java-технологією. Вона успадковує поняття і характеристики Java, такі як об'єктна орієнтованість, використання віртуальної машини, незалежність від апаратно-програмної платформи, інформаційна безпека і т.п. У JavaBeans немає нічого, щоб відрізнялося в термінах мови Java.

Основою середовища JavaBeans є компонентна об'єктна модель, що представляє собою сукупність архітектури і прикладних програмних інтерфейсів. Архітектуру утворюють основні поняття і зв'язки між ними. Прикладні програмні інтерфейси характеризують набір сервісів, що надаються елементами середовища. Вони описуються в термінах синтаксису і семантики Java-класів та інтерфейсів.

До числа основних понять архітектури JavaBeans відносяться компоненти і контейнери. Контейнери можуть включати в себе безліч компонентів, утворюючи загальний контекст взаємодії з іншими компонентами і з оточенням. Контейнери можуть виступати в ролі компонентів інших контейнерів.

Неформально компонент ("кавове зерно" - *Java Bean*) можна визначити як багаторазово використовуваний програмний об'єкт, що допускає обробку в графічному інструментальному оточенні і збереження в довготривалій пам'яті. З реалізаційної точки зору компонент - це Java-клас і, можливо, набір асоційованих додаткових класів.

Кожен компонент має набір методів, доступних для виклику з інших компонентів і / або контейнерів.

Компоненти можуть мати *властивості*. Сукупність значень властивостей визначає стан компонента. Властивості можуть бути доступні для читання та / або запис за допомогою методів вибірки і установки.

Компоненти можуть породжувати *події* (бути джерелами подій), сповіщаючи про них інші компоненти, що зареєструвались в якості підписників. Повідомлення полягає у виклику певного методу об'єктів-підписників.

Типовим прикладом події є зміна властивостей компонента. У загальному випадку компонент може надавати підписку отримувати інформацію про зміну та право забороняти зміни.

Методи, властивості і події утворюють набір афішованих *характеристик* компонента, тобто характеристик, доступних інструментальному оточенню і інших компонентів. Цей набір може бути з'ясований за допомогою механізму інтроспекції.

Стан компонентів може бути збережений в довготривалій пам'яті. Наявність методів для подібного збереження виділяє компоненти JavaBeans серед довільних Java-класів.

Компоненти JavaBeans можуть упаковуватися для більш ефективного зберігання та передачі по мережі. Опис відповідного формату є частиною специфікацій JavaBeans.

## 9.4 Життєвий цикл компонентів JavaBeans

Життєвий цикл компонентів JavaBeans можна підрозділити на три етапи:

- розробка та реалізація компонента;
- збірка застосунків із компонентів;
- виконання застосунка.

Розробка та реалізація компонентів JavaBeans по суті не відрізняється від створення довільних Java-об'єктів, хоча і може включати реалізацію специфічних методів.

Збірка застосунків виконується, як правило, в інструментальному оточенні, що дозволяє проаналізувати характеристики компонентів, налаштувати значення властивостей, зареєструвати підписку на отримання подій, організувавши тим самим взаємодію компонентів. Розробник компонента може реалізувати спеціальні методи для використання виключно в інструментальному оточенні.

Компоненти взаємодіють між собою і з інструментальним оточенням. Взаємодія здійснюється двома способами - викликаючи метод і поширенням подій.

Специфікації JavaBeans описують тільки локальну взаємодію компонентів, здійснену в межах однієї віртуальної Java-машини. (Нагадаємо, втім, що Java-аплети розраховані на передачу по мережі, так що можливо зібрати застосунок з компонентів, спочатку розподілених по мережі.) Видалені об'єкти можуть зв'язуватися по протоколах архітектури CORBA, за допомогою віддаленого виклику методів (*Remote Method Invocation* – RMI) або іншими способами, що не відносяться до області дії специфікації JavaBeans.

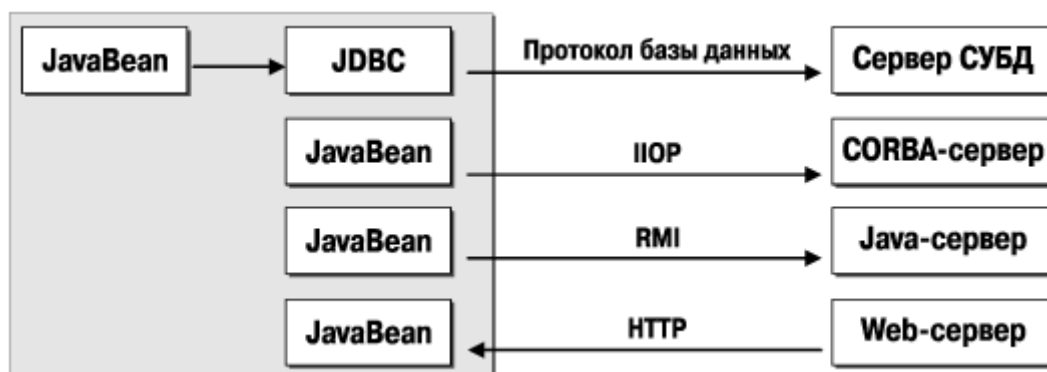


Рисунок 9.1 – Нелокальна взаємодія компонентів

### Практичне завдання:

Необхідно створити компонент JavaBean, що працює в ролі таймера. Він повинен виводити кількість спрацьовувань з моменту активації. Інтервал таймера задається у властивостях компонента.

### Контрольні питання

1. Що таке JavaBeans
2. Де використовуються компоненти JavaBeans?
3. Що являє собою процес створення компонента?
4. Етапи життєвого циклу компонентів JavaBeans?
5. Що описують специфікації JavaBeans?
6. Поняття моделі JavaBeans.
7. Які властивості компонентів Java Beans?

## ЛЕКЦІЯ 10. ОСНОВИ МЕРЕЖНОЇ ВЗАЄМОДІЇ

### План

- 10.1 Комп'ютерні мережі.
- 10.2 Мережна взаємодія.
- 10.3 Централізовані мережі.
- 10.4 Децентралізовані мережі.
- 10.5 Типи комп'ютерних мереж.

### 10.1 Комп'ютерні мережі

Інформаційно-комунікаційні технології, що з'явилися у другій половині XX ст., суттєво змінили життя людства. Саме вони створили передумови формування інформаційного суспільства, в якому визначальну роль відіграють інформація та нові знання. Саме в такому суспільстві ми з вами сьогодні живемо.

Перші ЕОМ були призначені лише для швидкої обробки числових даних. Згодом обчислювальна техніка стала широко використовуватися в наукових дослідженнях, виробництві, освіті, побуті тощо. У користувачів віддалених один від одного комп'ютерів з'явилася потреба у швидкому обміні даними. Для цього було запропоновано об'єднати комп'ютери в єдину систему і таким чином передавати дані від одного комп'ютера до іншого. Так були створені комп'ютерні мережі.

*Комп'ютерна мережа* — це сукупність комп'ютерів та інших пристроїв, зв'язаних каналами передавання даних.

Комп'ютерні мережі забезпечують спільний доступ до даних. У мережі виділяють комп'ютери, на яких розміщують великі масиви даних, а користувачі інших комп'ютерів мережі одержують доступ до них. Це дає можливість, наприклад, людям, котрі працюють над одним проектом, використовувати дані, створені іншими, тобто працювати над проектом одночасно.

За допомогою комп'ютерної мережі стає можливим спільне користування периферійними пристроями: принтерами, сканерами, модемами тощо. Невигідно мати їх біля кожного персонального комп'ютера, наприклад, у комп'ютерному класі або в банку.

Комп'ютерні мережі також дозволяють у короткі терміни розв'язувати складні пуккові та інженерні задачі (прогнозування стихійних лих, проектування аерокосміч-апаратів, обробка знімків Землі, отриманих зі супутників, моделювання й аналіз периментів у фізиці тощо). У 2006 р. в Києві відкрито Центр суперкомп'ютерних обчислень. Створення комп'ютерних мереж відкрило нові можливості для електронного зв'язку. Сьогодні люди, що мають комп'ютери, можуть спілкуватися між собою, незважаючи на віддаль і час. З появою комп'ютерних мереж комп'ютер став своєрідним вікном у величезний світ інформації.

Основне призначення всіх комп'ютерних мереж — це спільний доступ до мережних ресурсів (апаратного забезпечення комп'ютерів, периферійних пристроїв), спільне використання даних та швидкий обмін ними, спільне використання програмного забезпечення.

## 10.2 Мережна взаємодія

Мережна взаємодія передбачає віддалений доступ до мережних ресурсів та відбувається за технологією. Залежно від повноважень комп'ютери в мережі розподіляються на сервери та клієнтів.

*Клієнт* — це комп'ютер користувача, який здійснює запит, *сервер* - комп'ютер, що обробляє цей запит і відповідає на нього.

Звертаємо вашу увагу: сервером та клієнтом називаються як *комп'ютери* в мережі, так і *програмне забезпечення*, що працює на цих комп'ютерах.

## 10.3 Централізовані мережі

У *централізованих мережах* виділяється один потужний комп'ютер — *виділений сервер*, що виконує основні функції з організації роботи мережі. Такі мережі ще називаються „*клієнт–виділений сервер*”. Усі клієнти отримують доступ до ресурсів мережі через сервер.

На сервері встановлюється спеціальна операційна система (наприклад, Linux). Операційна система дозволяє організувати і контролювати роботу комп'ютерів і користувачів у мережі, надавати кожному користувачеві певні права доступу до ресурсів і даних цієї мережі. Для цього кожен користувач отримує ім'я користувача (логін) та пароль для входу до мережі.

Прикладами такої мережі можуть бути комп'ютерні мережі банків, корпорацій, вищих навчальних закладів, деяких шкіл м. Києва та інші.

Перевагами централізованих комп'ютерних мереж є висока швидкість обміну даними і можливість розподіляти права доступу користувачів у них. Але суттєвим недоліком є те, що при виході з ладу сервера вся мережа перестає працювати.

## 10.4 Децентралізовані мережі

У *децентралізованих мережах* немає виділеного сервера, будь-який комп'ютер може бути як сервером, так і клієнтом. Такі мережі ще називаються багаторанговими. Як клієнт, комп'ютер в одноранговій мережі може здійснювати запит щодо доступу до ресурсів інших комп'ютерів мережі. Як сервер, комп'ютер повинен обробляти запити від інших комп'ютерів мережі та надавати потрібні дані.

В *одноранговій мережі* всі комп'ютери мають однакові права (*ранги*) щодо доступу до ресурсів кожного й до периферійних пристроїв. Кожен



користувач мережі може на своєму жорсткому диску визначити папки і файли, які він надає для загального користування.

У таких мережах на всі комп'ютери встановлюється операційна система, яка забезпечує їм рівні можливості.

Перевагою однорангових мереж є працездатність мережі при виході з ладу будь-якого з комп'ютерів, а недоліком — неможливість розподіляти права клієнтів щодо роботи в мережі.

Прикладом такої мережі може бути мережа комп'ютерного класу у більшості шкіл.

### **10.5 Типи комп'ютерних мереж**

Об'єднані в мережу комп'ютери можуть бути розташовані в одній кімнаті, одному будинку, районі, місті, країні чи навіть у різних країнах. У багатьох школах України комп'ютери, встановлені в комп'ютерному класі, у кабінетах адміністрації, бібліотеці, кінолекційній залі та інших кабінетах, об'єднані в мережу.

#### **Контрольні питання**

1. Поняття комп'ютерних мереж.
2. Мережна взаємодія.
3. Що таке централізовані мережі?
4. Що таке децентралізовані мережі?
5. Які типи комп'ютерних мереж ви знаєте?

## ЛЕКЦІЯ 11. ОСНОВИ JAVA EE

### План

11.1 Платформа Java EE.

11.2 Масштабовність архітектури Java EE.

11.3 Проблеми, пов'язані з продуктивністю і масштабованістю.

### 11.1 Платформа Java EE

*Java Platform, Enterprise Edition*, скорочено Java EE (до версії 5.0 - Java 2 Enterprise Edition або J2EE) - набір специфікацій та відповідної документації для мови Java, яка описує архітектуру серверної платформи для задач середніх і великих підприємств.

Специфікації деталізовані настільки, що забезпечують переносимість програм з однієї реалізації платформи на іншу. Основна мета специфікацій – забезпечити масштабованість застосунків і цілісність даних під час роботи системи. J2EE багато в чому орієнтована на використання її через веб як в інтернеті, так і в локальних мережах. Вся специфікація створюється і затверджується через JCP (*Java Community Process*) в рамках ініціативи Sun Microsystems Inc.

J2EE є промисловою технологією і в основному використовується у високопродуктивних проектах, в яких необхідна надійність, масштабованість, гнучкість.

Популярності J2EE також сприяє те, що Sun пропонує безкоштовний комплект розробки, SDK, що дозволяє підприємствам розробляти свої системи, не витрачаючи великих коштів. У комплект ліцензії для розробки входить *сервер застосунків*.

Платформа Java EE створювалася в розрахунок на розробку застосунків типу "*підприємство-покупець*" (*business-to-consumer* – B2C) і "*підприємство-підприємство*" (*business-to-business* – B2B). У якийсь момент підприємства відкрили для себе можливості Інтернету і почали використовувати їх для розвитку існуючих бізнес-відносин зі своїми партнерами і клієнтами. Ці інтернет-застосунки часто працювали у зв'язці з раніше створеними системами для корпоративної інтеграції (*enterprise integration system* – EIS). Цей акцент на B2C, B2B і EIS-застосунки знайшов своє відображення і в найбільш популярних тестах, таких як ECperf 1.1, SPECjbb2005 і SPECjAppServer2004, призначених для вимірювання продуктивності і масштабованості Java EE-серверів. Навіть стандартний приклад програми на Java – PetStore є нічим іншим, як застосунком для електронної комерції.

### 11.2 Масштабовність архітектури Java EE

У тестах також відображено безліч явних і неявних припущень щодо масштабованості архітектури Java EE:

- З клієнтської точки зору, найбільш важливою характеристикою, що впливає на продуктивність, є *пропускна здатність*.
- Головним фактором, що визначає продуктивність, є *довжина транзакції*, тому роботу застосунку можна прискорити, скоротивши тривалість кожної транзакції.
- Транзакції в основному *ізолювані* одна від одної.
- У більшості транзакцій, за винятком тривалих, змінюється *стан невеликого числа об'єктів*.
- Тривалість транзакцій *обмежується продуктивністю* сервера застосунків і систем EIS, що працюють в тому ж адміністративному домені.
- Витрати на передачу інформації по мережі (у разі звернення до локальних ресурсів) компенсуються перевагами використання *пулів з'єднань*.
- Тривалість транзакцій може бути знижена шляхом *оптимізації* конфігурації мережі та інвестицій в мережеве апаратне і програмне забезпечення.
- *Управління даними та контентом* знаходиться повністю у відомстві застосунка. При відсутності будь-яких залежностей від зовнішніх сервісів основним чинником, що обмежує передачу контенту споживачеві, є *пропускна здатність* каналу передачі даних.

Ці припущення породили нижчеперелічені принципи, які стали основою для розробки Java EE API:

- *Синхронні інтерфейси застосунків (API)*. У більшості випадку в Java EE використовуються синхронні API за винятком, мабуть, тільки великовагового та незручного у використанні сервісу повідомлень JMS (Java Message Service). Синхронний підхід забезпечує швидше зручність, ніж продуктивність. Синхронні інтерфейси легші у використанні і легше оптимізуються. Необмежений потоковий паралелізм може швидко привести до серйозних проблем, тому його використання при розробці застосунків Java EE вкрай не рекомендується.
- *Граничні пули потоків*. Досить швидко з'ясувалося, що потоки є досить важливим типом ресурсів, і продуктивність серверів застосунків різко погіршується, якщо кількість потоків перевищує певне порогове значення. Однак, беручи до уваги припущення, що всі операції повинні виконуватися за короткий час, їх можна розподілити між невеликим числом потоків для збереження високої пропускної здатності обробки запитів.
- *Обмежені пули з'єднань*. Складно досягти оптимальної продуктивності при роботі з базою даних, використовуючи єдине з'єднання. Зрозуміло, деякі операції можуть виконуватися паралельно, але збільшення числа з'єднань покращує продуктивність

тільки до певної межі, після якого відкриття подальших з'єднань призводить до істотного падіння швидкості роботи. Найчастіше кількість з'єднань з базою даних менше числа доступних потоків в пулі сервлету. Тому використовуються також пули з'єднань, завдяки яким серверні компоненти, такі як сервлети чи об'єкти EJB (*Enterprise Java Bean*), можуть отримувати з'єднання і повертати їх у пул після використання. Якщо доступних з'єднань немає, компонент переходить в режим очікування, блокуючи виконання потоку. Ця затримка зазвичай невелика, оскільки всі інші компоненти швидко повертають з'єднання в пул.

- *Фіксовані з'єднання з ресурсами.* Передбачається, що застосунки повинні використовувати невелику кількість зовнішніх ресурсів, отримуючи до них доступ через фабрику сполук, посилення на яку, в свою чергу, може бути отримане за допомогою інтерфейсу JNDI (*Java Naming and Directory Interface*) або механізму ін'єкції залежностей (*dependency injection*) в EJB 3.0. Фактично єдиним серйозним програмним інтерфейсом Java EE, що підтримує можливість підключення до ресурсів різних EIS, є API корпоративних Web-сервісів. Решта інтерфейси, як правило, працюють тільки з наперед визначеним списком ресурсів, і вважається, що для відкриття з'єднань з ресурсами потрібні лише параметри доступу (мандат користувача).

### 11.3 Проблеми, пов'язані з продуктивністю і масштабованістю

Платформа Java EE спочатку проектувалася для управління сервісами, які працюють з ресурсами, що належать до одного *адміністративному домену*. Малося на увазі, що в системах корпоративної інтеграції всі транзакції будуть короткими, а запити будуть оброблятися швидко, що дозволить платформі виконувати велику кількість операцій.

В даний час з'являється безліч нових рішень і технологій, таких як *пірінгові* (peer-to-peer) і *сервіс-орієнтовані* (SOA) архітектури, а також нові типи Web-застосунків, які отримали неофіційну назву Web 2.0, які не вписуються в рамки, встановлені Java EE. Зокрема, їм доводиться мати справу з тривалими транзакціями. Тому використання традиційних підходів і принципів Java EE при розробці застосунків Web 2.0 тягне за собою серйозні проблеми, пов'язані з продуктивністю і масштабованістю.

#### Контрольні питання

1. Особливості платформи Java EE.
2. В чому полягає масштабовність архітектури Java EE?
3. Які проблеми, пов'язані з продуктивністю і масштабованістю JavaEE можуть виникнути?

## ЛЕКЦІЯ 12. СЕРВЛЕТИ JSP

### План

12.1 Сервлети JSP.

12.2 Переваги технології JSP.

### 12.1 Сервлети JSP

У Java-програмуванні прийнято розрізняти *Java-застосунки* та *Java-аплети*. Перше з точки зору програмування цінне саме по собі, тому що нічого крім області виконання Java не вимагає. А друге цікаво тільки в контексті здатності браузерів виконувати Java-код аплетів.

Є, однак, і третя сторона Java – це програми, які виконуються *http-серверами*. Ось вони і називаються *сервлети*. Звичайно, сам по собі сервер не може виконувати Java-код. В сервер вбудовується модуль, який викликає Java-машину, виконуючу сервлети.

Як аплет не є самостійною Java-програмою, так і сервлет нею також не є. І те й інше, умовно кажучи, «латочки», які програміст вбудовує в більш загальний Java-код. Прикладному програмісту дається можливість написати тіла заздалегідь визначених через механізм інтерфейсів методів.

Сервлети в деякому сенсі схожі на CGI-скрипти. Як і в скриптах у сервлетів жорстко визначений механізм отримання даних, тобто вхідний потік, і механізм формування вихідного потоку. При цьому сервлетам крім усього іншого ще дозволено отримувати точно такі ж значення, як ті, що визначені в змінних оточення скриптів.

У свою чергу сервлети є базою для *Java Server Pages*. Про цей механізм кажуть, що він є повнофункціональним аналогом *Active Server Pages* компанії Microsoft. Насправді механізм JSP ширше ASP, і набагато більш гнучкий.

JSP – це послідовний розвиток ідеї вбудовування програмного коду в HTML і XML сторінки. Тільки це не проста мова SSI і не VBScript, а повноцінні Java-конструкції.

Насправді це тільки з точки зору людини, яка створює JSP, йдеться про «начинення» кодом старого доброго HTML. З точки зору сервера така сторінка - це *сервлет*. При чому реально трансляція *файлу* із Java-кодом, що потім зберігається в пам'яті, відбувається тільки при першому зверненні до JSP-сторінки. При всіх наступних зверненнях ніякої трансляції відбуватися вже не буде, а буде використовуватися раніше відтрансльований сервлет.

### 12.2 Переваги технології JSP

З вище сказаного слідують ті плюси, про які так любить розмірковувати фахівці, які «захворіли» даною технологією, не всі вони є абсолютними «плюсами», але тим не менше їх слід перерахувати:

- *Ефективність*. Не треба створювати новий процес. У стандартній ситуації все вже в «голові» і відразу починає виконуватися. Зрозуміло, що тут сервлети порівнюються зі CGI-скриптами. В принципі, навіть інтегрований в браузер Perl, модуль *mod\_perl* в Apache, наприклад, не повинен теоретично забезпечувати такої продуктивності, яку обіцяють сервлети.
- *Єдність середовища розробки програмного забезпечення*. Вивчив Java і пиши собі код на здоров'я. Не треба вчити інші мови програмування, наприклад, Perl, VBScript або C. Правда, сама ця мова, м'яко кажучи, не дуже проста.
- *Потужність*. Java – мова багата. Написати на ньому можна багато чого. Правда, Perl в цьому сенсі не гірше. У всякому разі, там, де не треба розписувати віконечка і кнопки екранних інтерфейсів. А програмування на стороні сервера – це якраз такий випадок. Perl та Java – це основні мови розробки Web-застосунків. З точки зору складності вони приблизно однакові. Правда, кожна по своєму.
- *Переносимість*. Інтерпретовані програми по великому рахунку всі перенесимі – був би на відповідній платформі потрібний інтерпретатор. Java краще тільки в тому плані, що для неї централізовано підтримується єдина специфікація мови.
- *Безкоштовність*. Може бути для наших закордонних колег це і велика перевага, але для нас, де 80% рідного Web-а працює на безкоштовному Apache, це норма. Але добре, що й за кордонами магістральний шлях розвитку поки залишається безкоштовним.

### **Контрольні питання**

1. Що собою являють сервлети JSP?
2. Які переваги технології JSP ви знаєте?

## ЛЕКЦІЯ 13. РІВЕНЬ БІЗНЕС-ЛОГІКИ В JAVA EE

### План

13.1 Класифікація рівнів бізнес-логіки в Java EE.

13.2 Enterprise JavaBeans (EJB).

### 13.1 Класифікація рівнів бізнес-логіки в Java EE

Існують такі рівні *бізнес-логіки* технології Java EE:

- *Верхній рівень* (рівень *дизайнерів, верстальників*) – рівень представлення, визначає інтерфейс користувача. Це може бути HTML, CSS, Flash, зображення та інші технології, за допомогою яких створюється зовнішній вигляд застосунка. Рекомендується вивчати: JSF.
- *Середній рівень* (рівень *програмістів*) – рівень бізнес-логіки. Рівень на якому і відбувається програмування. Створюються класи, реалізується бізнес логіка програми (наприклад, авторизація користувача в системі, обробка тих чи інших подій). Рекомендується вивчати: EJB.
- *Нижній рівень* (також рівень *програмістів*) – рівень даних, забезпечує зберігання даних, найчастіше в базах даних. Рекомендується вивчати: JPA.

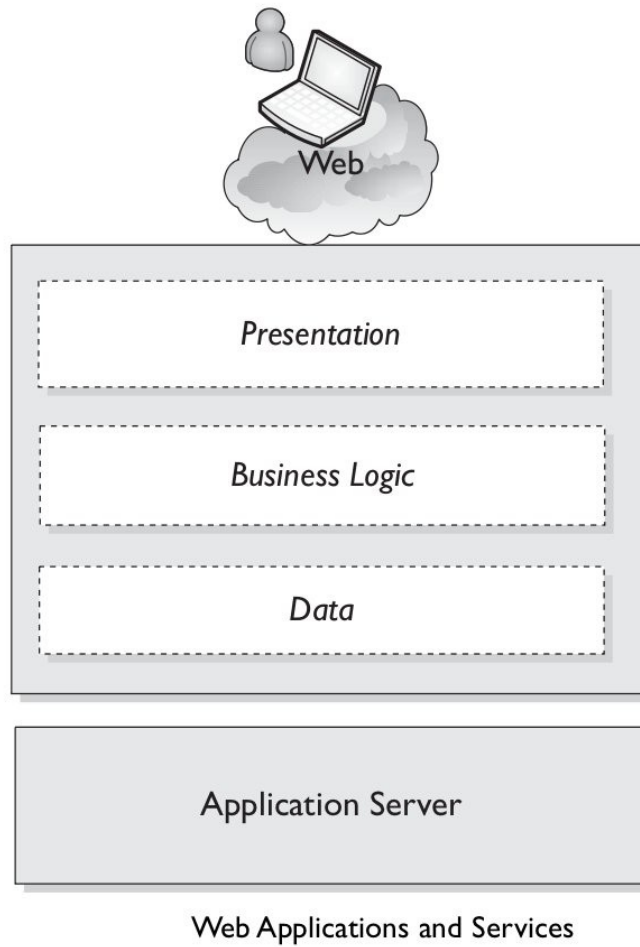


Рисунок 13.1 – Web служби-застосунки

Технології Java EE, за допомогою яких можна реалізувати роботу того чи іншого рівня наведено на рисунку 13.2.



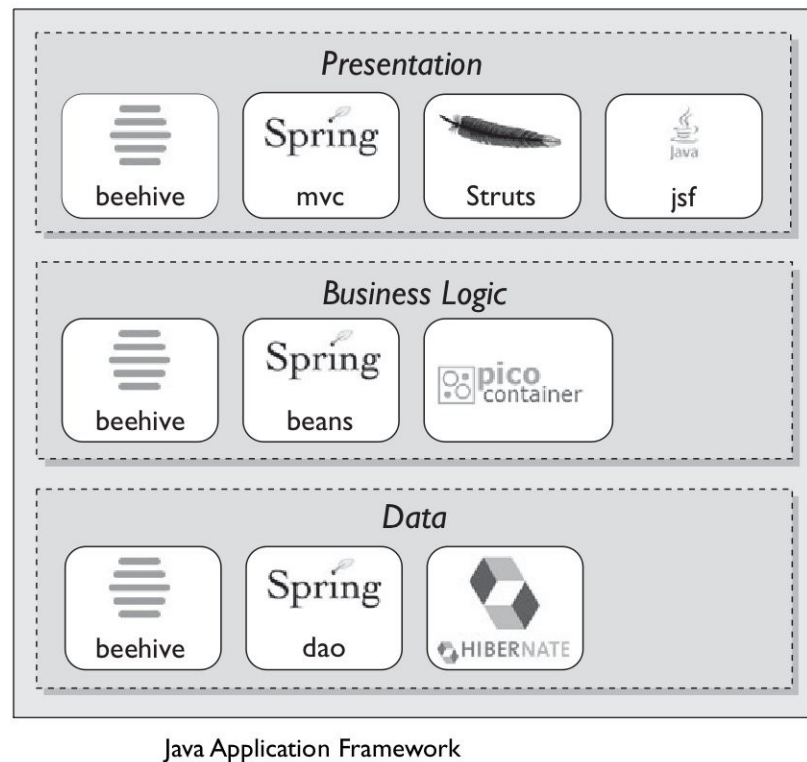


Рисунок 13.2 – Технології Java EE, за допомогою яких можна реалізувати роботу того чи іншого рівня

На рисунку 13.2 на рівні даних відсутній *EclipseLink* (попередня його назва *Oracle TopLink*. *TopLink* був переданий Eclipse корпорацією Oracle).

### 13.2 Enterprise JavaBeans (EJB)

У минулому зміни, що вносяться в специфікації EJB, робилися без втрати сумісності. При цьому у версії 3,0 був серйозно змінений синтаксис програмної моделі EJB. Тим не менш минулі програмні моделі - з версій 2,1 і більш ранніх - як і раніше підтримуються. Навіть програмна модель версії 3,0 залишилась несумісна, що дозволяє реалізовувати клієнтські і серверні компоненти на різних версіях специфікації:

- Сесійні об'єкти на даний момент досить *стабільні*, і фундаментальних змін їх поведінки не передбачається, хоча деякі нові функції можуть з'явитися.
- Також не повинно бути змін в об'єктах, керованих повідомленнями (*Message-Driven Beans*).
- Частина специфікації, що відповідає за СМР / ВМР-збереження, досить стабільна, але її можуть замінити нові програмні інтерфейси JPA (*Java Persistence API*) у випадках, коли потрібно збереження даних в реляційних базах даних, і архітектура JCA (*Java EE Connector Architecture*), у решти випадках.
- Одним з можливих серйозних змін в наступній версії специфікації може стати інтеграція сесійних об'єктів з об'єктами, керованими

повідомленнями. Таким чином, з'явиться новий тип компонентів, які можуть взаємодіяти як з синхронною, так і по асинхронною схемою. Але на даний момент все говорить про те, що розвиток технології буде відбуватися розумно і без втрати сумісності.

**Контрольні питання**

1. Які рівні бізнес-логіки Java EE ви знаєте?
2. За допомогою яких технологій Java EE можна реалізувати роботу того чи іншого рівня бізнес-логіки?

## ЛЕКЦІЯ 14. ОСНОВНІ ЕЛЕМЕНТИ ТЕХНОЛОГІЇ JSF

### План

- 14.1 Загальні поняття.
- 14.2 Розробка моделі.
- 14.3 Створення представлення.
- 14.4 Налаштування переадресації.

### 14.1 Загальні поняття

Не зважаючи на свою простоту і досить великі можливості, технологія JSP встигла морально застаріти і на зміну їй прийшли потужні і функціональні фреймворки. *Java Server Faces* став неймовірно популярним серед Java Web-розробників.

Сучасне, яскраве і в той же час зручне для підтримки веб-застосунків створити досить важко. Це й зрозуміло – якщо його «ядро» «вилизано» і підігнано під всі правила рефакторинга, то інтерфейс, побудований на компонентах HTML, буде виглядати не дуже презентабельно, а в разі використання AJAX / JQuery потрібні додаткові зусилля і витрати часу. Концентрація ж на інтерфейсі, як показує практика, залишає бажати кращого для бізнес-логіки і рівня доступу до даних, що так само не є добре.

Для розробки більшої частини веб-застосунків програмісту потрібні засоби автоматизації, якими є численні вузькоспеціалізовані або універсальні фреймворки. У загальному випадку такий фреймворк повинен бути і генератором HTML-сторінок, і менеджером вводу-виводу, і навігатором по дереву проекту. На думку багатьох розробників, найкращою архітектурою, яку можуть надати фреймворки, є *Модель-Представлення-Контролер* (*Model-View-Controller*, MVC). У застосунках такого роду контролер відповідає за прийом даних від користувача і видачу відповідного подання, подання формує для браузера HTML сторінку, а модель містить дані, отримані з веб-форм і ту інформацію, яку необхідно вивести на екран. Цей підхід дозволяє захистити рівень представлення даних від бізнес-логіки й, слід зазначити, робить це добре. Фреймворків, що підтримують MVC, є достатньо, але *Java Server Faces* (JSF) – стандарт для застосунків Java EE, і в цьому його перевага. По-перше, технологія активно розвивається Oracle, а по-друге, для неї на даний момент створено безліч бібліотек, що дозволяють використовувати нестандартні UI компоненти, засновані на jQuery. JSF – добре спроектована і проста у використанні платформа, що поєднує компонентний підхід до програмування і легкі POJO для збору і зберігання даних.

Якщо міркувати формалізованими термінами мови, тоді:

- *Представлення* – це файл \*.jsf або \*.xhtml, відповідальний за вивід даних в браузер і містить посилання на конкретні дані в моделі.

- *Модель* – *Java Bean*, який зберігає ту чи іншу інформацію в приватних полях і надає для них методи читання/запису властивостей поряд з методами обробки такої інформації.
- *Контролер* – це і є внутрішній механізм JSF, що дозволяє провести лінкування першого з другим.

JSF-застосунок зазвичай містить два типи компонентів, причому обидва гранично прості у використанні і узгоджуються з філософією POJO.

Сторінки JSF, створюються з XML тегів. Кожен тег представляє конкретний UI-компонент. Як веб-розробнику, вам не потрібно вдаватися в написання HTML розмітки або вставок на JavaScript, так як вони повністю генеруються компонентними тегамі JSF. Так як кожен компонент по-суті незалежний і містить певну поведінку (тобто «знає», як отримати свої дані і відмалювати себе в браузері), JSF надає підхід до програмування UI, дуже схожий на принцип оперування POJO.

Динамічні дані на JSF сторінках моделюються за допомогою POJO, так званими керованими компонентами (*JSF Backing Beans*). Його життєвий цикл управляється *контейнером*. Наприклад, ви можете пов'язати змінні в поточній сесії з відповідними полями *бекінг-біна*, таким чином переглядаючи зміни.

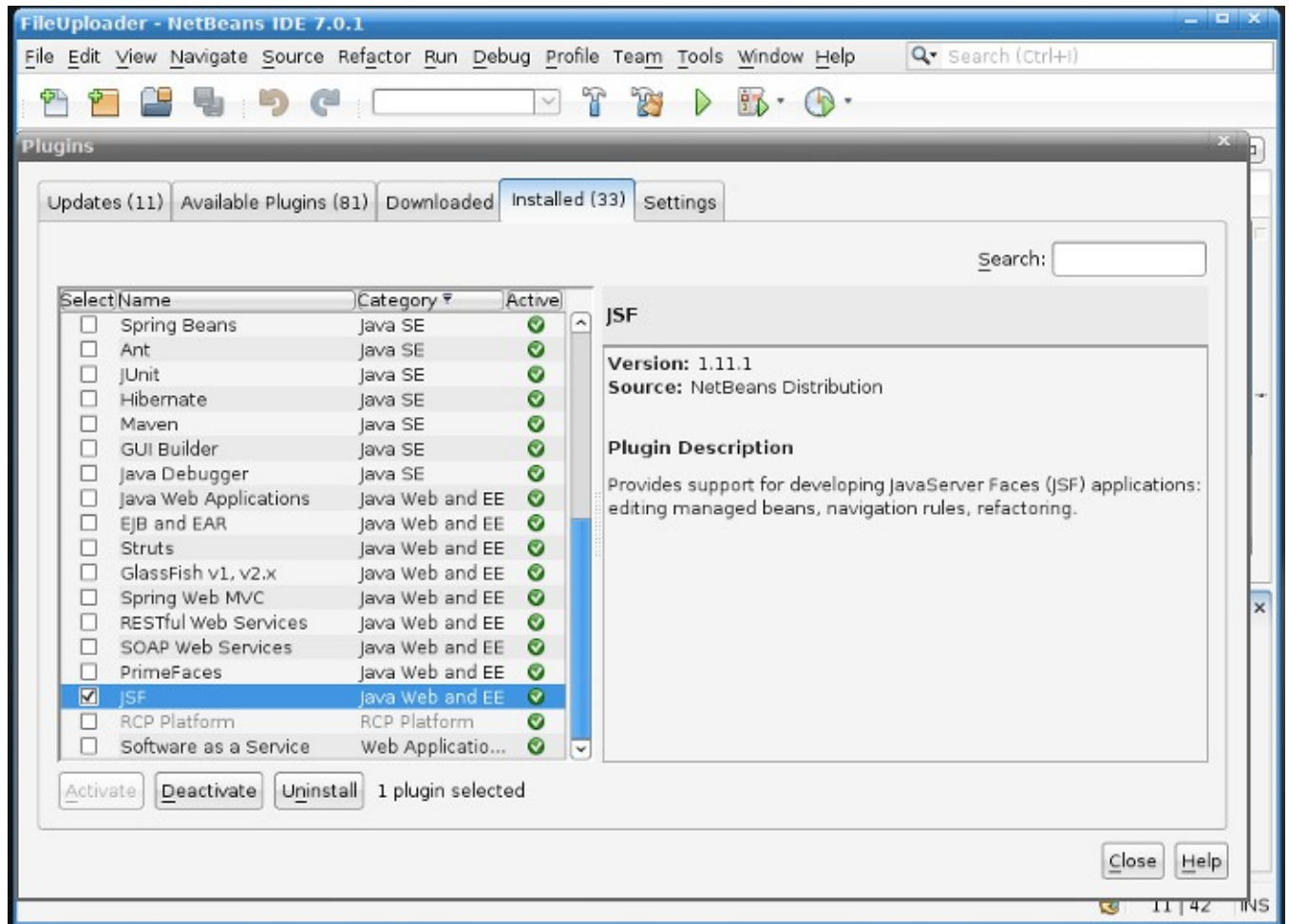
Компонентні моделі UI і POJO дозволяють JSF заручитися підтримкою різних середовищ розробки. Фактично, багато IDE для Java підтримують інтерактивні *drag-and-drop* розробники UI-інтерфейсу для JSF. Компонентна модель JSF також дозволяє розробляти бібліотеки компонентів, які значно розширюють функціональність фреймворку. Серед таких розробок величезною популярністю користуються *Prime Faces*, *IceFaces* і деякі інші проекти. Що ж стосовно недоліків, недоробок, багів і інших неприємностей - їх не надто багато. З розвитком платформи JavaEE багато з них були усунені або усуваються. Програмування JSF стало набагато простіше з приходом техніки анотацій, що дозволила відмовитися від складного і незграбного конфігурування компонентів за допомогою XML.

Отже, *Java Server Faces* вирішує багато історичних проблем JavaEE за допомогою введення прозорої реалізації архітектури *Model-View-Controller* і завдяки наданню ефективного компонентного підходу до розробки. Такий підхід дозволив стороннім розробникам «нарощувати» і збагачувати технологію, тому, за всіма передумовами, подальший розвиток JSF буде припинено ще не скоро.

Більшість серверів Java EE мають вбудовану підтримку JSF, тому вам необхідно лише скачати API, прикріпити його до проекту і приступати до розробки. Фреймворк поширюється в трьох версіях: JSF 1.2 (безнадійно застаріла), JSF 2.0 і JSF 2.1, отримати їх можна на сайті спільноти.

Насправді, ручне вбудовування JSF в проект – справа неприємна. Простіше доручити це завдання IDE, для прикладу, Netbeans. Таким чином, при створенні веб-застосунків Вам слід всього лише вказати *Java*

*Server Faces* в якості допоміжної платформи – інтеграція відбудеться автоматично. Рекомендується вибирати версію JSF 2.0 як найнадійнішу і найстабільнішу. Якщо Netbeans не має її в наявності – зверніться до меню плагінів, там він точно присутній (рисуюнок 14.1).



Рисуюнок 14.1 – Меню плагінів

## 14.2 Розробка моделі

Модель представлена керованим компонентом (JSF Managed Bean), який по суті своїй є POJO і має поля для зберігання даних. Відмінність такого компонента від простого об'єкта Java полягає в тому, що Managed Bean може бути викликаний із JSF-сторінки.

Як дати зрозуміти контейнеру, що він має справу з керованим компонентом, а не з чимось іншим? Для цього є два підходи. Перший, більш старий, передбачає декларування такого класу в спеціальному файлі конфігурації *faces-config.xml*. Другий – використання анотації `@ManagedBean` перед класом керованого компонента.

Розглянемо перший спосіб. Його перевага – це можливість конфігурування проекту "на льоту":

```

<?xml version='1.0' encoding='UTF-8'?>
<faces-config version="2.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-
facesconfig_2_0.xsd">
    <managed-bean>
        <managed-bean-
name>myManagedBean</managed-bean-name>
        <managed-bean-
class>beans.MyManagedBean</managed-bean-class>
        <managed-bean-scope>request</managed-
bean-scope>
    </managed-bean>
</faces-config>

```

Властивість **managed-bean-name** визначає ім'я керованого компонента. Саме з цього імені і буде проводитися звернення до моделі з рівня представлення.

**managed-bean-class** – відповідно клас компонента.

**managed-bean-scope** – атрибут відповідає за життєвий цикл компонента. `request`, як у прикладі, означає, що компонент буде перестворюватися кожного разу при зверненні.

Другий спосіб простіше: він дозволяє відмовитись від XML-конфігурації. Слід лише анотувати потрібний клас як `@ManagedBean` і `@XXScoped`, де `XXX` може приймати значення `Request`, `Session` і т.д.

```

package beans;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.RequestScoped;
@ManagedBean
@RequestScoped
public class myModelBean {
    public myModelBean() {
    }
    String username;
    String userpass;
    String loginMessage;
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
}

```

```

    }
    public String getUserpass() {
        return userpass;
    }
    public void setUserpass(String userpass) {
        this.userpass = userpass;
    }
    public String getLoginMessage() {
        return loginMessage;
    }
    public void setLoginMessage(String
loginMessage) {
        this.loginMessage = loginMessage;
    }
    public String login() {
        if(username.equals("root") &&
userpass.equals("pass")){
            loginMessage = "Access granted!";
            return "accept";
        } else {
            return "deny";
        }
    }
}

```

Звернемо увагу на сам клас. Ми маємо три поля, які будемо використовувати при зверненні до моделі – `username`, `userpass` і `loginMessage` (обумовили, що застосунок буде виконувати просту функцію: приймати ім'я і пароль, перевіряти їх і видавати повідомлення про результат входу). Для полів передбачені публічні методи отримання і установки, які будуть використовуватися контролером. Публічний метод `login()` перевіряє значення полів імені та пароля, порівнює їх з деякими значеннями і видає результат у вигляді рядка. На цьому розробку найпростішої моделі закінчена.

### 14.3 Створення представлення

Як було сказано на початку, представленням є деякі файли з розширенням `*.jsf` (в JSF 1.2) або `*.xhtml` (в JSF 2.0), не будемо вникати в їх відмінності. За зовнішнім виглядом уявлення нагадує HTML-розмітку, проте деякі особливості таки мають місце.

```

<?xml version='1.0' encoding='UTF-8' ?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

```

```

<h:head>
    <title>Hello World App</title>
</h:head>
<h:body>
    <h:form>
        <h:panelGrid columns="2">
            <h:outputText
value="Username"/>
            <h:inputText id="username"
value="#{myModelBean.username}"/>
            <h:outputText
value="Password"/>
            <h:inputSecret id="userpass"
value="#{myModelBean.userpass}"/>
            <h:commandButton value="Login!"
action="#{myModelBean.login()}"/>
            <h:outputText
value="#{myModelBean.loginMessage}"/>
        </h:panelGrid>
    </h:form>
</h:body>
</html>

```

Впадає в очі цікава річ: деякі теги використовуються разом з префіксами простору імен. Префікс - це покажчик, яку саме бібліотку тегів використовувати. Наприклад **h** = HTML, **f** = *Facelets*, **p** = *Primefaces* і т.д. Можливо ви помітили що параметр **value** у деяких компонентів виглядає вельми незвично. Це і є зв'язування подання з моделлю. Придивіться уважніше: `myModelBean` – це назва нашого керованого компонента, а `username`, `userpass`, `loginMessage` – назви його полів. Що стосується кнопки, тут теж все досить очевидно. При натисканні викликається метод `login()`, який працює з тими даними, які були внесені за допомогою поточної форми.

## 14.4 Налаштування переадресації

Цікавим моментом в керованому компоненті є повернення строкових значень функцією `login()`. Навіщо вони потрібні і чому саме текст, а не інші типи мови Java? Відповідь криється в тому, що контролер займається не тільки зв'язуванням моделі і JSF-сторінки. Крім усього іншого, він відповідає за переходи по дереву проекту.

До прикладу, в нашому застосунку, при невдалій спробі введення пароля та імені, метод `login()` повертає рядок `"deny"`, в разі успіху - `"accept"`. Так чому б не фіксувати ці повернення значень для навігації?



Контролер забезпечує це. Однак слід заздалегідь вказати такі правила переходу. Описуються вони в тому ж файлі *faces-config.xml*.

```
<faces-config version="2.0"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-
facesconfig_2_0.xsd">
    <navigation-rule>
        <from-view-id>/index.xhtml</from-view-id>
        <navigation-case>
            <from-action>#{myModelBean.login()}</from-
action>
            <from-outcome>accept</from-outcome>
            <to-view-id>/index.xhtml</to-view-id>
        </navigation-case>
        <navigation-case>
            <from-action>#{myModelBean.login()}</from-
action>
            <from-outcome>deny</from-outcome>
            <to-view-id>/error.jsp</to-view-id>
            <redirect/>
        </navigation-case>
    </navigation-rule>
</faces-config>
```

Розглянемо вищенаписане. Тег **<navigation-rule>** являє правило переходу, асоційоване зі сторінкою *index.xhtml*. У нього входять два вкладених тега **<navigation-case>** – вони представляють випадки переходу. Призначення інших тегів:

**<from-action>**      `#{myModelBean.login() }`      **</from-action>** – метод, ініціює випадок переходу

**<from-outcome>** `accept` **</from-outcome>** – його повернене значення

**<to-view-id>** `/ index.xhtml` **</to-view-id>** – сторінка-адресат

При запуску такого застосунка будемо бачити приблизно наступне (рисунок 14.2):

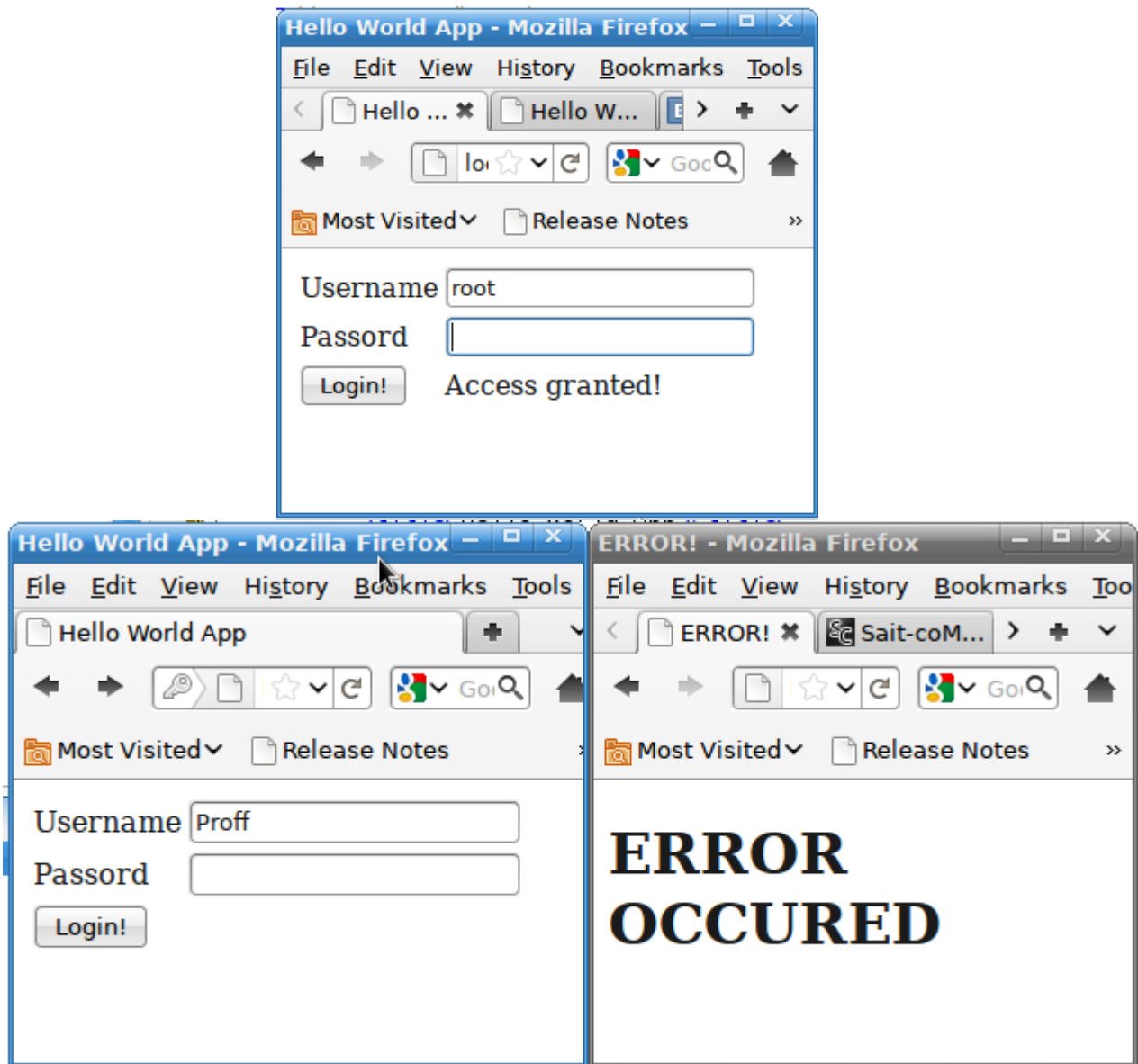


Рисунок 14.2 – Результати виконання застосунка у веб-браузері

### Контрольні питання

1. Архітектура JSF.
2. Використання моделі Model-View-Controller (MVC).
3. Компоненти Managed Bean.
4. Розробка JSF-застосунка.
5. Склад типового JSF-застосунку.
6. Етапи розробки JSF-застосунку.
7. Модель MVC у фреймворку Spring.

## ЛЕКЦІЯ 15. ОСОБЛИВОСТІ ВИКОРИСТАННЯ ТЕХНОЛОГІЇ JSF

### План

- 15.1 Технологія Java Server Faces. Концепції
- 15.2 Специфікації
- 15.3 Принципові засади та функціональність
- 15.4 Технічні особливості
- 15.5 Концептуальні частини JSF-проектів та їх зв'язування
- 15.6 Переваги і недоліки
- 15.7 Версії JSF

### 15.1 Технологія Java Server Faces. Концепції

*Java Server Faces (JSF)* — це технологія розробки розвинених (багатих) інтерфейсів для Web-застосунків (JSF близька до традиційних технологій розробки інтерфейсів — “*Swing* для *Web-застосунків*”), ключовими засадами якої є:

- компонентний підхід;
- вирішення на боці сервера (*server side* технологія);
- фреймворк-орієнтоване рішення з використанням архітектури MVC.

*Java Server Faces (JSF)* — це стандартизована специфікація (стандартний фреймворк з GUI-компонентами для Web-застосунків) за поданням Sun Microsystem (JSR 127 Java Community Process).

JSF є стандартом і є частиною ще одного стандарту — Java EE.

До стандартної специфікації фірмою Sun додається ще й “стандартна реалізація” — *Reference Implementation (RI)* — набір *jar-бібліотек* Sun(RI) JSF.

Є й альтернативні реалізації, найбільш цитована — *Apache MyFace*. Є й розширення, наприклад, *Apache Trinidad*.

Належність до стандарту надає наявність потужної підтримки від розробників (Sun, IBM, Oracle etc):

- розвинені бібліотечні надбудови;
- WISYWYG -засоби;
- інтегровані засоби розробки (IDE).

Дві складові JSF :

- *Java API* для “традиційних” GUI-компонентів, а також для реалізації управління їх станом, обробки подій та деяких інших практично важливих задач: валідації та конвертування даних, підтримки навігації між сторінками.
- Бібліотеки тегів для GUI-компонентів сторінок браузера.

Стандартом підтримується тільки бібліотека для традиційних (*html*) браузерів — JSP-бібліотека *html-basic.tld*

## 15.2 Специфікації

Користь технології JSF обумовлена, в першу чергу, наявністю *специфікації* JSF. Специфікація дозволяє розробляти JSF фреймворки з різним призначенням та різною внутрішньою структурою. Вона лиш гарантує, що фреймворк буде підпорядкований певній структурі. Але з іншого боку специфікація дуже обмежує еталонну реалізацію в освоєнні нових можливостей. Тобто наприклад аяx-технології такі як Ajax4JSF включають дуже багато інтеграційного коду який виникає через потребу в узгодженні еталонної реалізації з основними вимогами специфікації. Можна відмітити, що сама специфікація розроблена досить неоднорідно. Плюсами специфікації є: дерево компонентів, підтримка різних технологій представлення ([JSP](#), [Facelets](#)), підтримка різноманітних рендерерів — класів, що відповідають за відображення компоненту, підтримка обробки подій і перевіркою інформації, що вводиться, визначення навігації, а також підтримку інтернаціоналізації (*i18n*) і доступності (*accessibility*). Але є й недоліки: дуже великий обсяг коду для реалізації ітераційних компонентів, відсутності обробки повідомлень в середині ітераційного компоненту, великий обсяг шаблонного коду, який можна було б опустити, при реалізації власних компонентів (*custom component*), непередуманість певних архітектурних рішень в специфікації, щодо реалізації аяx, управління станом дерева компонентів, пошуку по дереву компонентів. Специфікація JSF 1.0 та 1.1 була розроблена завдяки *Java Community Process* як JSR 127, а JSF 1.2 як JSR 252. Майбутня JSF 2.0 буде розроблена як JSR 314. Сама специфікація не належить до жодної компанії і розроблюється групою експертів з таких відомих компаній як [Sun](#), [Oracle](#), [IBM](#), [Novell](#), [Macromedia](#), [BEA Systems](#), [Hewlett-Packard](#), [Siemens AG](#). Таким чином технологію JSF можна віднести до відкритих стандартів.

## 15.3 Принципові засади та функціональність

- Відокремлення бізнес-логіки від інтерфейсу, MVC-архітектура.
- Використання традиційної концепції графічного інтерфейсу GUI на компонентній основі:
  - підтримка управління станом компонент;
  - реалізація та обробка подій (на боці сервера!) з використанням моделі публікації подій

Якщо у *Model 2* та в *action-орієнтованих* фреймворках підтримка концепції подій має примітивний характер (використовується схема “*заявка–обробка*”), то JSF пропонує повноцінний набір подій, включаючи вибір пункту меню, натискання на кнопку і т.д.

- *Rendering Kit* з метою підтримки принципово різних технологій відображення сторінок (браузерних технологій).

## 15.4 Технічні особливості

- Використовується модель публікації подій.
- Підтримується DI-container — контейнер для залежних компонентів (залежність декларується у конфігураційному файлі фреймворку).
- Використовується концепція життєвого циклу (життєвого циклу JSF) з урахуванням потреб щодо:
  - валідації уведених даних;
  - конвертування даних.
- Використання декларативного підходу (*конфігураційних файлів*) для управління переходами між сторінками (для управління навігацією).
- Можлива інтеграція з іншими фреймворками (можна, наприклад, замінити стандартну підтримку DI у JSF на *DI /IoC Spring*).

## 15.5 Концептуальні частини JSF-проектів та їх зв'язування

- *Управляючі об'єкти (managed-bean)* – дозволяють управляти станом і поведінкою у проекті. Саме вони забезпечують зв'язок між шаром моделі (*Model*) і шаром представлення (*View*).
- *JSF-сторінки*. Вони фактично надають презентаційну частину – *View* архітектурного патерну MVC. JSF-сторінка розглядається як *дерево* з JSF-компонентів. Часто таке дерево називають презентацією сторінки (*View* сторінки). Компонент-корінь дерева у визначається парою тегів `<f:view>` та `</f:view>`.

При класичному web-проектуванні (для HTML-браузерів) на основі JSF традиційно використовуються JSP-сторінки з двома додатковими бібліотеками:

```
<%@ taglib uri="http://java.sun.com/jsf/html"
  prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core"
  prefix="f"%>
```

## 15.6 Переваги і недоліки

- генерація *серверної частини* інтерфейса користувача;
- базується на *компонентах* (ніякого HTML);
- наявна обробка *подій (event)* та *станів (states)*;
- різноманітні *view-технології* — не тільки HTML та JavaScript;
- розробка з урахуванням доступного *інструментарію*;
- наявний *стандарт* і він включений до Java EE;
- рольова *модель* розробки веб;

- потрібно *багато часу* для вивчення та освоєння технології;
- повинні бути *потужні* обчислювальні можливості серверу.

### 15.7 Версії JSF

- JSF 1.0 (11.03.2004) — первісний реліз по специфікації JSF;
- JSF 1.1 (27.05.2004) — виправлення помилок. Істотних змін немає;
- JSF 1.2 (11.05.2006) — оновлення, виправлення помилок;
- JSF 2.0 (2008) — об'єднання з Java EE 6.

#### *Підтримка IDE*

- [Eclipse JSF Tools Project](#);
- [NetBeans](#) Visual Web Pack.

#### **Контрольні запитання:**

1. У чому полягає відмінність JSF від більшості MVC-фреймворків?
2. Що можна віднести до недоліків JSF?
3. Призначення технології JSF.
4. Завдяки якій технології виникла JSF

## ЛЕКЦІЯ 16. WEB-СЛУЖБИ НА ПЛАТФОРМІ JAVA EE

### План

16.1 JAX-WS 2.0.

16.2 Асинхронні веб-служби.

16.3 Архітектура SOA.

Використання анотацій в платформі Java, істотно поліпшило і спростило підтримку її веб-служб. У цьому відношенні необхідно згадати наступні специфікації: JSR 224, "API Java для веб-служб на основі XML (JAX-WS) 2.0"; JSR 222, "Архітектура для прив'язки XML (JAXB) 2.0"; та JSR 181, "*Метадані веб-служб для платформи Java*".

### 16.1 JAX-WS 2.0

JAX-WS 2.0 являє собою новий інтерфейс API платформи Java EE для веб-служб. JAX-WS 2.0, що є наступником JAX-RPC 1.1, зберігає природну модель програмування RPC, вдосконалену за кількома напрямками: прив'язка даних, незалежність від протоколу і транспорту, підтримка стилю веб-служб REST і простота розробки.

Найважливіша відмінність від JAX-RPC 1.1 полягає в тому, що всі прив'язки даних тепер делеговані JAXB 2.0. Завдяки цьому для веб-служб на основі JAX-WS забезпечується можливість використання схем XML на 100%, що призводить до підвищення рівня взаємодії і простота використання. Ці дві технології відмінно інтегровані, так що відпадає необхідність у жонгливанні двома наборами засобів. При початку роботи з класами технології Java, JAXB 2.0 дозволяє створювати документи схеми XML, автоматично впроваджені в документ WDSL (мова опису веб-служб), в результаті чого користувачі позбавлені від виконання вручну цієї, схильної до помилок, процедури інтеграції.

У JAX-WS 2.0 без додаткового налаштування підтримуються протоколи SOAP 1.1, SOAP 1.2 і XML / HTTP. Розширюваності протоколів з самого початку приділялася величезна увага, і JAX-WS 2.0 дозволяє постачальникам підтримувати додаткові протоколи і кодування для підвищення продуктивності (наприклад, FAST *InfoSet*) або для спеціалізованих застосунків. Веб-служби, в яких використовуються вкладення для оптимізації передачі і прийому великих обсягів двійкових даних, виграють від використання стандарту MTOM / XOP (механізм оптимізації передачі повідомлень / упаковка XML з оптимізацією для двійкових даних) від W3C без негативного впливу на модель програмування. (Відомості про MTOM / XOP наведені на даній сторінці.) До технології Java EE 5 для визначення веб-служб були потрібні довгі, громіздкі дескриптори. Тепер достатньо розмістити анотацію `@WebService` в класі технології Java. Всі загальнодоступні методи класу автоматично публікуються у вигляді операцій веб-служби, і всі їх

аргументи прив'язуються до типів даних схеми XML за допомогою JAXB 2.0.

REST (*Representational State Transfer*) – відносно новий спосіб спілкування з веб-службами. Веб-сервіси, побудовані за моделлю REST (інакше звані *RESTful*) – це колекції ідентифікованих веб-ресурсів. Кожен документ і кожен процес моделюється як веб-ресурс з унікальним URI, оперувати яким можна за допомогою передачі команд в HTTP-заголовках. Тут не використовуються ні SOAP, ні WSDL, ні WS-\* стандарти, тому обмін інформацією не обмежується форматом XML - окрім нього можна використати JSON, HTML і т.д. У більшості випадків, браузер служить в ролі клієнта. Слід зауважити, що внаслідок своєї простоти і зручності, створення *RESTful веб-служб* стає популярною альтернативою використанню моделі SOAP. REST – підходяща технологія для застосунків, що не вимагають безпечної передачі даних, і для яких протокол HTTP є кращим. Далі розглядаються саме *RESTful-служби*, зважаючи на їх популярність і широке розповсюдження.

Базовим протоколом REST є HTTP, тому розробнику надані лише чотири операції: **GET**, **PUT**, **POST** і **DELETE**. Використання кожної декларується документом RFC 2616, відповідно до якого:

- **POST** - використовується для створення ресурсів на сервері;
- **GET** - служить для отримання ресурсів і витягання даних;
- **PUT** - змінює стан ресурсу;
- **DELETE** - служить для видалення ресурсу з сервера.

Технологія JAX-RS (JSR 311) надає розробникам API для створення *RESTful служб* на мові Java. Будучи невід'ємною частиною платформи Java EE 6, JAX-RS повністю підтримує принципи REST. JAX-RS API використовує анотації – вони значно полегшують розробку служб. Анотації, поряд з класами та інтерфейсами, наданими в API, дозволяють створювати веб-служби на основі простих POJO. Ну і оскільки анотації підтримуються в Java з версії 5, слід заручитися підтримкою відповідної платформи. Популярною реалізацією JAX-RS від Sun / Oracle є проект *Jersey*. Якщо ви використовуєте сервер застосунків GlassFish – будьте впевнені, *Jersey* присутній в ньому як один з компонентів (при використанні версій нижче Glassfish 3, ви можете встановити його за допомогою *Update Tool*).

## 16.2 Асинхронні веб-служби.

Оскільки виклики веб-служб проводяться по мережі, час їх виконання передбачити неможливо. Продуктивність багатьох клієнтів, особливо інтерактивних, таких як настільні застосунки на базі JFC / Swing, значно *знижується* через необхідність *очікування відповіді* сервера. Щоб уникнути зниження продуктивності в JAX-WS 2.0 передбачено новий асинхронний інтерфейс API клієнта. Цей інтерфейс API дозволяє прикладним програмістам обійтися без самостійного створення *потоків*.



Замість цього управління віддаленими викликами з тривалим часом виконання покладається на JAX-WS.

*Асинхронні методи* можна використовувати в поєднанні з будь-якими інтерфейсами, створюваними WSDL, а також з більш динамічним інтерфейсом *API Dispatch*. Для зручності програміста при імпорті документа WSDL можна запросити створення асинхронних методів для будь-яких операцій, визначених веб-службою.

***Існують дві моделі використання:***

У моделі *опитування* спочатку виконується виклик. Потім, у міру готовності, запитуються результати.

У моделі *зворотного виклику* спочатку реєструється обробник. По мірі прибуття відповідей видається оповіщення.

Слід зазначити, що підтримка асинхронних викликів реалізована повністю на стороні клієнта, таким чином, будь-яких змін цільової веб-служби не потрібний.

У середовище IDE входять засоби для роботи з JAX-WS. Для створення артефактів JAX-WS можна використовувати шаблони у майстрі створення файлів. *Асинхронні веб-служби* можна створювати за допомогою редактора налаштування веб-служб. У функціональні можливості доповнення коду входять *анотації*, придатні для веб-служб.

### 16.3 Архітектура SOA.

Базова архітектура SOA складається з *провайдера сервісів, сервісу і (необов'язкового) каталогу сервісів*. Для обміну інформацією використовується механізм обміну повідомленнями типу "*застосунок до застосунку*".

Подібність між цією моделлю і чистими Web-сервісами цілком очевидна, оскільки в обох випадках застосовується WSDL-документ, який є *контрактом* з активізації, що зберігаються в каталозі сервісів, з якого цей сервіс може бути запитаний і витягнутий за допомогою механізму UDDI. Web-сервіси в дійсності є реалізацією архітектури SOA на найбільш базовому рівні.

У цій моделі базовий сценарій такий. Спочатку провайдер сервісу створює *сервіс*, приймає рішення відкрити цей сервіс і *публікує* його. Публікація виконується шляхом відправки інформації про сервіс в *каталог сервісів*. З іншого боку, ініціатор запитів сервісу (*service requester*), потребуючи в певному сервісі, переглядає каталог сервісів в пошуках того з них, який задовольняє необхідному критерію. Після виявлення такого сервісу і використання доступної в каталозі сервісів інформації ініціатор запитів сервісу може безпосередньо звернутися до *провайдера сервісів* належним способом для задоволення бізнес-потреби.

***Web-сервіси є наріжним каменем архітектури SOA з наступних причин:***

- Вони змушують застосовувати *стандарти* і тим самим сприяють *сумісності* і *переносимості*.
- Вони *не залежать* від платформи і мови програмування.
- Вони *підтримуються* повсюдно, що істотно полегшує впровадження SOA.
- Вони орієнтовані на *повідомлення*.
- Вони забезпечують більш швидко підтримку інструментальними засобами, що прискорює реалізацію SOA.

### **Життєвий цикл SOA:**

#### *Етап моделювання*

Етап моделювання включає в себе аналіз бізнес-діяльності та збір вимог, за якими є моделювання та оптимізація бізнес-процесу. Модель допомагає сформувати загальне розуміння процесу, його цілей і результатів. Вона також гарантує відповідність проекту бізнес-вимогам і забезпечує вихідні критерії для подальшого вимірювання продуктивності.

#### *Етап збірки*

На цьому етапі існуючі активи (наприклад, ERP-системи – системи планування корпоративних ресурсів), фінансові системи, застосунки IBM CICS ® і т.д.), які будуть використовуватися в модельованих процесах, інкапсулюються в сервіси, тоді як відсутні потрібні функціональності реалізуються і тестуються. Коли всі сервіси стають доступними, можна скоординувати їх для реалізації бізнес-процесу.

#### *Етап розгортання*

На етапі розгортання можна налаштувати середовище виконання на необхідний рівень якості сервісу і на задоволення вимог системи безпеки. Середовище може масштабуватися і оптимізуватися для підвищення надійності роботи критичних процесів і для забезпечення гнучкості, що дозволяє динамічно оновлювати її в разі будь-яких змін.

#### *Етап управління*

На цьому етапі виконується керування і моніторинг ряду аспектів, таких як активи сервісів, доступність сервісів, час реакції, контроль версій сервісів. Важливу роль на цьому етапі відіграє моніторинг *ключових показників продуктивності* (*key performance indicators* – KPI) процесу. Він допомагає запобігти або ізолювати і діагностувати проблеми, що виникають у реальному часі, а також забезпечити *зворотний зв'язок* з метою поліпшення продуктивності бізнес-процесу та усунення вузьких місць. Цей зворотний зв'язок передається на етап моделювання (перший етап) з метою удосконалення процесу.

### **Контрольні питання**

1. Принципи SOA.
2. У чому полягає сутність SOA?
3. Архітектура WEB-служби JXA-WS.
4. Якими анотаціями описується Web-служба JXA-WS?

5. Життєвий цикл SOA.
6. Архітектура SOA.
7. У чому полягає різниця між Web-службами JXA-WS та RESTful?
8. Що становить Web-служба RESTful?
9. Який зв'язок між протоколом HTTP та RESTful Web-службою?

## ЛЕКЦІЯ 17. РОЗРОБЛЕННЯ RICH INTERNET APPLICATIONS

### План

- 17.1 Поняття Rich Internet Applications
- 17.2 Переваги RIA
- 17.3 Недоліки RIA
- 17.4 Розробка RIA

### 17.1. Поняття Rich Internet Applications

*Rich Internet Application* (RIA, “Насичений (багатий) [веб-застосунок](#)”) — це [застосунок](#), доступний через [Інтернет](#), і насичений функціональністю традиційною для [прикладних програм](#), який надається або унікальною специфікою [браузера](#), або через [плагін](#), або за допомогою [«пісочниці»](#).

Як правило, насичений інтернет-застосунок:

- передає веб-клієнту необхідну частину користувацького *інтерфейсу*, залишаючи більшу частину (ресурси програми, дані, тощо) на сервері;
- запускається в *браузері* та не потребує додаткового встановлення ПЗ;
- запускається *локально* в середовищі безпеки — [«пісочниці»](#).

На сьогодні найпоширенішими подібними платформами є [Adobe Flash](#), [Java/JavaFX](#) і [Microsoft Silverlight](#) із рівнем проникнення 96%, 76% і 66% відповідно (за станом на серпень 2011).

Термін «RIA» вперше використала компанія [Macromedia](#) в офіційному повідомленні в березні [2002](#) року. Проте ця концепція існувала кількома роками раніше з такими назвами:

- *Remote Scripting*, [Microsoft](#), близько [1998](#)
- *X Internet*, Forrester Research, жовтень [2000](#)
- *Rich (web) client*
- *Rich web application*

Робота традиційних [веб-застосунків](#) сконцентрована довкола [клієнт-серверної архітектури](#) з [тонким клієнтом](#) ([комп'ютер](#) або [програма-клієнт](#) в мережах з [клієнт-серверною](#) або термінальною архітектурою, який переносить всі або більшу частину завдань з обробки інформації на сервер. Прикладом тонкого клієнта може бути комп'ютер з [браузером](#), який використовується для роботи з веб-застосунками). Такий клієнт переносить усі задачі з обробки інформації на сервер, а сам використовується лише для відображення статичного контенту (тут — [HTML](#)). Основним недоліком цього підходу є те, що вся взаємодія із застосунком має оброблятися сервером, що потребує постійного відсилення даних на сервер, очікування відповіді сервера та завантаження сторінки назад до браузера. За використання технології запуску застосунків *на боці клієнта*, RIA може обійти цей повільний цикл

синхронізації за рахунок більшої взаємодії із користувачем. Ця відмінність приблизно аналогічна такій між архітектурою з «тонким клієнтом» ([англ. Thin client](#)) та архітектурою з «товстим клієнтом» ([англ. Fat client](#)), чи між [терміналом](#) і [мейнфреймом](#).

Поступовий розвиток стандартів мережі Інтернет призвів до можливості реалізувати подібні технології на практиці, хоча й складно провести чітку межу між тим, які саме технології включають у собі RIA, а які ні. Проте всі RIA мають одну схожу особливість: вони включають у собі певну проміжну частину коду застосунку, що знаходиться між користувачем і сервером, яку як правило називають «*рушієм клієнта*». Цей рушій завантажується із самого початку та далі може довантажуватися в ході роботи застосунку. Рушій клієнта відіграє роль надбудови браузера та як правило відповідає за [рендеринг](#) користувацького інтерфейсу та взаємодію із сервером.

Те, що може виконати RIA, може обмежуватися можливостями користувацької системи. Проте загалом, інтерфейс користувача створювався з метою виконувати функції, які в сподіваннях розробників повинні були покращити користувацький інтерфейс і прискорити обробку користувацьких запитів, порівняно до можливостей стандартного web-браузера. Також, просте додавання *рушія клієнта* не забороняє застосунку відходити від нормальної синхронної моделі взаємодії браузера та сервера, більшість рушіїв RIA дозволяють виконувати додаткові асинхронні запити до сервера.

## 17.2. Переваги RIA

Не зважаючи на те, що розробка web-застосунків для браузера має обмеження та складніша порівняно до розробки стандартних застосунків, зусилля звичайно виправдані, оскільки:

- Не потрібно *встановлювати* застосунок; поширення застосунку — швидкий і автоматизований процес;
- *Оновлення* версій відбувається автоматично;
- Користувачі можуть використовувати застосунок на *будь-якому комп'ютері*, який має з'єднання з Інтернет; при цьому неважливо, яка [операційна система](#) на ньому встановлена;
- Під час роботи web-застосунку комп'ютер користувача набагато менше наражається на *вірусні небезпеки*, ніж під час запуску виконуваних бінарних файлів.

Оскільки RIA використовують рушій клієнта, щоб взаємодіяти із користувачем, вони:

*Багатші.* RIA пропонують користувацький інтерфейс, не обмежений лише використанням мови HTML, застосовної в стандартних *web-застосунках*. Розширена функціональність дозволяє використовувати такі можливості користувацького інтерфейса, як *drag-and-drop* або повзунець для змінювання даних, а також обчислення, які не відсилаються назад на

сервер, а виконуються безпосередньо на машині користувача (наприклад, іпотечний калькулятор).

*Інтерактивніші.* Інтерфейси RIA інтерактивніші, ніж стандартні інтерфейси веб-браузерів, які вимагають постійної взаємодії з віддаленим сервером.

Найскладніші RIA пропонують зовнішній вигляд і функціональність, близькі до настільних застосунків. Використання рушія клієнта дозволяє досягти й інших переваг продуктивності:

*Збалансованість клієнт-сервера.* Використання обчислювальних ресурсів клієнта і сервера краще збалансовано. Тому сервер не мусить бути «робочою конячкою», як у традиційних *web-застосунках*. Це вивільняє обчислювальні ресурси сервера, дозволяє обробляти більшу кількість сесій одночасно коштом того ж самого апаратного забезпечення.

*Асинхронна комунікація.* Рушій клієнта може взаємодіяти із сервером, не дочекавшись, поки користувач виконає дію в застосунку, натиснувши кнопку чи посилання. Це дозволяє користувачу переглядати сторінку та взаємодіяти з нею асинхронно за допомогою комунікації між рушієм і сервером. Ця можливість дозволяє розробникам RIA передавати дані між клієнтом і сервером не очікуючи на користувача. В *Google Maps* ця техніка використовується для того, щоби підвантажувати прилеглі сегменти мапи, перш ніж користувач пересуне її, щоби їх переглянути.

### 17.3. Недоліки RIA

**Основними недоліками й обмеженнями RIA є:**

*«Пісочниця».* Оскільки RIA завантажуються в локальному середовищі безпеки — «пісочниці» — вони мають обмежений доступ до системних ресурсів. Якщо права на доступ до ресурсів порушено, RIA можуть працювати некоректно.

*Підключення скриптів.* Як правило, для роботи RIA потрібна JavaScript або інша скриптова мова. Якщо користувач відключив активні сценарії у своєму браузері, RIA може не функціонувати належним чином або взагалі не працювати.

*Швидкість обробки клієнтом.* Щоби забезпечити платформну незалежність, деякі RIA використовують скриптову мову на боці клієнта, подібну до JavaScript, із частковою втратою продуктивності (серйозна проблема для мобільних пристроїв). Проте ця проблема не виникає за використання вбудованої мови, скомпільованої на стороні клієнта, такого як Java, де продуктивність порівнянна з використанням традиційних вбудованих мов, або з [Flash](#), або з [Silverlight](#), в яких програмний код запускається безпосередньо в плагіні Flash Player або Silverlight відповідно.

*Час завантаження скрипта.* Навіть якщо немає необхідності в установленні скрипта, рушій клієнта RIA повинен бути переданий клієнту сервером. Оскільки більшість скриптів зберігаються в кеші, він повинен бути переданий хоча б один раз. Залежно від розміру й типу передачі,

завантаження скрипта може зайняти досить багато часу. Розробники RIA можуть зменшити наслідки цієї затримки шляхом *стиснення* скриптів, а також за рахунок *розбивання* передачі застосунка на декілька сторінок.

## 17.4 Розробка RIA

Поява *технології* RIA супроводжувалося значними складностями в розробці веб-застосунків. Традиційні веб-застосунки, створені на основі стандартного HTML, що має порівняно просту архітектуру й досить обмежений набір функцій, були відносно прості в розробці й управлінні. Особи й організації, що впроваджують веб-застосунки на основі технології RIA, часто зіштовхуються з додатковими складностями в розробці, тестуванні, вимірюваннях і підтримці.

Застосування технології RIA ставить нові задачі з управління послугами SLM (*service level management*), не всі з яких вирішені на сьогоднішній день. Питання відносно SLM не завжди враховуються розроблювачами застосунків і майже не сприймаються користувачами. Однак вони життєво важливі для успішного впровадження застосунка в мережі Інтернет. Основними аспектами, що ускладнюють процес розробки RIA, є:

*Більша технологічна складність робить розробку важчою.* Можливість передавати код застосунка безпосередньо клієнтам дає більшу творчу свободу розроблювачам і дизайнерам. Але це, у свою чергу, ускладнює розробку застосунка, збільшує ймовірність помилок при впровадженні й утрудняє тестування програмного забезпечення. Ці ускладнення сповільнюють процес розробки незалежно від специфіки методології й процесу розробки. Деякі із цих проблем можуть бути скорочені за рахунок використання *каркаса програмної системи під веб* (*web application framework*) для стандартизації розробки RIA. Проте, зростаюча складність програмних рішень може ускладнити й подовжити процес тестування при збільшенні числа *тестованих варіантів використання* (*use cases*). Неповне тестування знижує якість і надійність застосунка в ході його використання.

*Архітектура RIA ламає парадигму веб-сторінки.* Традиційні веб-застосунки представляють із себе набір веб-сторінок, кожна з яких вимагає окремого звантажування, ініційованого запитом HTTP **GET**. Ця модель була описана як парадигма веб-сторінки. RIA ламає цю парадигму, вносячи додатковий сервер асинхронної комунікації для підтримки більше інтерактивного інтерфейсу. Повинні бути розроблені нові технології вимірювання для RIA, що надають інформацію про кількість витраченого часу. При відсутності подібних стандартних засобів розроблювачі RIA повинні додати у свої застосунки засобу вимірювання даних, необхідні для SLM.

*Асинхронна комунікація ускладнює виявлення проблем продуктивності.* Парадоксально, але заходи, прийняті для зниження часу



відгуку застосунка утрудняють саме його визначення, вимірювання й керування. Деякі RIA не роблять ніяких подальших HTTP **GET**-запитів із браузера після одержання першої сторінки, використовуючи асинхронні запити за допомогою рушії клієнта для наступних завантажень. *Клієнт RIA* може бути запрограмований таким чином, щоб постійно завантажувати новий контент і обновляти дисплей, або (у застосунках, що використовують підхід *Comet*) рушій на стороні сервера може постійно передавати новий контент браузеру через *постійно відкрите з'єднання*. У цьому випадку концепція «завантаження сторінки» більше не застосовна. Усе це привносить певні труднощі у вимірювання й розділення часу відгуку застосунка, які є фундаментальними вимогами для ізоляції проблем і SLM. Інструменти, створені для вимірювання традиційних веб-застосунків, залежно від специфіки й інструментарію застосунка можуть розглядати кожен веб-сторінку, запитану по HTTP, окремо або як набір не пов'язаних між собою показників. Однак, жоден із цих підходів не показує, що в дійсності відбувається на рівні застосунка.

*Рушій клієнта ускладнює вимірювання часу відгуку застосунка.* Для традиційних веб-застосунків вимірювальне програмне забезпечення може розташовуватися на клієнтській машині й на машині, близькій до сервера, таким чином, воно може спостерігати за потоком мережного трафіка на TCP- і HTTP-рівнях. Оскільки це синхронізовані й передбачувані протоколи, пакет зі *сніфером* може читати й інтерпретувати дані пакетного рівня й виводити висновок про час відгуку за допомогою засобів відстеження повідомлень HTTP і часу підтвердження пакетів TCP на нижньому рівні. Але архітектура RIA зменшує можливості підходу з використанням *пакетного сніфінга*, оскільки рушій користувача розбиває взаємодію між клієнтом і сервером на два окремих цикли, що працюють асинхронно — *цикл переднього плану (користувач-рушій)* і *цикл заднього плану (рушій-сервер)*. Обидва цих цикли мають важливе значення, оскільки їхній загальний взаємозв'язок визначає поведінку застосунка. Але це відношення залежить тільки від побудови самого застосунка, що в більшості випадків не може бути спрогнозовано вимірювальними інструментами, особливо першим, котрий може спостерігати тільки один із двох циклів. Тому найповніше вимірювання RIA може бути отримано тільки з використанням інструментів, які є на боці клієнта й спостерігача в обох циклах.

### Контрольні питання

1. Що таке Rich Internet Application? Які характеристики його роботи?
2. Який принцип роботи тонкого клієнта?
3. Які є переваги RIA?
4. Що таке асинхронна комунікація?
5. Які є недоліки RIA?
6. Які аспекти ускладнюють розробку RIA?



7. В чому полягає ламання веб-сторінки архітектурою RIA?

## ЛЕКЦІЯ 18. ОГЛЯД СУЧАСНИХ ТЕХНОЛОГІЙ РОЗРОБЛЕННЯ БАГАТОЛАНКОВИХ ЗАСТОСУНКІВ

### План

18.1. Основи Hibernate. Архітектура Hibernate.

18.2. Основи Spring Framework.

### 18.1 Основи Hibernate. Архітектура Hibernate

*Hibernate* — засіб відображення між об'єктами та реляційними структурами (*object-relational mapping*, ORM) для платформи Java. Hibernate є вільним програмним забезпеченням, яке поширюється на умовах GNU Lesser General Public License. Hibernate надає легкий для використання каркас (*фреймворк*) для відображення між об'єктно-орієнтованою моделлю даних і традиційною реляційною базою даних.

Метою Hibernate є звільнення розробника від значних типових завдань із програмування взаємодії з [базою даних](#). Розробник може використовувати Hibernate як при розробці з нуля, так і для вже існуючої бази даних.

Hibernate піклується про зв'язок [класів](#) з таблицями бази даних (і [типів даних](#) мови програмування із типами даних [SQL](#)), і надає засоби автоматичної побудови SQL запитів й зчитування/запису даних, і може значно зменшити час розробки, який зазвичай витрачається на ручне написання типового SQL і [JDBC](#) коду. Hibernate генерує SQL виклики і звільняє розробника від ручної обробки результуючого набору даних, конвертації [об'єктів](#) і забезпечення сумісності із різними базами даних.

Hibernate забезпечує прозору підтримку збереження даних, тобто їхньої *персистентності* ([англ.](#) *persistence*) для [«POJO»-об'єктів](#), себто для звичайних Java-об'єктів; єдина сувора вимога до класу, що зберігається — типовий [конструктор](#) (для коректної поведінки у деяких застосунках потрібно приділити особливу увагу методам `equals()` і `hashCode()`).

Архітектура Hibernate є *шаруватою*, щоб утримати користувача від необхідності знати основні API. Hibernate використовує бази даних та конфігурації даних, щоб забезпечити збереження послуг (і постійних об'єктів) в застосунок.

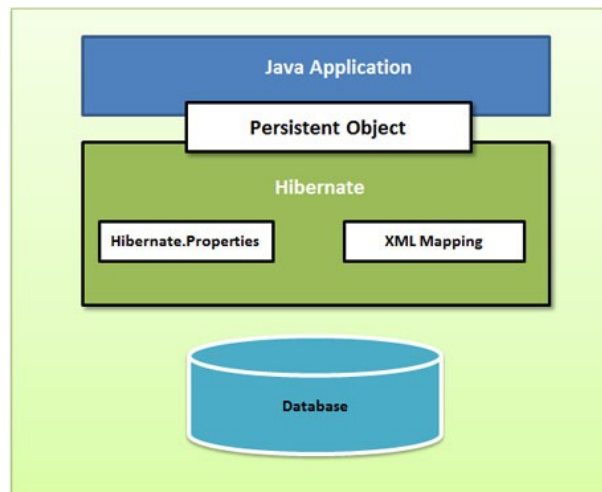


Рисунок 18.1 – Архітектура Hibernate

Нижче наводиться докладний огляд архітектури застосунків Hibernate з кількома важливими основними класами.

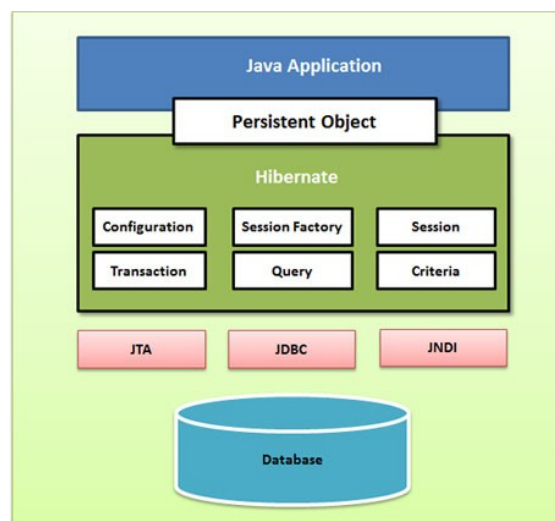


Рисунок 18.2 – Архітектура застосунків

Визначення об'єктів, що зображені на діаграмі (рисунок 18.2):

*Session Factory* (`org.hibernate.SessionFactory`) – незмінний кеш компільованих відображень для єдиної бази даних. Фабрика для Сесії і клієнта `ConnectionProvider`, `SessionFactory` може містити кеш даних, який повторно використовується між транзакціями в рівні процесу, або кластера.

*Session* (`org.hibernate.Session`)

Однопоточний, недовгий об'єкт, що представляє взаємодію між застосунком і постійною пам'яттю. Сесія тримає обов'язковий першорівневий кеш постійних об'єктів, які використовуються для навігації графічних об'єктів або знаходження об'єктів за ідентифікатором.

*Постійні об'єкти і колекції*

Нетривалі, *однопоточні* об'єкти, що містять постійну встановлену та ділову функцію. Це може бути звичайний *Java Beans / POJOs*. Вони зв'язуються з однією Сесією. Після того як Сесія буде закрита, вони будуть окремими і вільними для використання в будь-якому прикладному рівні.

#### *Швидкоплинні і окремі об'єкти і колекції*

Зразки *постійних* класів, які в даний момент не зв'язуються з Сесією. Вони можуть бути підтверджені застосунком, або вони, можливо, були підтверджені закритою Сесією.

*Транзакція* (`org.hibernate.Transaction`) – однопоточний, тимчасовий об'єкт, використовуваний застосунком, щоб конкретизувати атомні одиниці роботи. Це абстрагує застосунок від основного JDBC, JTA або CORBA транзакціями. Сесія може охоплювати кілька транзакцій в деяких випадках.

#### *ConnectionProvider*

(`org.hibernate.connection.ConnectionProvider`)

Фабрика для підключення JDBC. Це абстрагує застосунок, що лежить в основі Datasource або DriverManager. Це може бути розширено і / або реалізовано розробником.

*Transaction Factory* (`org.hibernate.TransactionFactory`) – фабрика для Операційних зразків. Це не є застосунком, але це може бути розширено і / або реалізовано розробником.

#### *Інтерфейси розширення*

Hibernate пропонує діапазон розширення інтерфейсу за допомогою якого можна здійснювати налаштування поведінки рівня сталості.

Компоненти Hibernate:

- *Hibernate Core* — ядро Hibernate для Java, власний API та метадані відображенні у форматі XML;
- *Hibernate Annotations* — відображення за допомогою анотацій [JDK 5.0](#), як стандартних для [JPA](#), так і власних розширень;
- *Hibernate EntityManager* — реалізація *Java Persistence API* для Java SE і Java EE;
- *Hibernate Shards* — горизонтальна структура поділу даних;
- *Hibernate Validator* — цілісність даних аотацій і перевірка API;
- *Hibernate Search* — Hibernate-інтеграції з *Lucene* для індексації і запитів даних;
- *Hibernate Tools* — засоби розробки для [Eclipse](#) та [Ant](#);
- *NHibernate* — *NHibernate-service* для [.NET](#) платформи;
- *JBoss Seam* — каркас для [JSF](#), [Ajax](#), та [EJB 3.0/Java EE 5.0](#) застосунків;

*Mapping* (співставлення, буквально – картування) Java класів з таблицями бази даних здійснюється за допомогою конфігураційних [XML](#) файлів або *Java-анотацій*. При використанні файлу XML, Hibernate може генерувати скелет вихідного коду для класів *тривалого зберігання* (*persistent*). У цьому немає необхідності, якщо

використовується *анотація*. Hibernate може використовувати файл XML або анотації для підтримки *схеми* бази даних.

Забезпечуються можливості з організації відношення між класами «*один-до-багатьох*» і «*багато-до-багатьох*». На застосунок до управління зв'язками між об'єктами, Hibernate також може керувати рефлексивними асоціаціями, де об'єкт має зв'язок «*один-до-багатьох*» з іншими примірниками свого власного типу даних.

Hibernate підтримує відображення користувацьких типів значень. Це робить можливим такі сценарії:

- Перевизначення типового типу SQL, який Hibernate вибирає при відображенні стовпчика властивості.
- Картування перераховуваного типу Java до колонок БД, так ніби вони є звичайними властивостями.
- Картування однієї властивості в декілька колонок.

Hibernate забезпечує прозоре збереження POJO (*Plain Old Java Objects* – *простих старих об'єктів Java*). Єдина сувора вимога для персистентного класу — конструктор без аргументів, не обов'язково публічний. Для правильної поведінки деяких програм також потрібна особлива увага до методів `equals()` і `hashCode()`.

Колекції об'єктів даних, як правило, зберігаються у вигляді колекцій Java-об'єктів, таких як набір (**Set**) і список (**List**). Підтримуються узагальнені класи (**Generics**), введені в Java 5. Hibernate може бути налаштований на «*ледачі*» (*відкладені*) завантаження колекцій. Відкладені завантаження є типовим варіантом, починаючи з Hibernate 3.

Зв'язані об'єкти можуть бути налаштовані на каскадні операції. Наприклад, батьківський клас, Album (музичний альбом), може бути налаштований на каскадне збереження і/або видалення свого нащадка Track. Це може скоротити час розробки і забезпечити цілісність. Функція перевірки зміни даних (*dirty checking*) дозволяє уникнути непотрібного запису дій в базу даних, виконуючи SQL оновлення тільки при зміні полів персистентних об'єктів.

Hibernate може використовуватись як у самостійних програмах [Java](#), так і в програмах [Java EE](#), що виконуються на сервері (наприклад, *сервлети* чи [EJB session beans](#)). Також він може включатись як додаткова можливість до інших мов програмування. Наприклад, [Adobe](#) інтегрував Hibernate у дев'яту версію *ColdFusion* (що запускається на серверах з підтримкою застосунків [J2EE](#)) з рівнем абстракції нових функцій і синтаксису, доданих до [CFML](#).

## 18.2 Основи Spring Framework

The *Spring Framework* (або коротко: Spring) – універсальний фреймворк з відкритим вихідним кодом для [Java](#)-платформи. Перша версія була написана [Родом Джонсоном](#).

Незважаючи на те, що *Spring Framework* не забезпечував якусь конкретну модель програмування, він став широко поширеним в Java-співтоваристві головним чином як альтернатива і заміна моделі *Enterprise Java Beans*. *Spring Framework* надає більшу свободу Java розробникам в проектуванні, крім того, він надає добре документовані і легкі у використанні засоби вирішення проблем, що виникають при створенні застосунків корпоративного масштабу.

Між тим, особливості ядра *Spring Framework* застосовні в будь-якому Java-застосунку, і існує безліч розширень і удосконалень для побудови [веб-застосунків](#) на [Java Enterprise платформі](#). З цих причин Spring придбав велику популярність і визнається розробниками як стратегічно важливий фреймворк.

Spring Framework забезпечує вирішення багатьох завдань, з якими стикаються Java розробники та організації, які хочуть створити інформаційну систему, засновану на платформі [Java](#). Через широку функціональність важко визначити найбільш значущі структурні елементи, з яких він складається. Spring Framework не цілком пов'язаний з платформою [Java Enterprise](#), незважаючи на його масштабну інтеграцію з нею, що є важливою причиною його популярності.

*Spring Framework*, ймовірно, найбільш відомий як джерело розширень (*features*), потрібних для ефективної розробки складних бізнес-застосунків поза великоваговими програмними моделями, які історично були домінуючими в промисловості. Ще одна його перевага в тому, що він ввів раніше невикористовувані функціональні можливості в сьогоденні панівні методи розробки, навіть поза платформу Java.

Цей фреймворк пропонує послідовну модель і робить її придатною до більшості типів застосунків, які вже створені на основі платформи Java. Вважається, що *Spring Framework* реалізує модель розробки, засновану на кращих стандартах індустрії, і робить її доступною в багатьох областях Java.

*Spring Framework* може бути розглянутий як колекція менших фреймворків або фреймворків під фреймворком. Більшість цих фреймворків може працювати незалежно один від одного, проте, вони забезпечують більшу функціональність при спільному їх використанні. Ці фреймворки поділяються на структурні елементи типових комплексних застосунків:

- Контейнер *Inversion of Control*: конфігурування компонентів застосунків і управління життєвим циклом Java об'єктів.
- Фреймворк [аспектно-орієнтованого програмування](#): працює з функціональністю, яка не може бути реалізована можливостями [об'єктно-орієнтованого програмування](#) на Java без втрат.
- Фреймворк доступу до даних: працює з [системами керування базами даних](#) на Java платформі використовуючи [JDBC](#) і *Object-*



*relational mapping*-засоби забезпечуючи вирішення завдань, які повторюються у великому числі *Java-based environments*.

- *Фреймворк управління транзакціями*: координація різних [API](#) управління транзакціями та інструментарій налаштовуваного управління транзакціями для об'єктів Java.
- *Фреймворк Model-view-controller*: каркас, заснований на [HTTP](#) і [сервлетах](#); надає безліч можливостей для розширення та налаштування (*customization*).
- *Фреймворк віддаленого доступу*: конфігурується передача Java-об'єктів через мережу в стилі [RPC](#), підтримує [RMI](#), [CORBA](#), [HTTP-based протоколи](#), включаючи *web-services* ([SOAP](#)).
- *Фреймворк аутентифікації і авторизації*: конфігурований інструментарій процесів аутентифікації й авторизації, підтримує багато популярних індустріальних стандартів протоколів, інструментів, практик, через дочірній проект *Spring Security* (раніше відомий як *Acegi*).
- *Фреймворк віддаленого управління*: керування Java об'єктами для локальної або віддаленої конфігурації за допомогою [JMX](#).
- *Фреймворк роботи з повідомленнями*: конфігурована реєстрація об'єктів-слухачів повідомлень для прозорої обробки повідомлень з [черги повідомлень](#) за допомогою [JMS](#), поліпшена відправка повідомлень за стандартом JMS API.
- *Тестування*: каркас, що підтримує класи для написання модульних та інтеграційних тестів.

Центральною частиною *Spring Framework* є контейнер *Inversion of Control*, який надає засоби конфігурування та управління об'єктами Java за допомогою *рефлексії*. Контейнер відповідає за управління життєвим циклом об'єкта: створення об'єктів, виклик методів ініціалізації та конфігурування об'єктів шляхом зв'язування їх між собою.

Об'єкти, які створюють контейнер, називаються *керованими об'єктами*, або *Beans*. Зазвичай конфігурування контейнера здійснюється шляхом завантаження XML файлів, що містять визначення *Bean*'ів і надають інформацію необхідну для створення *bean*'ів.

Об'єкти можуть бути отримані за допомогою або *пошуку залежності*, або *впровадження залежності*. *Пошук залежності* – шаблон проектування, коли об'єкт-викликач запитує в об'єкта-контейнера екземпляр об'єкта з певним ім'ям або певного типу. *Впровадження залежності* – шаблон проектування, коли контейнер передає екземпляри об'єктів по їх імені іншим об'єктам або за допомогою *конструктора*, або властивості, або *фабричного методу*.

*Spring MVC* – фреймворк, орієнтований на запити та надає деякі можливості для розробника:

- Ясний і прозорий *поділ* між шарами в MVC і запитах.
- Стратегія інтерфейсів – кожен інтерфейс робить тільки свою частину роботи.

- Інтерфейс завжди може бути замінений *альтернативною реалізацією*.
- Інтерфейси тісно пов'язані з *Servlet API*.
- Високий рівень *абстракції* для веб-застосунків.
- У веб-застосунках можна використовувати різні частини *Spring Framework*, а не тільки Spring MVC.

Spring надає свій шар доступу до баз даних і підтримує всі популярні бази даних: [JDBC](#), [iBatis/MyBatis](#), [Hibernate](#), [JDO](#), [JPA](#), [Oracle TopLink](#), [Apache OJB](#), [Apache Cayenne](#) і т.д.

Для всіх цих фреймворків, Spring надає такі особливості:

- *Управління ресурсами* – автоматичне отримання та звільнення ресурсів бази даних;
- *Обробка виключень* – передача виключень при доступі до даних в виключення Spring-а;
- *Транзакційні* – прозорі транзакції в операціях з даними;
- *Розпаковування ресурсів* – отримання об'єктів бази даних із пулу з'єднань;
- *Абстракція для обробки [BLOB](#) і [CLOB](#)*.

Управління транзакціями в Spring рамках приносить абстракції механізм для платформи Java. Основні можливості:

- робота з *локальними* та *глобальними транзакціями*;
- робота із *вкладеними транзакціями*;
- робота з *точками збереження* в транзакціях.

*Spring Integration* – фреймворк для [J2EE](#) який надає функції необхідні для надсилання повідомлень або для побудови *подієво-орієнтованої архітектури*.

***Пристрої, на яких доцільно інтегровувати дані технології:***

- маршрутизатори;
- трансформери;
- адаптери для інтеграції з іншими технологіями і системами ([HTTP](#), [AMQP](#), [JMS](#), [XMPP](#), [SMTP](#), [IMAP](#), [FTP](#) (FTPS/SFTP), файлові системи і т. д.);
- фільтри;
- активатори сервісів;
- засоби аудиту та управління.

### **Контрольні питання**

1. Що таке Hibernate?
2. З яких компонентів складається архітектура Hibernate?
3. З яких компонентів складається архітектура застосунків Hibernate?
4. З яких компонентів складається Hibernate?
5. Що таке Spring Framework? Кі можливості надає розробнику [Spring MVC](#)?
6. Які особливості надає [Spring](#) для фреймворків?



## ЛІТЕРАТУРА

**Основна**

1. Вандер Вер Эмили JavaScript для "чайников": Уч. пос./Под ред. В.М.Неумоина .-3-е изд.-М.:Изд. дом Вильямс,2001. – 304 с.-(Ил.) .-5-8459-0134-0 Шифр: 681.3 Авторський знак: В17
2. Янг Майкл Дж.Visual C++6. Полное руководство. Т.1.- К.:Ирина,1999 .-544 Шифр: 681.3.06 Авторський знак: Я60
3. Янг Майкл Дж. Visual C++6.Полное руководство. Т.2 .- К.:Ирина,1999 .-560 Шифр: 681.3.06 Авторський знак: Я60
4. Глушаков С.В., Коваль А.В., Черепнин С.А. Программирование на Visual C++ .-Харьков:Фолио,2002 .-726 с.- Учебный курс .-966-03-1776-X Шифр: 681.3(075.8) Авторський знак: Г55
5. Монсон Хейфел Р. Enterprise JavaBeans / Р. Хейфел Монсон ; пер. с англ. – 3-е изд. – СПб. : СимволПлюс, 2002. – 672 с. : ил.
6. Олифер В.Г. Компьютерные сети. Принципы, технологии, протоколы : учебник для вузов / В. Г. Олифер, Н. А. Олифер. – 4-е изд. – СПб. : Питер, 2010. – 944 с. : ил.
7. Перри Б. Java сервлеты и JSP: сборник рецептов // Б. Перри ; пер с англ. – М. : Кудиц-пресс, 2006. – 768 с.
8. Таменбаум Э. Распределенные системы. Принципы и парадигмы / Э. Таненбаум, М. ван Стеен. – СПб. : Питер, 2003. – 877 с. : ил.
9. Эккель Б. Философия Java. Библиотека программиста. / Б. Эккель. – 4-е изд. – СПб. : Питер, 2009. – 640 с. : ил.
10. Calvert K. L. TCP/IP Sockets in Java Practical Guide for Programmers / K. L. Calvert, M. J. Donahoo. – 2nd ed. – Burlington : Morgan Kaufmann, 2007. – 193 p.
11. Heffelfinger D. Java EE 6 with GlassFish 3 Application Server / D. Heffelfinger – Packt Publishing.
12. Linwood J. Beginning Hibernate, Second Edition / J. Linwood, D. Minter ; 2nd ed. – NY. : Apress, 2010. – 401 p.
13. Mak G. Spring Enterprise Recipes: A Problem-Solution Approach / G. Mak, J. Long; 2nd ed. – NY. : Springer, 2009. – 1104 p.
14. Mike Keith, ol. Pro JPA 2. Mastering the Java™ Persistence API / Mike Keith. Merrick Schincari. – New York : Apress, 2009. – 238 p.
15. Парфьонов Ю. Е., Поляков А. О. Робоча програма навчальної дисципліни "КРОСПЛАТФОРМОВІ ТА БАГАТОЛАНКОВІ ТЕХНОЛОГІЇ" Харків. Вид. ХНЕУ, 2012
16. Макарова Н.В., Волков В.Б. Информатика: Учебник для вузов. – СПб.: Питер, 2011. – 576 с.: ил.
17. Іванніков Є.Ю. Послуга повної довірчої конфіденційності для захищеної ОС на базі GNU/LINUX з роширенням RSBAC//Проблеми програмування.-2010.-№3 .-с.513-518

18. Анісімов А.В., Іванніков Є.Ю. Послуга "КО-1. Повторне використання об'єктів" для захищеної ОС на базі GNU/LINUX з розширенням RSBAC//Проблеми програмування.-2010.-№4 .-с.11-20
19. Блинов И. Н. Java. Промышленное программирование : практ. пособ. / И. Н. Блинов, В. С. Романчик. – Мн : УниверсалПресс, 2007. – 768 с.
20. Дейтел Х. М. Технологии программирования на Java 2: Книга 2. Распределенные приложения / Х. М. Дейтел, П. Дж. Дейтел. ; пер. с англ. – М. : ООО "Бином-Пресс", 2003. – 464 с. : ил.
21. Дейтел Х. М. Технологии программирования на Java 2: Книга 3. Корпоративные системы / Х. М. Дейтел, П. Дж. Дейтел, С. И. Самтри ; пер. с англ. – М. : ООО "Бином-Пресс", 2003. – 672 с. : ил.
22. Дэвид М. Гери, JavaServer Faces. Библиотека профессионала. JavaServer Faces. CORE / Дэвид М. Гери, Кей С. Хорстманн. – 3-е изд. – М. : Издательский дом "Вильямс", 2011. – 544 с.
23. Брюс Еккель, Thinking in Java., пер. Є. Матвеев. Бібліотека програміста, в-во "Пітер", 2009 - 640 с.

### ***Ресурси мережі Internet***

1. Всесвітня щорічна конференція з Java технологій [Електронний ресурс]. – Режим доступу : <http://www.oracle.com/javaone/index.html>
2. Документація Java™ Platform, Standard Edition 7 API Specification [Електронний ресурс]. – Режим доступу : <http://docs.oracle.com/javase/7/docs/api/>
3. Документація Java™ Platform, Enterprise Edition 6 API Specification [Електронний ресурс]. – Режим доступу : <http://docs.oracle.com/javase/6/api/>
4. Документація Java™ 2 Platform Enterprise Edition, 5.0. API Specifications [Електронний ресурс]. – Режим доступу : <http://docs.oracle.com/javase/5/api/>
5. Офіційна документація JavaFX [Електронний ресурс]. – Режим доступу : <http://docs.oracle.com/javafx/>
6. Статті всесвітніх експертів з Java-технологій [Електронний ресурс]. – Режим доступу : <http://www.javaworld.com>
7. Статті експертів компанії IBM з Java-технологій [Електронний ресурс]. – Режим доступу : <http://www.ibm.com/developerworks/ru/java/>
8. Універсальний фреймворк з відкритим вихідним кодом для Java-платформи. The Spring Framework [Електронний ресурс]. – Режим доступу : <http://www.springsource.org/>

9. JSR 318: Enterprise JavaBeans™, Version 3.1. EJB Core Contracts and Requirements. Sun Microsystems. – November 5, 2009. – 626 p. [Electronic resource]. – Access mode : <http://jcp.org/en/jsr/detail?id=318>
10. JSR 315: Java™ Servlet Specification, Version 3.0. Oracle. – February 6, 2011. – 230 p. [Electronic resource]. – Access mode : <http://jcp.org/en/jsr/detail?id=315>
11. JSR 245: JavaServer Pages™ Specification, Version 2.2. Maintenance Release 2. Sun Microsystems. – December 10, 2009. – 594 p. [Electronic resource]. – Access mode : <http://jcp.org/en/jsr/detail?id=245>
12. JSR 245: Expression Language Specification, Version 2.2. Maintenance Release. A component of the JavaServer™ Pages Specification. Version 2.2. – December 10, 2009. 594 p. [Electronic resource]. – Access mode : <http://jcp.org/en/jsr/detail?id=245>
13. JSR 314: JavaServer™ Faces Specification, Version 2.1. Maintenance Release 2. Oracle. – November 8, 2010. – 468 p. [Electronic resource]. – Access mode : <http://jcp.org/en/jsr/detail?id=314>
14. JSR 311: JAX-RS: Java™ API for RESTful Web Services, Version 1.1. Final Release. Sun Microsystems. – September 17, 2009. – 51 p. [Electronic resource]. – Access mode : <http://jcp.org/en/jsr/detail?id=311>
15. JSR 224: The Java API for XML-Based Web Services (JAX-WS) 2.2 Rev a. Maintenance Release. Oracle. – May 13, 2011. – 181 p. [Electronic resource]. – Access mode : <http://jcp.org/en/jsr/detail?id=224>
16. Internet- інститут інформаційних технологій [Електронний ресурс]. – Режим доступу: – <http://www.intuit.ru>
17. NetBeans IDE. Учебная карта по Java EE и Java Web. [Електронний ресурс]. – Режим доступу : [http://netbeans.org/kb/trails/java- ee\\_ru.html](http://netbeans.org/kb/trails/java- ee_ru.html)
18. ORM Hibernate [Електронний ресурс]. – Режим доступу: <http://www.hibernate.org/>
19. The Java EE 6 Tutorial [Електронний ресурс ] // Oracle Corporation – July 2011.–Р. 906. – Режим доступу: <http://download.oracle.com/javase/6/tutorial/doc/javaeetutorial6.pdf>