ЛЕКЦІЯ 3. СТРАТЕГІЇ ІНТЕГРАЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

План

3.1. Модульна організація програмних систем

3.1.1. Принципи модульності

Модульність – принцип організації великих систем у вигляді наборів підсистем, модулів або **компонентів**. Цей принцип наказує організовувати складну систему у вигляді набору простіших систем – модулів, що взаємодіють один з одним через чітко визначені **інтерфейси**. При цьому кожне завдання, що вирішується всією системою, розбивається на простіші, вирішувані окремими модулями підзадачі, вирішення яких, будучи скомбіновано певним чином, дає у результаті рішення потрібної задачі. Після цього можна окремо розглядати кожну підзадачу і модуль, відповідальний за її рішення, і окремо – питання **інтеграції отриманого набору модулів в цілісну** систему, здатну вирішувати завдання. Виділення **чітких інтерфейсів** для взаємодії компонентів спрощує інтеграцію, дозволяючи проводити її на основі **чітко визначених інтерфейсів**, без звернення до багаточисельних внутрішніх елементів модулів, що призвело б до зростання складності.

Прикладом розбиття на модулі в .Net Framework є об'єднання класів в простори імен. Другим прикладом є створення власних бібліотек модулів (DLL-бібліотек).

При модульній (компонентній) організації програмної системи важливо дотримуватися принципів модульності:

Принципи модульності:

Розділення інтерфейсу і реалізації. Цей принцип використовується при відділенні зовні видимої структури модуля, опису завдань, які він вирішує, від способів вирішення цих завдань.

Слабка зв'язність (coupling) модулів і сильна спорідненість (cohesion) функцій в одному модулі. Обидва ці принципи використовуються для виділення модулів у великій системі і тісно пов'язані з розділенням відповідальності між модулями. Перший вимагає, аби залежностей між модулями було якомога менше. Модуль, залежний від багатьох модулів в системі, скоріше за все, треба перепроектувати – це означає, що **він вирішує** надто багато задач.

І навпаки, «спорідненість» функцій, що виконуються одним модулем, має бути як можна вищою. Хоча на рівні коду причини цієї «спорідненості» можуть

бути різними – робота з одними і тими ж даними, залежність від роботи один одного, необхідність синхронізації при паралельному виконанні і ін. – ціна їх розділення має бути досить високою. Найбільш важливим є те, що ці функції вирішують тісно зв'язані один з одним завдання.

Принцип повторного використання (reuse). Цей принцип вимагає уникати повторень описів одних і тих же знань – у вигляді структур даних, дій, алгоритмів, одного і того самого коду – в різних частинах системи. Замість цього в добре спроектованій системі виділяється одне джерело, одне місце фіксації для кожного елементу знань і організується його повторне використання у всіх місцях, де потрібно використовувати цей елемент знань. Така організація дозволяє, при необхідності (наприклад, при виправленні помилки або розширенні можливостей), зручним чином модифікувати код і документи системи відповідно до нового вмісту елементів знань, оскільки кожен з них зафіксований рівно в одному місці. Прикладом може служити організація класів у простори імен і динамічні бібліотеки.

3.1.2. Модульна декомпозиція системи

В модульній архітектурі програмних систем важливим є не лише правильно розділити систему на компоненти, але і вибрати раціональну стратегію їх інтеграції.

В структурних стратегіях розробки програмних систем існують дві основні стратегії: **низхідна (зверху-вниз) і висхідна (знизу-вгору).**

Низхідна (зверху-вниз) стратегія – розробка починається з визначення цілей вирішення проблеми, після чого йде послідовна деталізація, що закінчується детальною програмою.

При низхідному проектуванні завдання аналізується з метою визначення можливості розбиття її на ряд підзадач. Потім кожна з отриманих підзадач також аналізується для можливого розбиття на підзадачі. Процес закінчується, коли підзадачу неможливо або недоцільно далі розбивати на підзадачі.

За цією стратегією система має ієрархічну структуру: від головної програми до підпрограм самого нижнього рівня.

Компоненти, які ще не реалізовані, замінюються так званими "заглушками". Інтеграція виконується по мірі готовності компонентів, які замінюють "заглушки".

Висхідна стратегія (знизу-вгору) – розробка починається модулів (компонентів) нижніх рівнів, у той час коли опрацювання загальної схеми не закінчилося. Компоненти верхнього рівня замінюються так званими "драйверами", які імітують загальну структуру системи.

Ця методика є протилежною до методики програмування зверху «вниз», є менш бажаною в порівнянні з низхідним програмуванням, оскільки часто призводить до небажаних результатів, переробок і збільшення часу на розробку.

Обидві ці стратегії широко застосовувалися в ієрархічних системах і технологіях структурного програмування.

3.2. Специфікація інтерфейсу як контракту

Як ми обговорювали в лекції 2, взаємодія компонентів виконується за допомогою чітко визначених інтерфейсів, які ще називають **контрактом** прикладного рівня.

При цьому внутрішня реалізація компонента і його внутрішні дані приховані. Загалом, інтерфейс компонента значно менший, ніж набір всіх операцій і даних, які є в компоненті.

При проектуванні таких компонентів їх інтерфейси повинні відповідати певним вимогам (правилам):

Адекватність інтерфейсу означає, що інтерфейс компонента дає можливість вирішувати саме ті завдання, які потрібні користувачам цього компонента.

Повнота інтерфейсу означає, що інтерфейс дозволяє вирішувати всі значимі завдання в рамках функціональності модуля.

Мінімальність інтерфейсу означає, що операції, що надаються інтерфейсом, вирішують різні по сенсу завдання і жодну з них не можна реалізувати за допомогою всіх інших (або ж така реалізація досить складна і неефективна).

Простота інтерфейсу означає, що інтерфейсні операції досить елементарні і не можуть бути представлені у вигляді композицій деяких простіших операцій на тому ж рівні абстракції, при тому ж розумінні функціональності модуля.

3.3. Стратегії інтеграції компонентів

Крос-платформні технології забезпечують спільну експлуатацію різних апаратних і програмних платформ. Для цього вони повинні забезпечувати *взаємодію компонентів*, розроблених на різних платформах.

Однією з основних вимог, яким повинні задовольняти компоненти, є використання програмного забезпечення і технологій, що узгоджуються із **загальновизнаними стандартами**, що визначають принципи взаємодії компонентів.

2.3.1. Концепції взаємодії компонентів

***Програмні системи, згідно з компонентною ідеологією**, реалізуються у вигляді **слабо-зв'язаних компонентів з чітко-визначними інтерфейсами**.* При цьому в системі повинні бути реалізовані механізми взаємодії (інтеграції) компонентів.

***Під взаємодією компонентів будемо розуміти виклик компонентів і обмін даними між компонентами**.*

Існують дві концепції взаємодії програмних компонентів:

1) обмін повідомленнями між компонентами;

2) ***віддалений виклик процедур*** (**Remote Procedure Calls, RPC**) або методів об'єкту віддаленої компоненти по аналогії з локальним викликом процедури.

Оскільки на теперішній час будь-яка взаємодія між віддаленими компонентами зрештою заснована на мережевих сокетах TCP/IP, первинним є

низькорівневий обмін повідомленнями на основі мережевих сокетів, сервіс яких ніяк не визначає формат повідомлення.

На базі протоколів TCP або HTTP потім можуть бути побудовані прикладні протоколи обміну повідомленнями більш високого рівня абстракції для реалізації складнішого обміну повідомленнями або віддаленого виклику процедур.

Віддалений виклик є моделлю, що походить від мов програмування високого рівня, а не від реалізації інтерфейсу транспортного рівня мережевих протоколів.

Ці моделі взаємодії реалізовані в усіх сучасних операційних системах і надають **низькорівневий** інтерфейс для взаємодії компонентів.

Високорівневий інтерфейс прикладного рівня забезпечують компонентні моделі, реалізовані у вигляді так званого *проміжного програмного забезпечення*. Прикладом такої моделі є платформа .Net Framework. Існують і інші моделі проміжного рівня, про які ми згадували в лекції 2, і які будуть розглянуті далі більш детально.

Обмін повідомленнями

Існує *два методи* передачі повідомлень від однієї віддаленої системи до іншої – *безпосередній обмін повідомленнями* і *використання черг повідомлень*.

У першому випадку передача відбувається безпосередньо, і вона можлива лише в тому випадку, якщо приймаюча сторона готова прийняти повідомлення в цей же момент часу.

У другому випадку використовується посередник – *менеджер черг повідомлень*. Компонент посилає повідомлення в одну з черг менеджера, після чого він може продовжити свою роботу. Надалі одержуюча сторона отримає повідомлення з черги менеджера і приступить до його обробки.

Простою реалізацією безпосереднього обміну повідомленнями є використання транспортного рівня мережі через інтерфейс сокетів. Проте такий спосіб взаємодії зазвичай не застосовується в системах автоматизації підприємства, оскільки в цьому випадку реалізація всіх функцій обміну повідомленнями лягає на розробників системи. При такому підході складно отримати розширювану і надійну розподілену систему, тому для розробки прикладних розподілених систем зазвичай використовуються системи черг повідомлень.

Існує декілька розробок, що реалізовують високорівневі сервіси для обміну повідомленнями між програмними компонентами. До них відносяться, зокрема, Microsoft Message Queuing, IBM MQSeries і Sun Java System Message Queue. Такі системи дають можливість компонентам використовувати наступні базові примітиви по використанню черг:

– додати повідомлення в чергу;

– узяти перше повідомлення з черги, процес блокується до появи в черзі хоч б одного повідомлення;

– перевірити чергу на наявність повідомлень;

- встановити обробник, що викликається при появі повідомлень в черзі.

Менеджер черги повідомлень в таких системах може знаходитися на комп'ютері, відмінному від комп'ютерів з компонентами, що беруть участь в обміні. В цьому випадку повідомлення спочатку поміщається у вихідну чергу на комп'ютері з компонентом, що посилає повідомлення, а потім пересилається менеджерові.

Для створення великих систем обміну повідомленнями може використовуватися маршрутизація повідомлень, при якій повідомлення не передаються безпосередньо менеджерові, що підтримує чергу, а проходять через ряд проміжних менеджерів черг повідомлень.

Використання черг повідомлень орієнтоване на *асинхронний обмін даними.*

Віддалений виклик процедур

Ідея віддаленого виклику процедур (remote procedure call, RPC) з'явилася в середині 80-х років і полягала в тому, що *функцію на віддаленому комп'ютері можна викликати так само, як і функцію на локальному комп'ютері*. Аби віддалений виклик відбувався прозоро з точки зору застосунку, необхідно надати процедуру-перехідник (заглушку, stub), яка викликатиметься клієнтським застосунком. Після виклику процедури-перехідника проміжне середовище перетворить передані нею аргументи у вигляд, придатний для передачі по транспортному протоколу, і передасть їх на віддалений комп'ютер з функцією, що викликається. На віддаленому комп'ютері параметри витягуються проміжним середовищем з повідомлення транспортного рівня і передаються функції, що викликається. Аналогічним чином на клієнтську машину передається результат виконання викликаної функції (рис. 3.1).

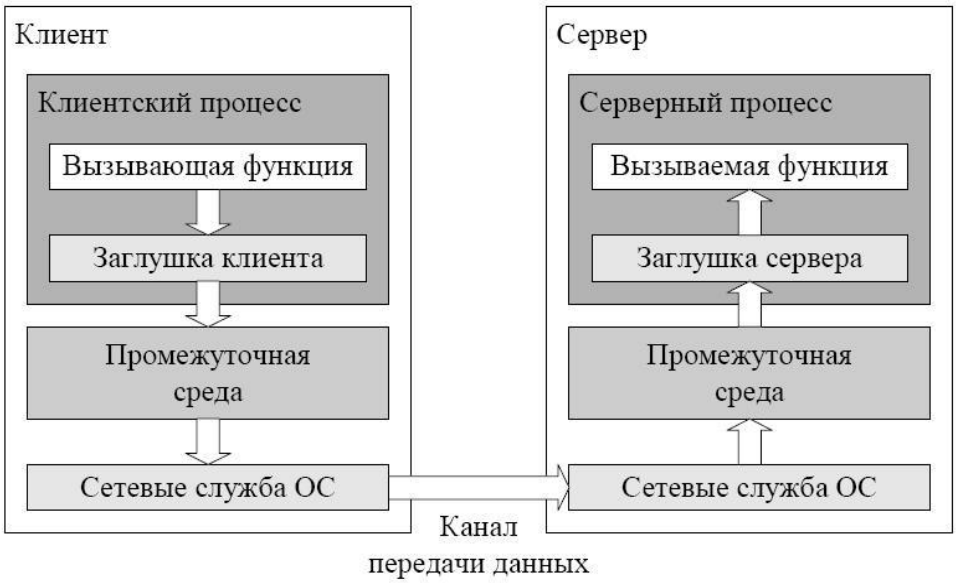

Рис. 3.1. Віддалений виклик процедур

Сама ідея віддаленого виклику проста, але не проста в реалізації, оскільки процес і процедура знаходяться на різних машинах в різних адресних просторах. Є проблеми і при передачі параметрів.

Існує три можливі варіанти віддаленого виклику процедур.

1. *Синхронний виклик*: клієнт чекає завершення процедури сервером і при необхідності отримує від нього результат виконання віддаленої функції;

2. *Однонаправлений асинхронний виклик*: клієнт продовжує своє виконання, отримання відповіді від сервера або відсутнє, або його реалізація покладена на розробника;

3. *Асинхронний виклик*: клієнт продовжує своє виконання, при завершенні сервером виконання процедури він отримує повідомлення і результат її виконання, наприклад через callback-функцію, що викликається проміжним середовищем при отриманні результату від сервера.

Віддалений виклик процедур (Remote Procedure Call, RPC) – це один з ранніх методів організації зв'язку, який лежить в основі багатьох сучасних технологій розподілених систем.

Сучасні моделі інтеграції, розроблені з використанням об'єктно-орієнтованого підходу, використовують виклик віддалених об'єктів.

3.3.2. Виклик віддалених об'єктів. Маршалінг і серіалізація

Технологія RMI (Remote Method Invocation) – це розвиток RPC (його об'єктна реалізація).

Віддалений об'єкт містить деякі дані, сукупність яких визначає його стан. Цей стан можна змінювати шляхом виклику його методів. Зазвичай можливий прямий доступ до даних віддаленого об'єкту, при цьому відбувається неявний віддалений виклик, необхідний для передачі значення поля даних об'єкту між процесами. Методи і поля об'єкту, які можуть використовуватися через віддалені виклики, доступні через деякий зовнішній інтерфейс класу об'єкта. Зовнішній інтерфейс компонента зазвичай збігається із зовнішнім інтерфейсом одного з класів компонента.

Як і в RPC, коли клієнт починає використовувати віддалений об'єкт, на стороні клієнта створюється клієнтська заглушка, яка називається посередником (proxy). Посередник реалізує той самий інтерфейс, що і віддалений об'єкт. Викликаючий процес використовує методи посередника, який перетворює (*маршалізує*) їх параметри для передачі по мережі, і передає їх по мережі серверу. Проміжне середовище на стороні сервера *десеріалізує* параметри і передає їх заглушці на стороні сервера, яку називають каркасом (skeleton) або, як і у віддаленому виклику процедур, заглушкою. Каркас зв'язується з деяким екземпляром віддаленого об'єкту. Це може бути як новий, так і існуючий екземпляр об'єкту, залежно від моделі використання віддалених об'єктів.

Весь описаний процес називається **маршалінгом** віддаленого об'єкту за **посиланням** (marshal by reference). На відміну від маршалінгу за значенням, екземпляр об'єкту знаходиться в процесі сервера і не покидає його, а для доступу до об'єкту клієнти використовують посередників.

При маршалінгу ж за значенням саме значення об'єкту серіалізується в набір байтів для його передачі між процесами, після чого слідує створення його копії в іншому процесі. На теперішній час частіше використовується XML-серіалізація.

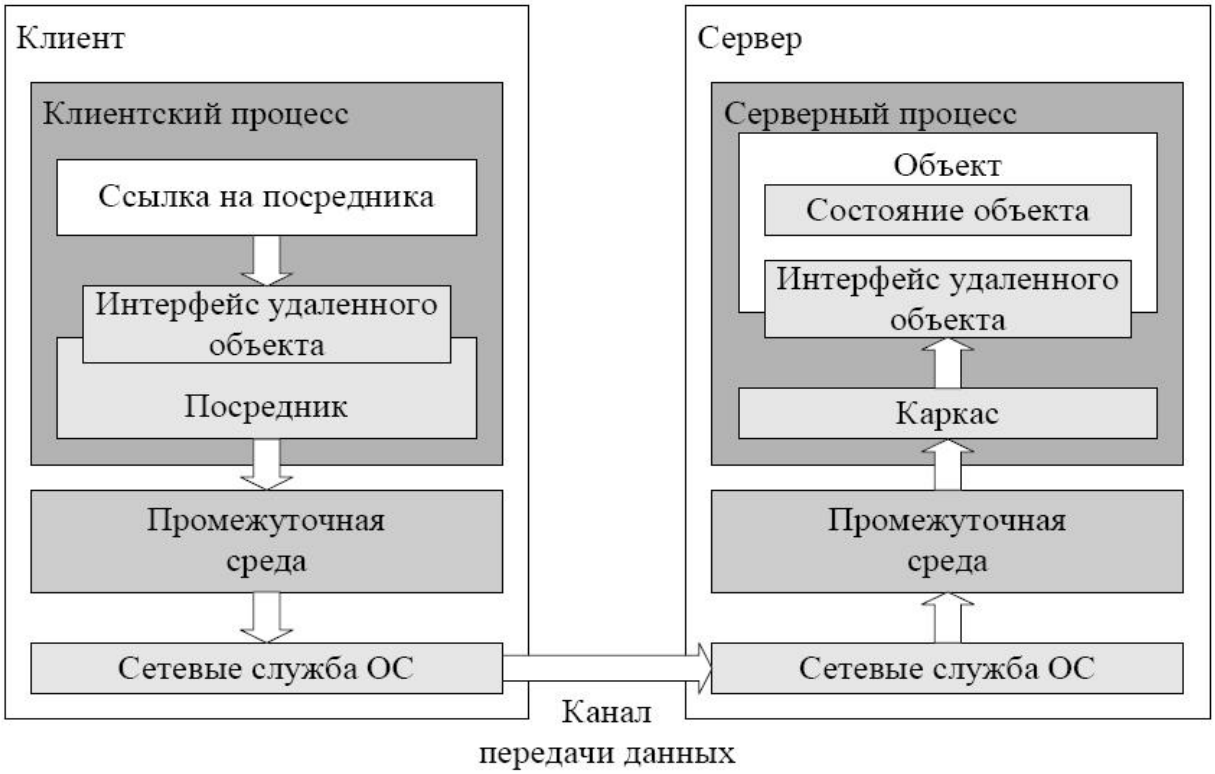

Рис. 3.2 Використання віддалених об'єктів

Таким чином, дамо визначення понять **маршалінг** та **серіалізація.**

Означення 1 (Вікіпедія)

Маршалінг (від англ. 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 демаршалінгом.

Серіалізация – процес перетворення якої-небудь структури даних в послідовність бітів. Зворотною до операції серіалізації є операція десеріалізації – відновлення початкового стану структури даних з бітової послідовності.

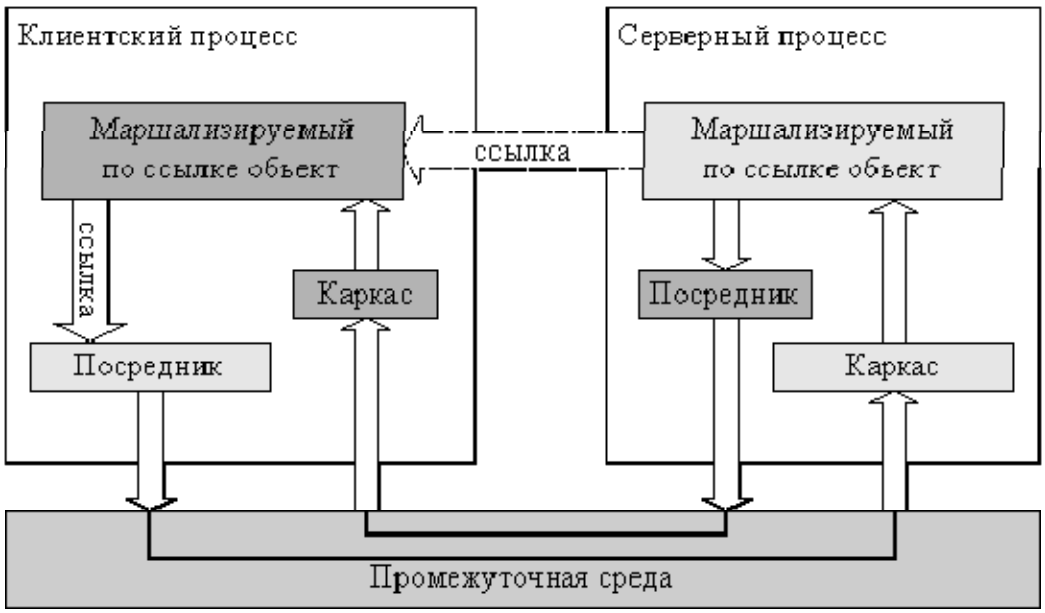

Рис. 3.3. Передача віддаленому методу посилання на об'єкт, який маршалізується за посиланням

3.3.3. Підходи до інтеграції компонентів інформаційних систем

Для оцінки існуючих технологій інтеграції компонентів інформаційних систем можна виділити наступні категорії технологічних рішень, що характеризують погляди конкретної організації на архітектуру інформаційної системи в цілому:

1. Спеціалізовані рішення;
2. Змішані механізми;
3. Віддалені виклики процедур (RPC);
4. Моделі розподілених об'єктів (CORBA, DCOM, .Net);
5. Frameworks;
6. Стандартна архітектура (Standard Architectures)

Розглянемо стисло кожну з категорій.

Технологічні рішення, що відносяться до першої категорії, базуються на основі власних (унікальних для даної організації) протоколів і інтерфейсів взаємодії. Такі рішення, в більшості випадків, призводять до проблем при спробі організації взаємодії компонентів даної системи з компонентами програмних систем, побудованих на основі інших рішень міжкомпонентної взаємодії.

До другої категорії відносяться технологічні рішення, що передбачають побудову інформаційних систем з розрахунку на конкретне завдання і спочатку не розраховані на використання технологій інтеграції систем. Тому в цьому випадку у якості засобів інтеграції використовуються такі механізми, як сокети (sockets) протоколу TCP/IP і ONC RPC (Object Network Computing Remote Procedure Call). Програмне забезпечення цієї категорії, як правило, розробляється для використання лише усередині

конкретної організації, що призводить до різкого збільшення витрат при інтеграції з іншими системами.

До третьої категорії відносяться технології на базі механізму виклику віддалених процедур (RPC), що надає інтерфейс низького рівня.

У четвертій категорії, при побудові інформаційних систем використовуються моделі проміжного рівня, які надають інтерфейс високого рівня, порівняно з RPC. В цих моделях застосовують сервіси і мову описи інтерфейсів (OMG IDL), визначені в специфікації CORBA, лише для забезпечення міжплатформеної взаємодії. У разі, коли міжплатформенна взаємодія не потрібна, використовуються власні механізми взаємодії компонентів системи. Слід зазначити, що ці системи ризикують технологічно застаріти, використовуючи специфічні механізми, що не забезпечують реальних переваг програмної архітектури CORBA.

Технологічні рішення п'ятої категорії передбачають розробку платформ взаємодії компонентів (frameworks) як основи програмної архітектури для використання в декількох проектах. Як правило, framework втілює ретельно продумані принципи побудови програмної архітектури.

До шостої категорії відносяться високоякісні технології, програмна архітектура і сервіси, які розраховані на повторне використання в багатьох різних проектах. Як правило, такі рішення спираються на стандарт CORBA, що зрештою дозволяє організаціям практично нанівець звести ризик застарівання систем. Прикладом вдалої реалізації сучасної платформи є .Net Framework Microsoft.

Висновки

Основним принципом побудови сучасних програмних систем є модульність. Розділення на модулі (компоненти) спрощує реалізацію великої системи, при цьому слід дотримуватися основних принципів модульності: розділення інтерфейсу і реалізації, слабка зв'язність та сильна спорідненість, повторне використання.

Важливу роль відіграють інтерфейси компонентів, які встановлюють правила взаємодії компонентів та інтерфейси проміжного рівня, які забезпечують взаємодію віддалених компонентів.

Контрольні запитання і завдання для самостійної роботи

1. Які основні принципи модульності?
2. Яка різниця між висхідною і низхідною стратегіями проектування системи? які переваги та недоліки має кожна з них?
3. Які основні вимоги висуваються до інтерфейсу компонента?
4. Чому інтерфейс компонента називають його **контрактом**?
5. У чому полягає ідея обміну повідомленнями? Назвіть можливі варіанти віддаленого виклику процедур.
6. У чому полягає ідея віддаленого виклику процедур?
7. Яке призначення менеджера черги повідомлень?

8. Що таке **маршалінг** і для чого він застосовується? Яка різниця між маршалінгом об'єкта за посиланням і за значенням?

9. Які можуть бути технології інтеграції компонентів?

10. Що означає принцип повторного використання компонентів? Які переваги він надає розробникам програмного забезпечення?