Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Інформаційні системи

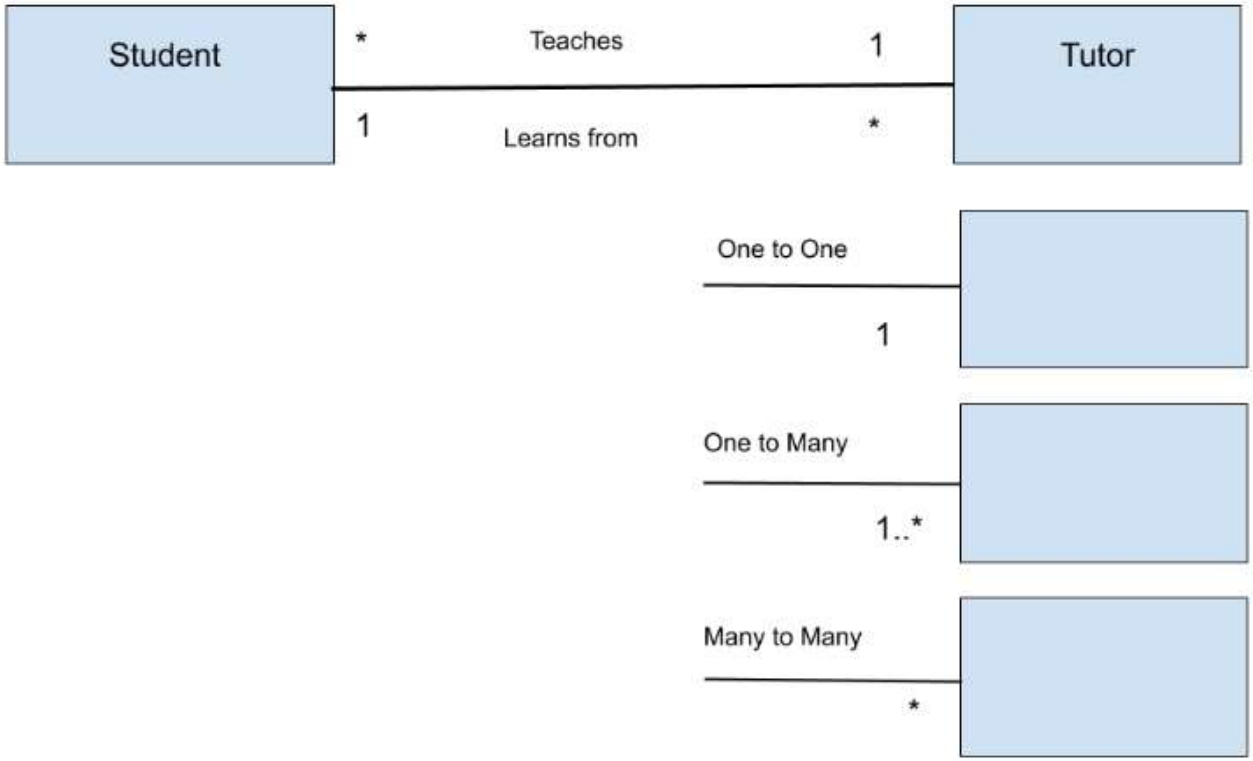
Викладач: к.т.н., доц. Саяпіна Інна Олександрівна

План заняття:

- **Типи відношень у UML Class Diagram**
- **Зчеплення (coupling) та зв’язність (cohesion)**
- **Їх вплив на архітектуру**
- **Архітектурні стилі та шаблони (паттерни): навіщо вони потрібні?**

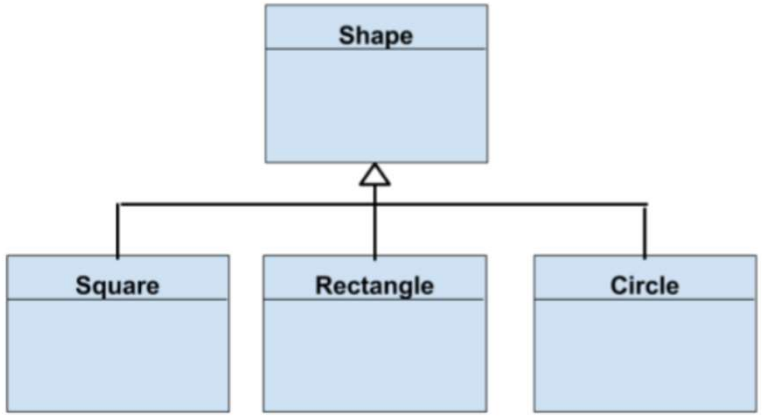
Типи відношень у UML Class Diagram

- **Асоціація**



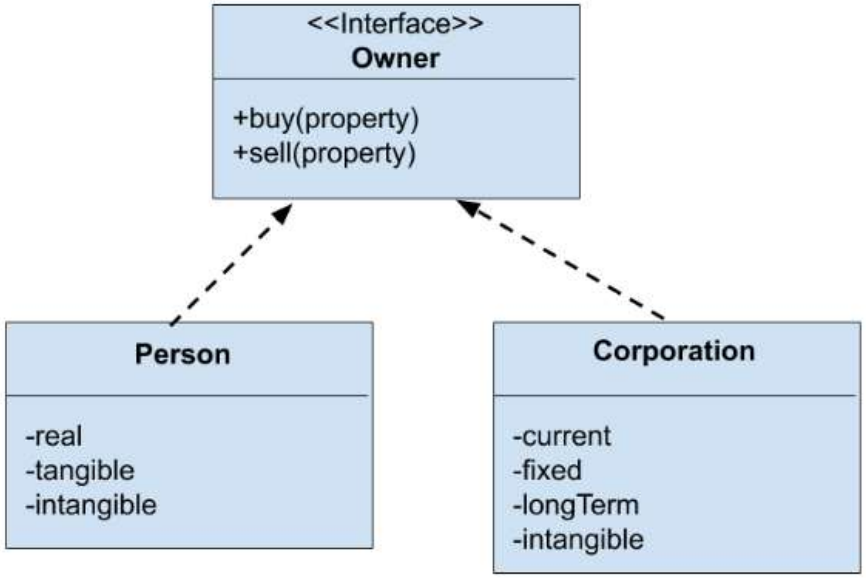
Типи відношень у UML Class Diagram

- **Узагальнення (наслідування)** це схематичне зображення відносин між батьківським класом та його спадкоємцями. Порожня стрілка завжди спрямована до батьківського класу.
- Можна зображувати успадкування як окремо кожного класу, і об'єднувати їх.
- Якщо успадкування походить від абстрактного класу, ім'я такого батьківського класу записується курсивом.



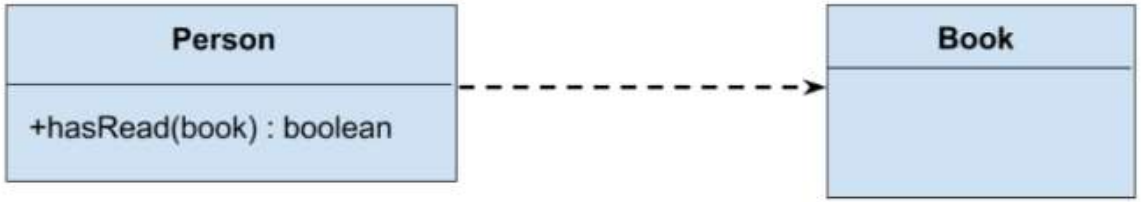
Типи відношень у UML Class Diagram

- Під **реалізацією** мається на увазі відношення інтерфейсу та об'єктів, що реалізують цей інтерфейс. Наприклад, інтерфейс Owner має методи для купівлі та продажу приватної власності, а відносини класів Person і Corporation, що реалізують цей інтерфейс, на діаграмі позначатимуться у вигляді пунктирної лінії зі стрілкою у напрямку до інтерфейсу.



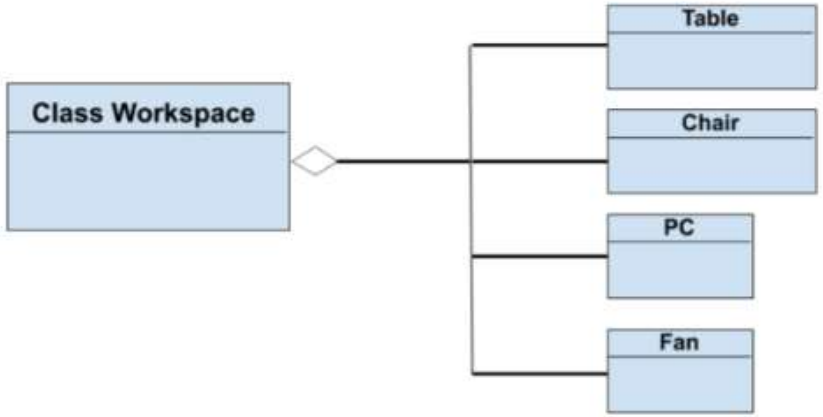
Типи відношень у UML Class Diagram

- Об'єкт одного класу може використовувати об'єкт іншого класу у своєму методі. Якщо об'єкт не зберігається у полі класу, такий вид міжкласових відносин моделюється як **залежність**.



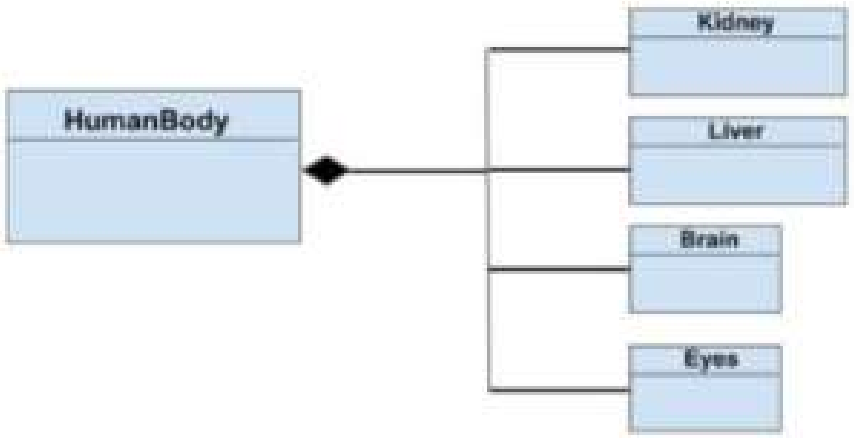
Типи відношень у UML Class Diagram

- Агрегація - особливий тип відносин між класами, коли один клас є частиною іншого.



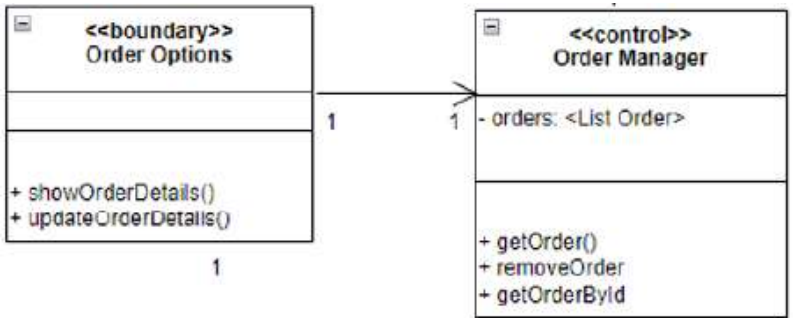
Типи відношень у UML Class Diagram

- **Композиція** - по суті, різновид агрегації, тільки в цьому випадку, класи, що є частиною іншого класу, знищують, коли знищується клас-агрегатор.

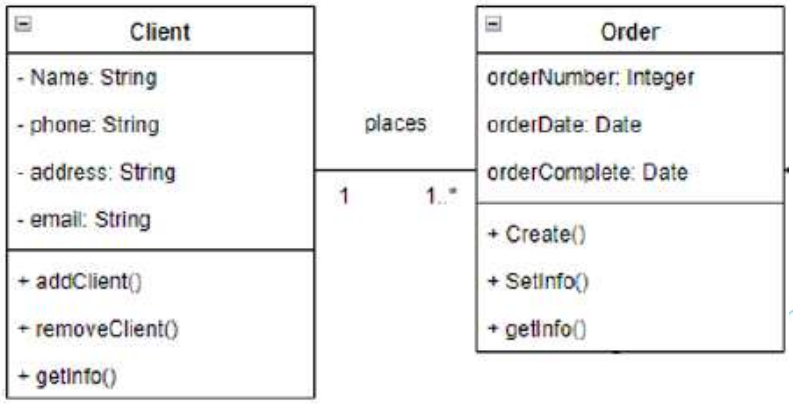


Двонаправлена і однонаправлена асоціація

- В об'єктно-орієнтованому програмуванні асоціація відноситься до зв'язку між двома або більше класами. Існує два типи асоціації:
- **однонаправлена**



- І **двонаправлена**



Обидва класи знають один про одного і можуть отримати доступ до властивостей і методів один одного.

Однонаправлена асоціація

- Однонаправлена асоціація стосується зв’язку між двома класами, де один клас містить посилання на інший клас, але інший клас не містить посилання на перший клас. Іншими словами, відносини односторонні.
- Тобто, однонаправленій асоціації тільки один клас знає про інший клас. Інший клас не знає про перший клас. Наприклад, розглянемо такий однонаправлений зв’язок між класами **Person** та **Address**:

```
public class Person {
    private Address address;

    public void setAddress(Address address) {
        this.address = address;
    }

    public Address getAddress() {
        return address;
    }
}

public class Address {
    private String street;
    private String city;
    private String state;
    private String zip;

    // getters and setters
}
```

Двонаправлена асоціація

- У двонаправленій асоціації два класи знають один про одного і можуть викликати методи один одного. Це означає, що зміни, внесені до одного класу, можуть вплинути на інший.
- Наприклад, розглянемо такий

двонаправлений зв’язок між класами

Customer та **Order**:

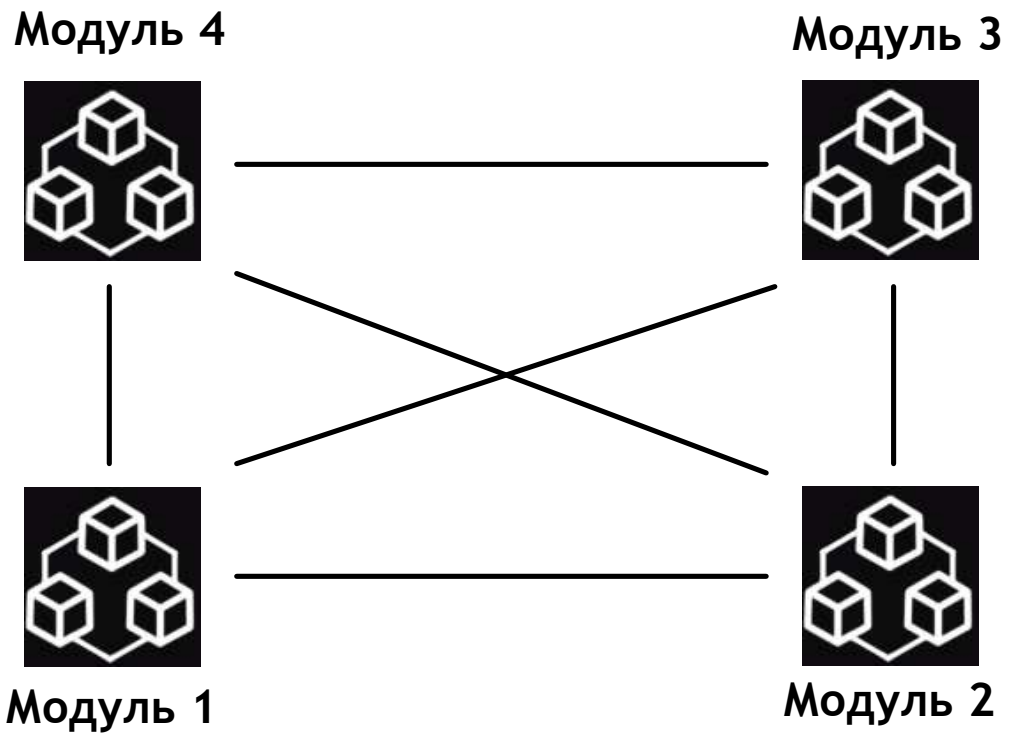
```
public class Customer {
    private List<Order> orders = new ArrayList<>();

    public void addOrder(Order order) {
        orders.add(order);
        order.setCustomer(this);
    }

    public List<Order> getOrders() {
        return orders;
    }
}

public class Order {
    private Customer customer;

    public void setCustomer(Customer customer) {
        this.customer = customer;
    }

    public Customer getCustomer() {
        return customer;
    }
}
```

Двонаправлена і однонаправлена асоціація

- Вибір між двонаправленою та однонаправленою асоціацією залежить від конкретного випадку використання та вимог системи, що проектується.
- Двонаправлені асоціації можуть забезпечити більшу гнучкість у взаємодії класів один з одним, але також можуть створити додаткову складність і потенційну можливість помилок.
- Однонаправлені асоціації можуть спростити дизайн і зменшити потенційні помилки, але можуть бути менш гнучкими у взаємодії класів.

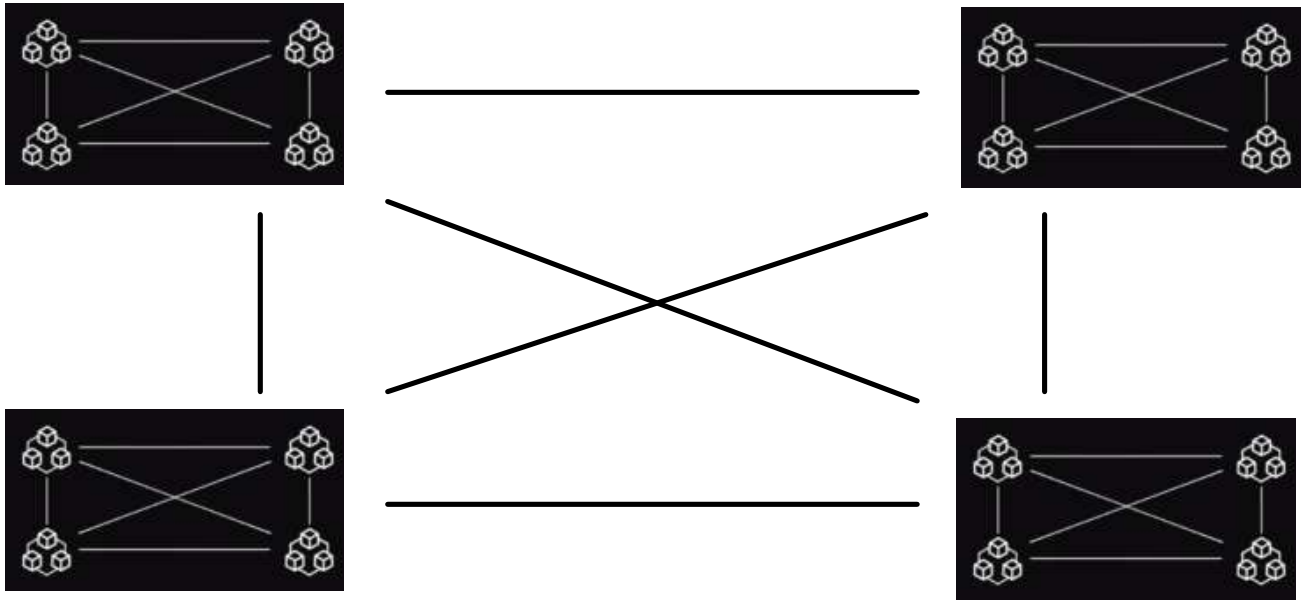
Зчеплення (coupling) та зв’язність (cohesion)

- *Що це таке?*



Зчеплення та зв’язність

- *Що це таке?*
- Хаотичні зв'язки зі зростанням проекту ускладнюють підтримку, рефакторинг та додавання нового функціоналу.



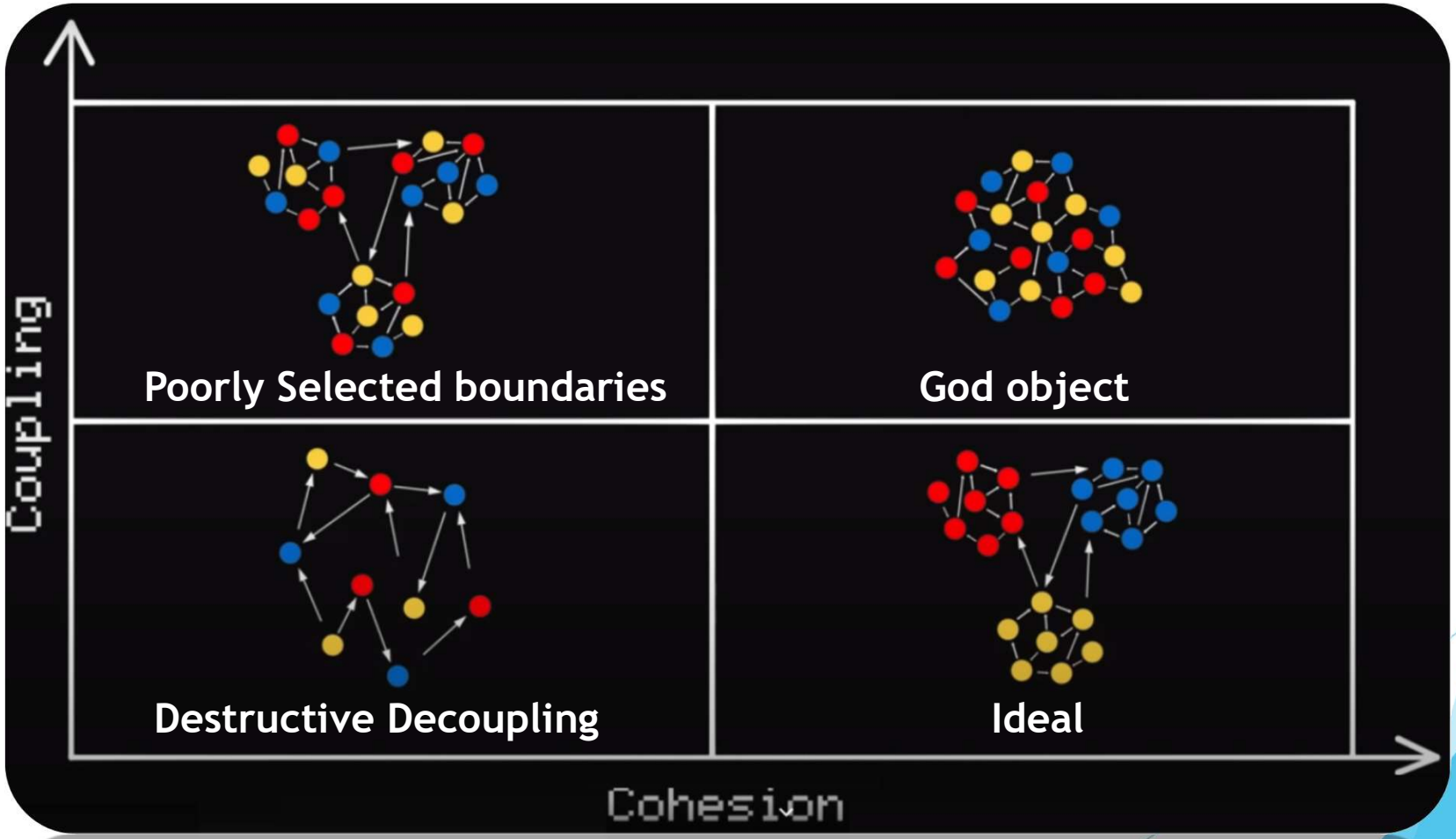
Зчеплення та зв’язність

- **Модуль** — фрагмент програмного коду, що є будівельним блоком для фізичної структури системи.
- Модульність — властивість системи, яка може піддаватися декомпозиції на ряд внутрішньо зв'язаних і слабо залежних один від одного модулів.
- Якщо система має поганий дизайн, то модулі можуть підключатися лише до певних конкретних модулів і ні до яких інших. Гарна конструкція дозволяє з'єднувати будь-які модулі між собою без особливих проблем. Іншими словами, у хорошому дизайні модулі сумісні один з одним, тому їх можна легко з’єднати та використовувати повторно.
- Метрики, які часто використовуються для оцінки складності дизайну, - це зв'язність і зчеплення.

Зчеплення (coupling) та зв’язність (cohesion)

- Модулі, з якими ми працюємо, повинні мати:
  - сильну зв’язаність (cohesion) - внутрішні компоненти модуля спрямовані на рішення однієї чіткої задачі
  - слабке зчеплення (coupling) - залежність від інших модулів повинна бути найменшою

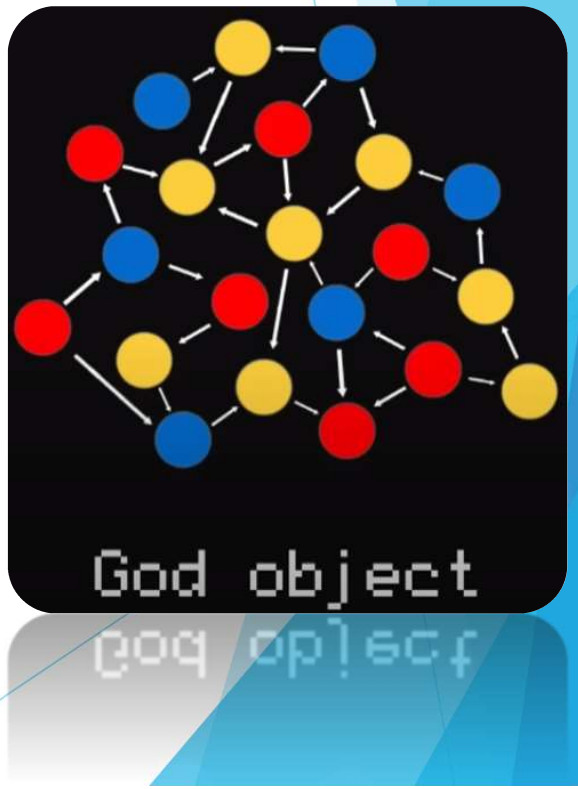
Зчеплення (coupling) та зв’язність (cohesion)



Вплив зчеплення (coupling) та зв’язності(cohesion) на архітектуру (01)

Антіпаттерн God object

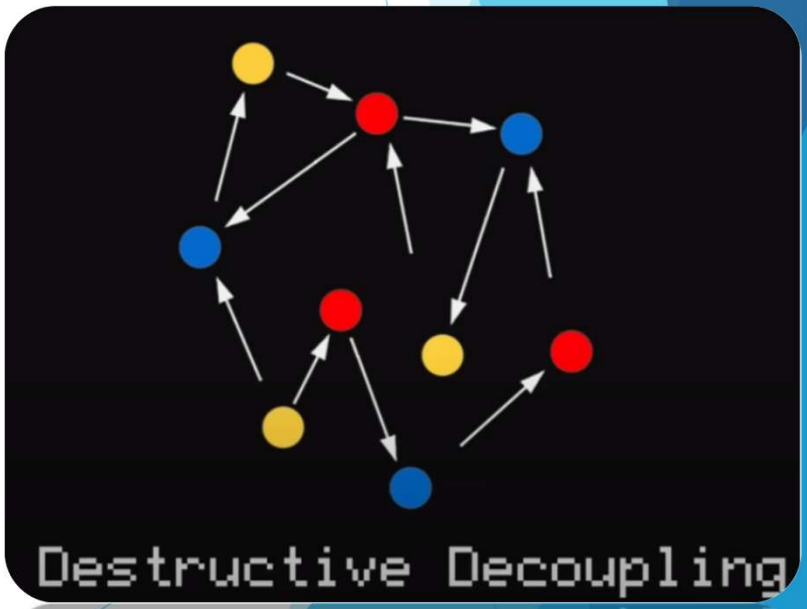
- Класи вирішують велику кількість непов’язаних задач
- Немає чітко виділених модулів
- Велика кількість зв’язків, які складно зрозуміти



Вплив зчепленності (coupling) та зв’язності(cohesion) на архітектуру (02)

Destructive Decoupling

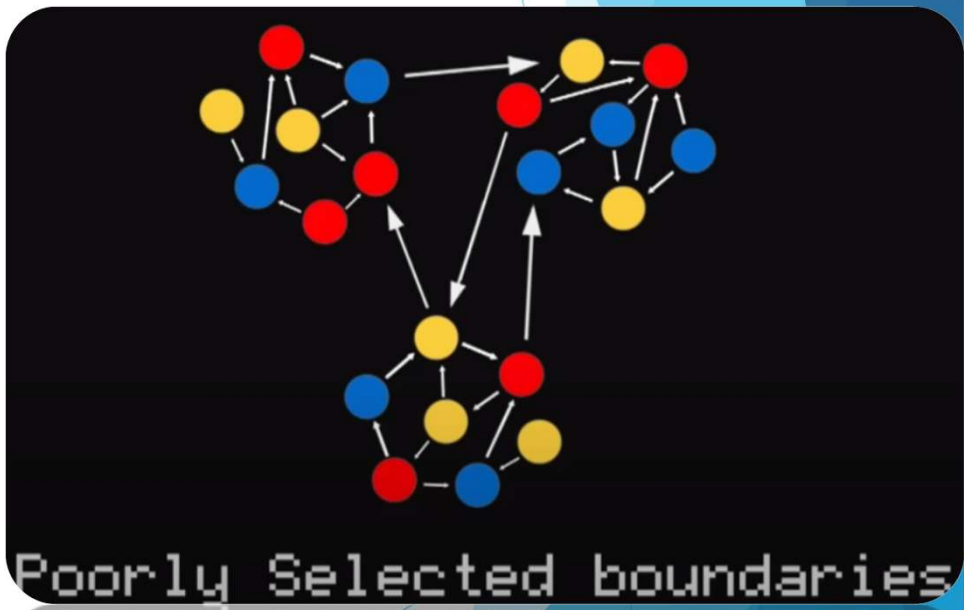
- Слабка зв’язаність та слабка зчепленість
- цеглинки, з яких будується модуль, вирішують різні завдання (різного кольору)
- всередині модуля є хаотичність і слабка зв'язаність цих цеглинок
- між модулями зв'язки не явні



Вплив зчепленості (coupling) та зв’язності(cohesion) на архітектуру (03)

Poorly Selected Boundaries

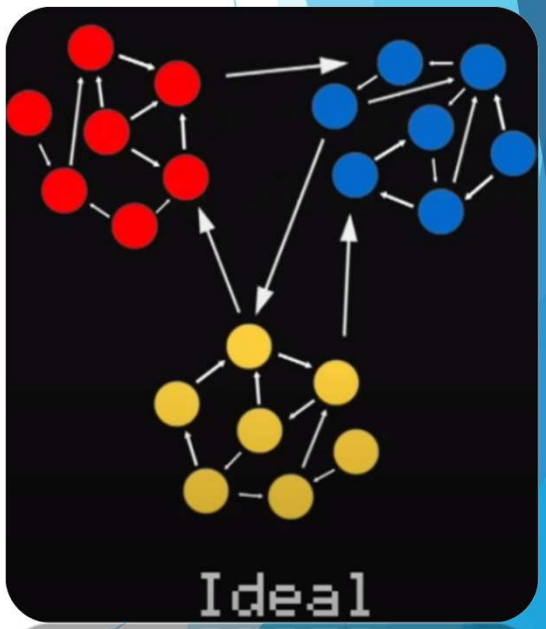
- Чітко виділені модулі та зв’язки між ними
- Слабка зв’язність усередині модулів
- Можна видалити модуль без шкоди для додатка



Вплив зчепленості (coupling) та зв’язності(cohesion) на архітектуру (04)

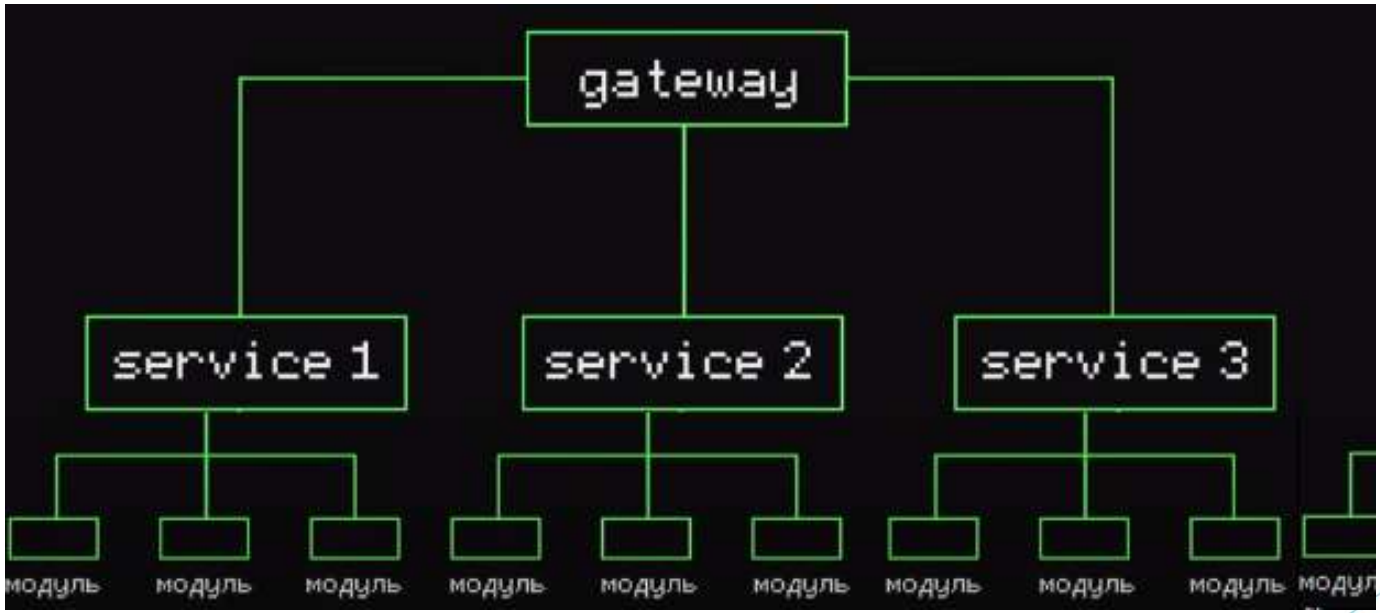
Ideal

- Зчепленість між модулями слабка, тобто модуль за необхідності легко можна видалити
- всередині модуля у нас компоненти вирішують одне завдання та сильно зв'язані один з одним
- Легко видаляти та змінювати модулі



Архітектура різних рівнів

Приклад: мікросервісна архітектура



Зчепленість (coupling)

- **Зчепленість** визначається складністю взаємодії між модулем та іншими модулями.
- Слабка зчепленість означає, що модулю легко підключитися до інших модулів через чітко визначені інтерфейси.
- Для того, щоб оцінити зв’язок модуля, необхідно враховувати такі по казники: ступінь, легкість і гнучкість.
- **Ступінь (degree)** - це кількість зв'язків між модулем та іншими. Ступінь повинен бути невеликим для зчепленості. Наприклад, модуль має з’єднуватися з іншими за допомогою лише кількох параметрів або вузьких інтерфейсів. Це означає малий ступінь та слабку зчепленість.
- **Легкість (ease)** полягає в тому, наскільки очевидними є зв’язки між модулем та іншими. Підключення має бути легким для встановлення без необхідності розуміти реалізацію інших модулів для цілей зв’язку.
- **Гнучкість (flexibility)** вказує на те, наскільки інші модулі є взаємозамінними для цього модуля. Інші модулі мають бути легко замінені на щось краще в майбутньому для цілей з’єднання.

Ознаки високої зчепленості (High coupling)

- Ознаками того, що система високу зчепленість та поганий дизайн, є:
- • модуль підключається до інших модулів через велику кількість параметрів або інтерфейсів
- • відповідні даному модулю інші модулі важко знайти
- • модуль можна підключити лише до певних інших модулів та він не може бути замінений

Зв’язність (cohesion)

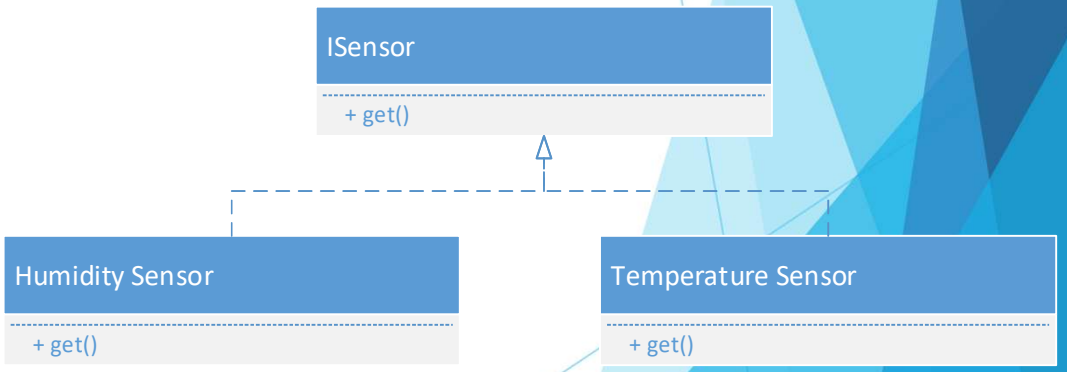
- **Зв’язність** визначає складність всередині модуля та чіткість його обов’язків.
- Модуль, який виконує одне завдання і нічого іншого, або має чітку мету та високу зв’язність. Це дбрий дизайн. З іншого боку, якщо модуль інкапсулює більше, ніж одну мету, якщо інкапсуляцію потрібно порушити, щоб зрозуміти метод, або якщо модуль має незрозумілу мету, він має низьку зв’язність.
- Якщо модуль має більше ніж одну відповідальність, гарною ідеєю буде розділити модуль.
- Важливо балансувати між низькою зчіпленістю та високою зв’язністю у проектуванні системи. У складних системах складність може бути розподілена між модулями або всередині модулів. Наприклад, Якщо модулі спрощуються для досягнення високої зв’язності, їм може знадобитися більше залежати від інших модулів, таким чином збільшуючи зчепленість. Якщо ж зв’язки між модулями спрощуються для досягнення низької зчепленості, модулям може знадобитися взяти на себе більше обов’язків, таким чином знижуючи зв’язність.

Приклад

- Є клас під назвою Sensor, який має дві мети: отримувати показання вологість (Humidity) та температури (Temperature). Наведений код методу get, який приймає прапорець нуль, якщо ви хочете повернути значення вологості, і приймає прапорець one, якщо ви хочете повернути значення температури. Що можна сказати про зв'язність та зчіпленість класу?

```
public void get (int controlFlag) {
     switch (controlFlag) {
          case 0:
               return this.humidity;
               break;
          case 1:
               return this.temperature;
               break;
          default:
               throw new UnknownControlFlagException();
     }
}
```

Для підвищення зв'язності внесемо зміни:



Навіщо потрібні архітектурні шаблони?

- Архітектурні стилі /шаблони описують визначений зв'язок компонентів, що охоплює різноманітні характеристики архітектури.
- Назва стилю архітектури діє як скорочення між архітекторами.
- Стиль архітектури описує топологію, передбачувані та типові характеристики архітектури, як корисні, так і шкідливі.

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★
Elasticity	★
Evolutionary	★
Fault tolerance	★
Modularity	★
Overall cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★

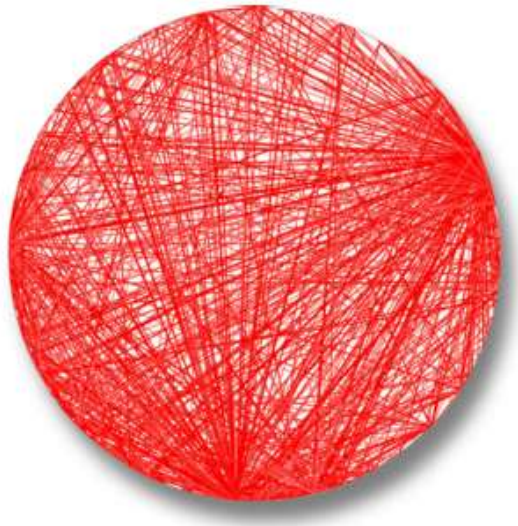
Приклад характеристик багатошарової архітектури

Як можна класифікувати архітектуру ПЗ?

- *За структурою*: організація компонентів програмної системи та їхніх взаємозв'язків. Приклади шаблонів: монолітна, багаторівнева архітектура, архітектура клієнт-сервер і архітектура мікросервісів.
- *За поведінкою*: ця категорія відноситься до динамічної поведінки програмної системи, включаючи її взаємодію з користувачами та іншими системами. Приклади моделей поведінкової архітектури: Архітектура, керовану подіями (Event-Driven architecture), Архітектура кінцевого автомата (State Machine architecture) та Архітектура класної дошки
- *За типом розгортання*: ця категорія стосується того, як програмна система розгортається на фізичному або віртуальному обладнанні. Приклади шаблонів архітектури розгортання включають хмарну архітектуру, розподілену архітектуру та кластерну архітектуру.
- *За архітектурними стилями*: ця категорія відноситься до широких характеристик, які визначають загальний вигляд програми. Приклади архітектурних стилів включають об'єктно-орієнтовану архітектуру, сервіс-орієнтовану архітектуру та архітектуру, керовану подіями.

Основні архітектурні шаблони

- *Анті-шаблон Big Ball of Mud* - термін, який використовується для опису програмної системи, якій бракує узгодженої архітектури і яка з часом перетворилася на заплутаний безлад коду.
- Відсутність структури ускладнює зміни. ПЗ такого типу страждає від проблем із розгортанням, тестуванням, масштабованістю та продуктивністю.
- Приклад: нашвидкоруч створений веб-додаток на основі Java, після кількох років роботи. Технічна візуалізація показує архітектурний зв’язок: кожна крапка по периметру кола представляє клас, а кожна лінія — зв’язки між класами. У цій базі коду будь-яка зміна класу ускладнює роботу та передбачити побічні ефекти будь-якої зміни для інших класів є жахливою справою.



Основні архітектурні шаблони

- *Unitary Architecture (Унітарна архітектура)* - коли вперше з'явилися персональні комп'ютери, програмне забезпечення було зосереджено на окремих машинах. Коли мережеві ПК стали поширеними, з'явилися розподілені системи (такі як клієнт/сервер).
- Існує небагато унітарних архітектур поза межами вбудованих систем та інших середовищ із високими обмеженнями. Як правило, функціональність програмних систем з часом збільшується, що вимагає відокремлення завдань для підтримки характеристик робочої архітектури, таких як продуктивність і масштабування.

Основні архітектурні шаблони

- *Клієнт-серверна архітектура* - стиль в архітектурі, що розділяє технічну функціональність між інтерфейсом і сервером, який називається дворівневою або клієнт-серверною архітектурою. Існує багато різновидів цієї архітектури, залежно від епохи та обчислювальних можливостей.
  - **Desktop + database server**: рання архітектура персонального комп'ютера заохочувала розробників писати багатофункціональні програми в інтерфейсах користувача, відокремлюючи дані на окремому сервері бази даних.
  - **Browser + web server**: з появою сучасної веб-розробки загальним мвсцем поділу став веб-браузер, підключений до веб-сервера (який, у свою чергу підключений до сервера бази даних).
  - **Three-tier** (трьохрівнева архітектура): у Java та .NET стали популярними сервери додатків, компанії почали будувати ще більше рівнів у своїй топології: рівень бази даних, використовуючи промисловий сервер бази даних, рівень додатків, керований сервером додатків, інтерфейс, закодований у згенерованому HTML, JavaScript.

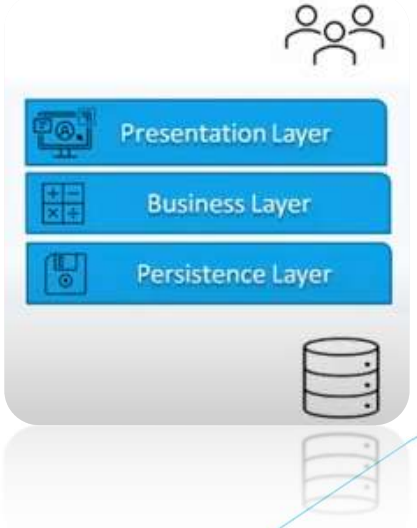
Різниця між N-tier (багаторівневою) та N-layer (багатошаровою) архітектурами

N-tier

- описує фізичні компоненти, тобто процеси додатку, розбиваючи їх на яруси.

N-layer

- Описує логічну структуру компонент, розбиваючи їх на горизонтальні шари: Presentation, Business, Persistence.



Монолітні vs Розподілені архітектури

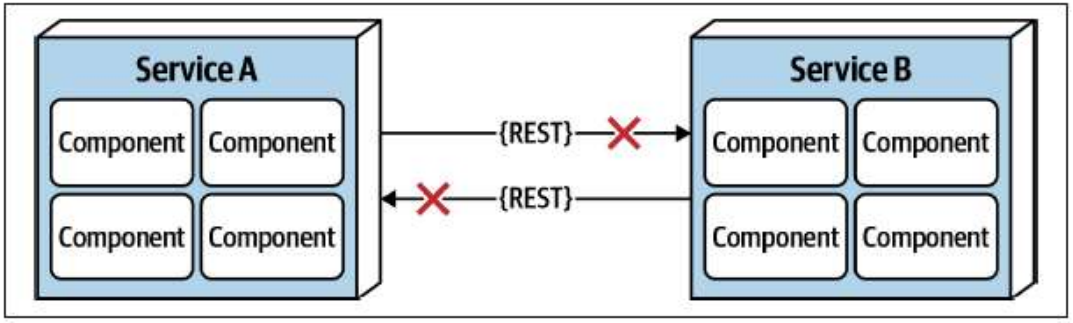
- Стилі архітектури можна поділити на два типи: монолітні (один блок розгортання всього коду) і розподілені (кілька блоків розгортання, з'єднаних через протоколи віддаленого доступу). Хоча жодна схема класифікації не є ідеальною, всі розподілені архітектури мають схожі виклики, з якими не стикаються монолітні стилі архітектури, тому можна навести таку класифікацію:
- *Monolithic*
  - • Layered architecture (Багатошарова архітектура)
  - • Pipeline architecture (Конвеєрна архітектура)
  - • Microkernel architecture (Архітектура мікроядра)
- *Distributed*
  - • Service-based architecture (Сервісна архітектура)
  - • Event-driven architecture (Керована подіями архітектура)
  - • Space-based architecture (Архітектура на основі простору)
  - • Service-oriented architecture (Сервісно-орієнтована архітектура)
  - • Microservices architecture (Архітектура мікросервісів)

Помилкові уявлення про організацію розподілених архітектур

- Стилі розподіленої архітектури набагато потужніші з точки зору продуктивності, масштабованості та доступності, ніж стилі монолітної архітектури. Але задля цього вони змушені йти на значні компроміси. Виклики, з якими стикаються всі розподілені архітектури, описані в помилкових уявленнях розподілених обчислень, вперше сформульованих Л.Пітером Дойчем та інші колеги з Sun Microsystems у 1994 році.
- Помилкові уявлення – це те, що вважається істинним, але не є таким. Описано вісім помилок розподілених обчислень, що застосовуються до розподілених архітектур сьогодні

Помилкове уявлення #1
Мережа є надійною

- Розробники і архітектори припускають, що мережа надійна, але це не так. Це важливо для всіх розподілених архітектур, тому що всі стилі розподіленої архітектури покладаються на мережу для зв'язку до та від служб, а також між службами.
- Наприклад, служба B може бути справною, але служба A не може отримати доступ до неї через проблему з мережею; або ще гірше, служба A надіслала запит службі B для обробки деяких даних і не отримала відповіді через проблему з мережею.
- Ось чому між службами існують такі речі, як тайм-аути та Circuit breakers. Чим більше система покладається на мережу (наприклад, архітектура мікросервісів), тим потенційно менш надійною вона стає.



Representational State Transfer (REST)

- Це архітектурний стиль, який визначає набір принципів і обмежень для створення веб-сервісів. Основні принципи архітектури RESTful включають:
  - **Архітектура клієнт-сервер**. Клієнт і сервер є окремими та взаємодіють за допомогою чітко визначеного інтерфейсу, зазвичай за допомогою HTTP.
  - **Зв'язок без збереження стану**: кожен запит від клієнта до сервера повинен містити всю необхідну інформацію для обробки запиту. Сервер не зберігає клієнтський контекст між запитами.
  - **Кешування**: відповіді від сервера можуть кешуватися клієнтом або будь-якими проміжними кешами для підвищення продуктивності.
  - **Багаторівнева архітектура**: клієнт взаємодіє з сервером через уніфікований інтерфейс, не знаючи про будь-які основні деталі реалізації або проміжні компоненти.
  - Веб-служби RESTful зазвичай використовують такі методи HTTP, як GET, POST, PUT і DELETE для взаємодії з ресурсами на сервері. Ресурси ідентифікуються унікальним URL, і сервер повертає представлення цих ресурсів у відповідь на запити клієнта.
- Популярність веб-сервісів RESTful значно зросла за останні роки завдяки їх простоті, масштабованості та зручності використання. Вони широко використовуються для створення API, мікросервісів та інших розподілених систем.