



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

Інформаційні системи

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

План заняття:

- ▶ Типи нотацій документації програмного забезпечення
- ▶ Засоби для побудови діаграм
- ▶ Use-case Diagram (Діаграма прецедентів)
- ▶ Об'єктно-орієнтований аналіз



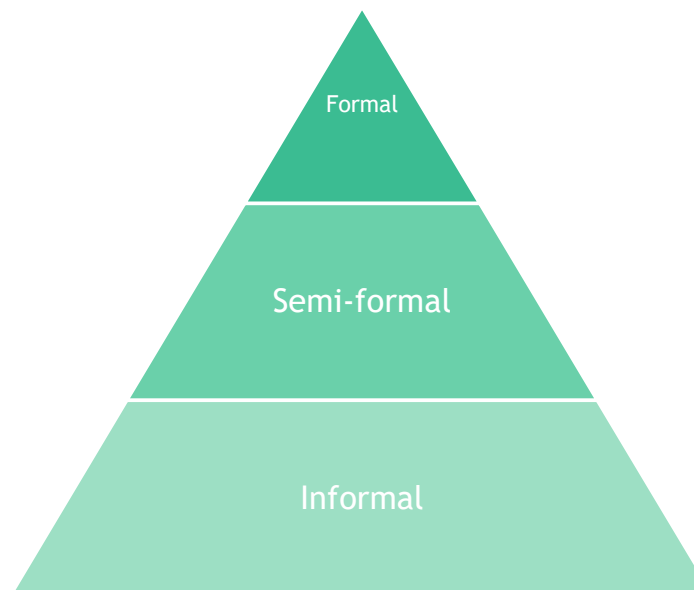
Архітектурна документація

- ▶ **Архітектурна документація** — це схема впровадження системи та проекту, яка визначає, як будуть (виконуються) вимоги до функціональних і якісних атрибутів, фіксує важливі архітектурні рішення та візуалізує архітектуру системи, щоб усі могли її зрозуміти.



Типи нотацій

- ▶ неформальні,
- ▶ напівформальні
- ▶ формальні.



Неформальні нотації

Довільна
форма



Цей тип нотації дозволяє візуалізувати архітектуру на діаграмах довільної форми за допомогою будь-яких інструментів малювання загального призначення. Крім того, немає ніякої особливої візуальної умовності.

Семантика



Семантику можна позначати за допомогою різних форм і кольорів.

Аналіз



Завдяки вільній візуалізації не існує формальних засобів для аналізу діаграм цього типу нотацій.

Інтерпретація



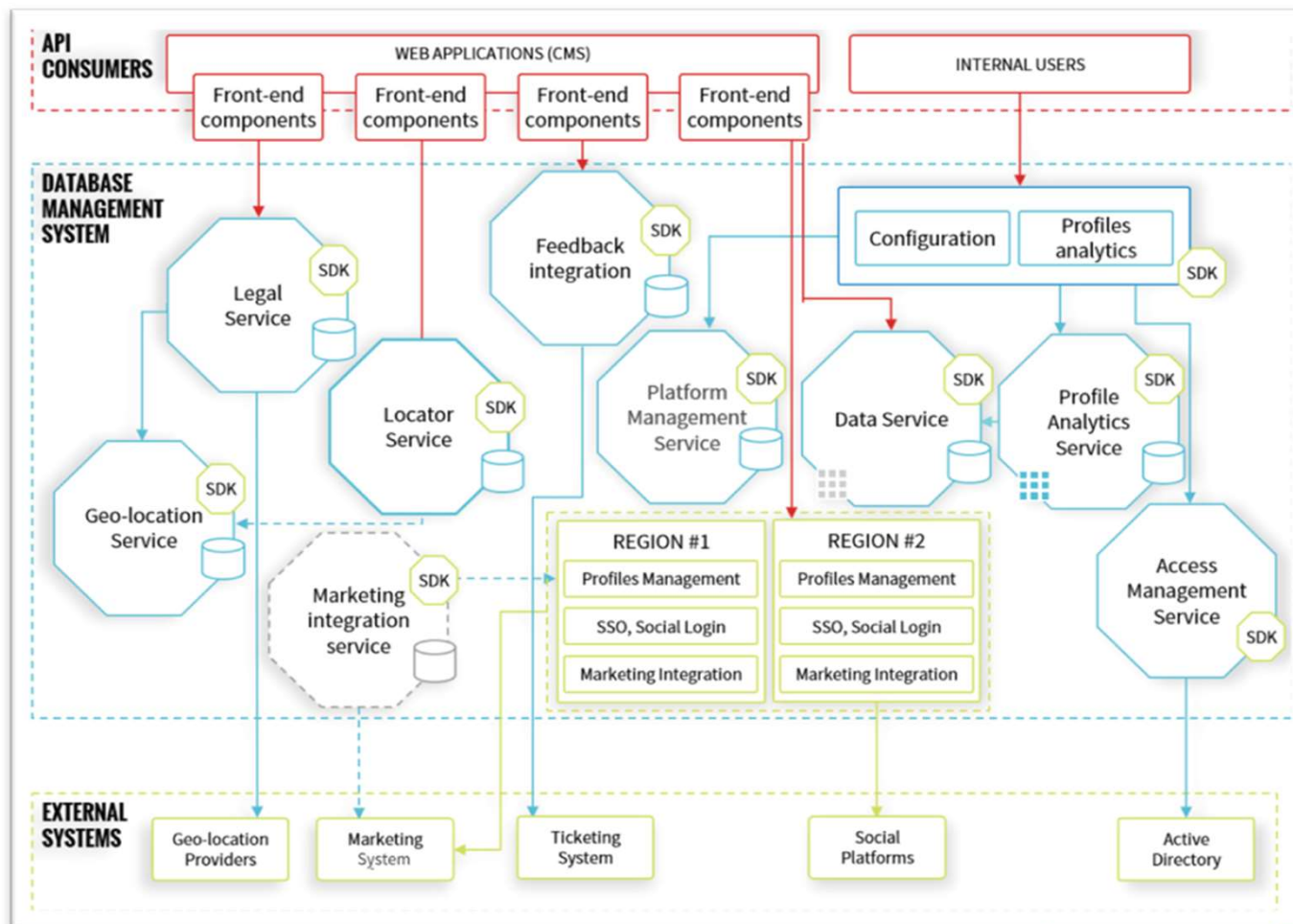
Такі діаграми часто легко зрозуміти людям, які не мають технічних знань. У той же час їх можна неправильно зрозуміти, оскільки неформальні візуалізації важко інтерпретувати.

PowerPoint

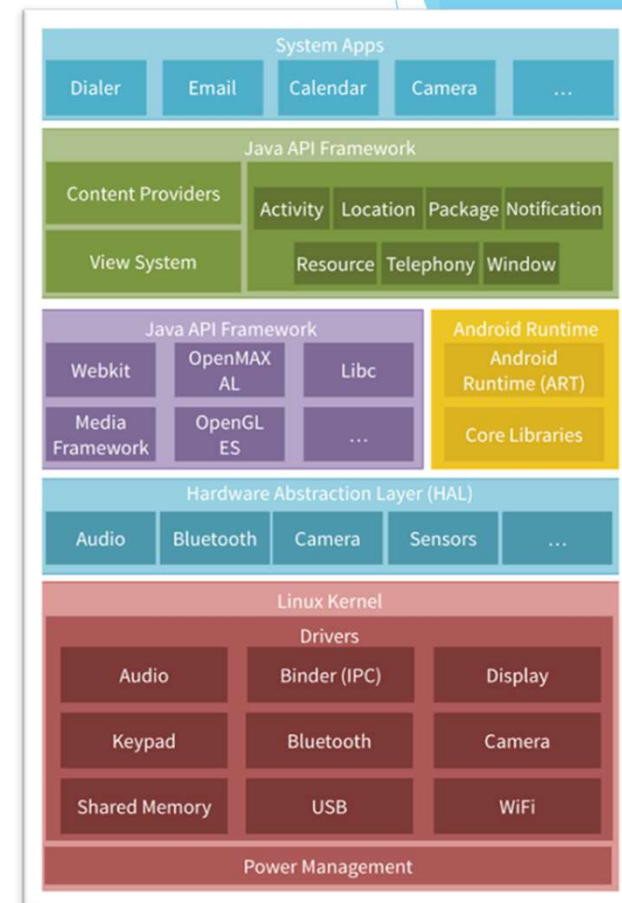
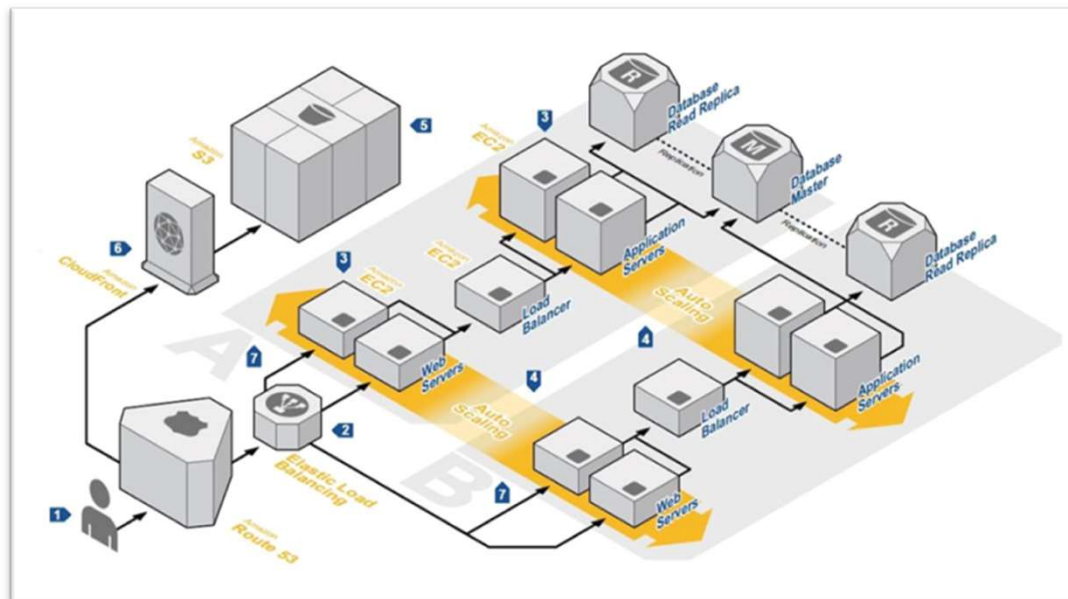


Діаграми неформального типу нотації можна легко створювати за допомогою блоків і ліній у PowerPoint.

Приклади



Приклади



Напівформальні нотації

Стандарт



Напівформальна нотація — стандартизована нотація, де прописані графічні елементи та правила їх використання.

Семантика



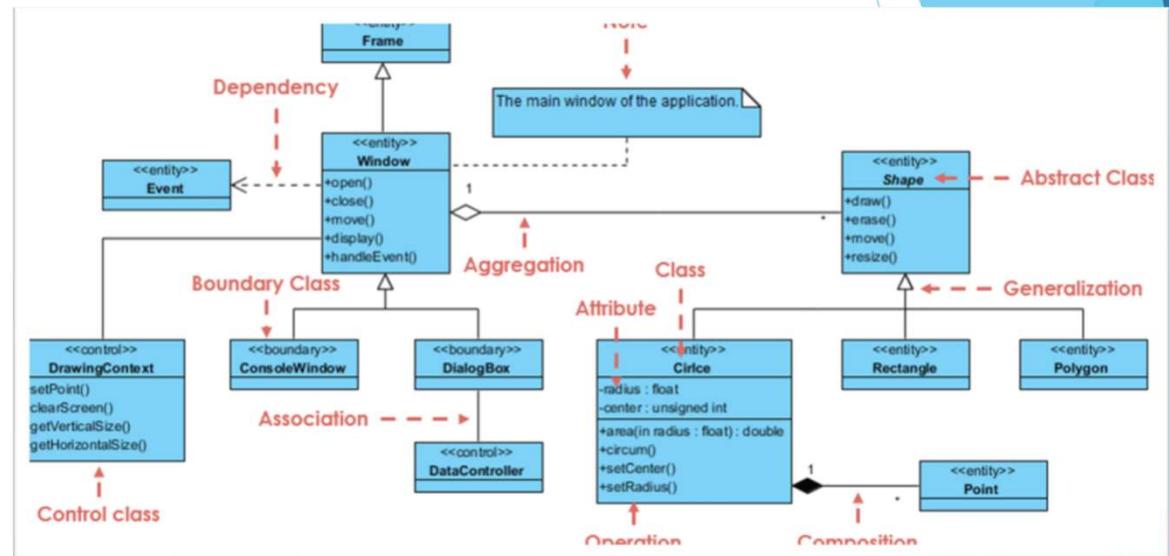
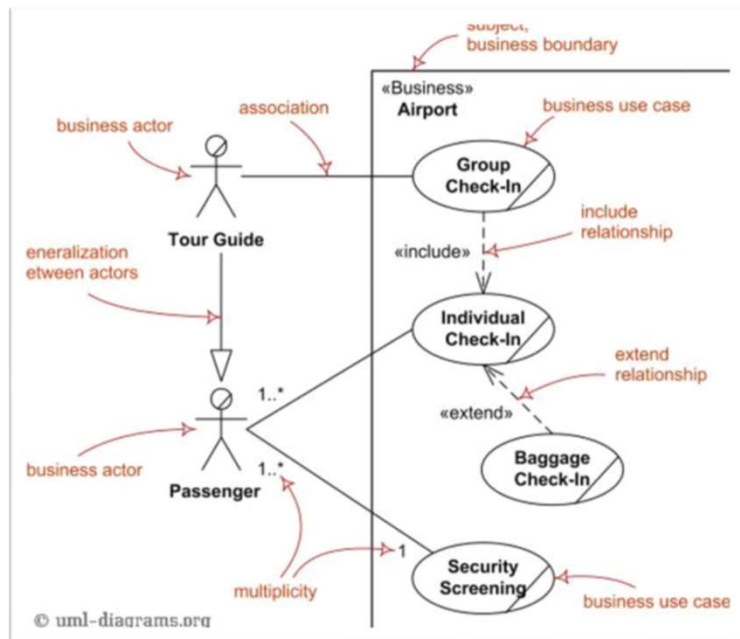
Напівформальний тип нотації не забезпечує повної семантичної обробки значення графічних елементів.

Analysis



Кожен елемент цього типу нотацій містить відповідний контекст і закодований зміст. Це надає можливості для простого аналізу діаграм. Такий простий аналіз може визначити, чи задовольняє опис синтаксичним властивостям.

Приклади



Формальні нотації

Семантика



Має точно визначену семантику

Аналіз



Можливий формальний аналіз як синтаксису, так і семантики дизайну

Автоматизація



Аналіз може виконуватись в автоматизованому режимі за допомогою вищначених інструментів з можливою генерацією коду

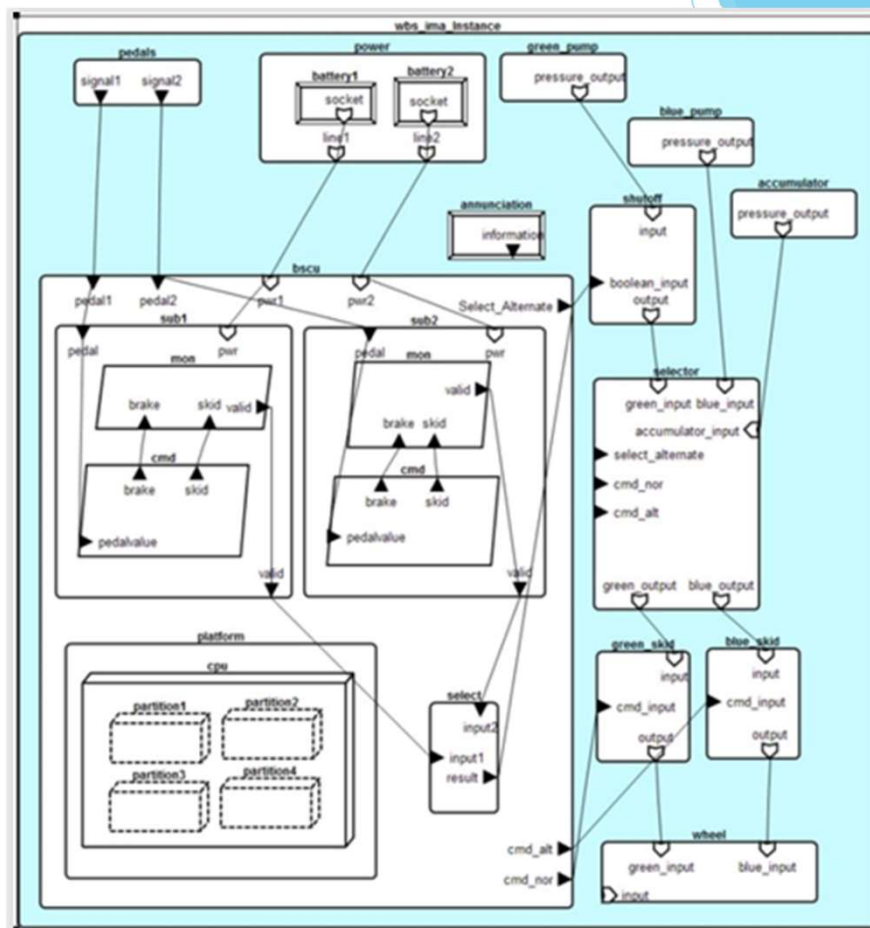
Приклад



Прикладом формальної нотації є AADL(Architecture analysis and Design Language).

Приклад

- ▶ На практиці формальні нотації дуже складні та потребують додаткового програмного забезпечення та знань для ефективного використання.
- ▶ Тому вони зазвичай не використовуються для програмного забезпечення. Але використовуються для інженерії апаратного забезпечення.



UML - (англ. "Unified Modeling Language") - стандартизована мова моделювання при проєктуванні програм.

► **Види діаграм UML:**

○ **Структурні діаграми:**

- Класів (Class diagram)
- Компонентів (Component diagram)
- Композитної/складеної структури (Composite structure diagram)
- Розгортання (Deployment diagram)
- Об'єктів (Object diagram)
- Пакетів (Package diagram)

○ **Діаграми поведінки:**

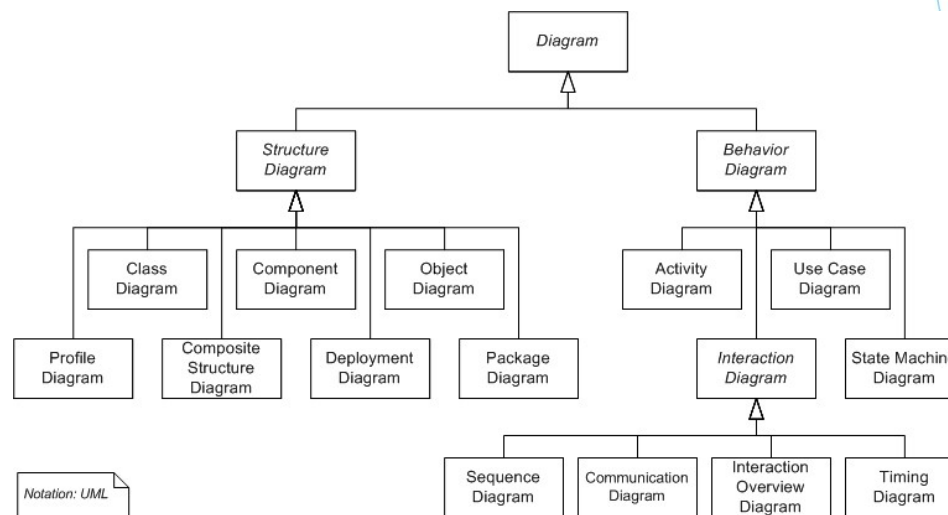
- Діяльності (Activity diagram)
- Станів (State Machine diagram)
- Прецедентів/варіантів використання (Use case diagram)

○ **Діаграми взаємодії:**

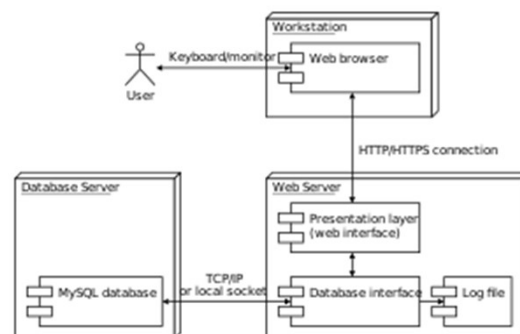
- Комунікації (Communication diagram)
- Огляду взаємодії (Interaction overview diagram)
- Послідовності (Sequence diagram)
- Синхронізації (UML Timing Diagram)

Приклади діаграм UML

- Діаграма класів



- Діаграма розгортання

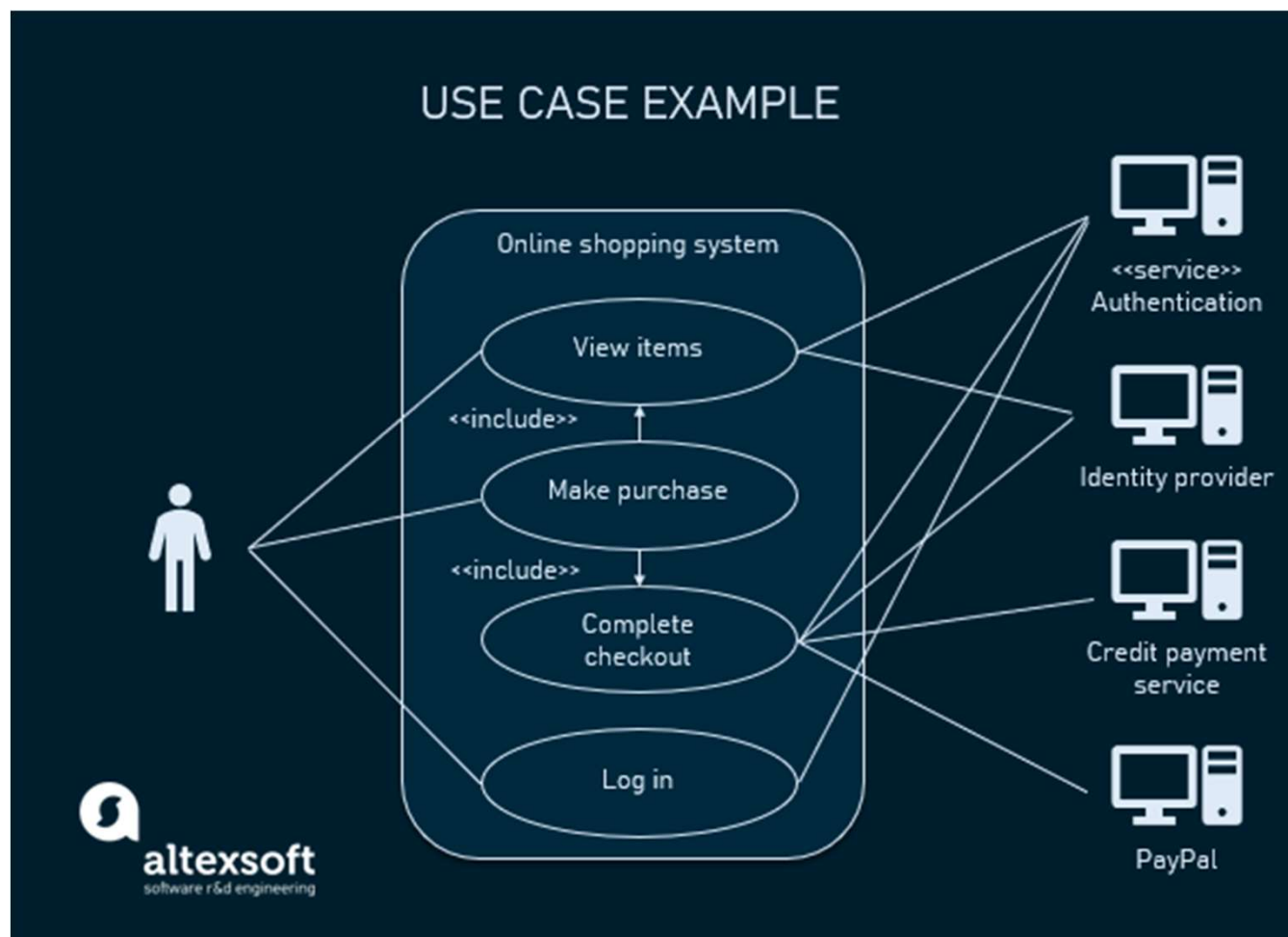


Засоби для створення діаграм

- ▶ Diagrams.net
- ▶ Lucid.app
- ▶ <https://plantuml.com/>
- ▶ <https://mermaid.js.org/intro/>
- ▶ <https://kroki.io/>

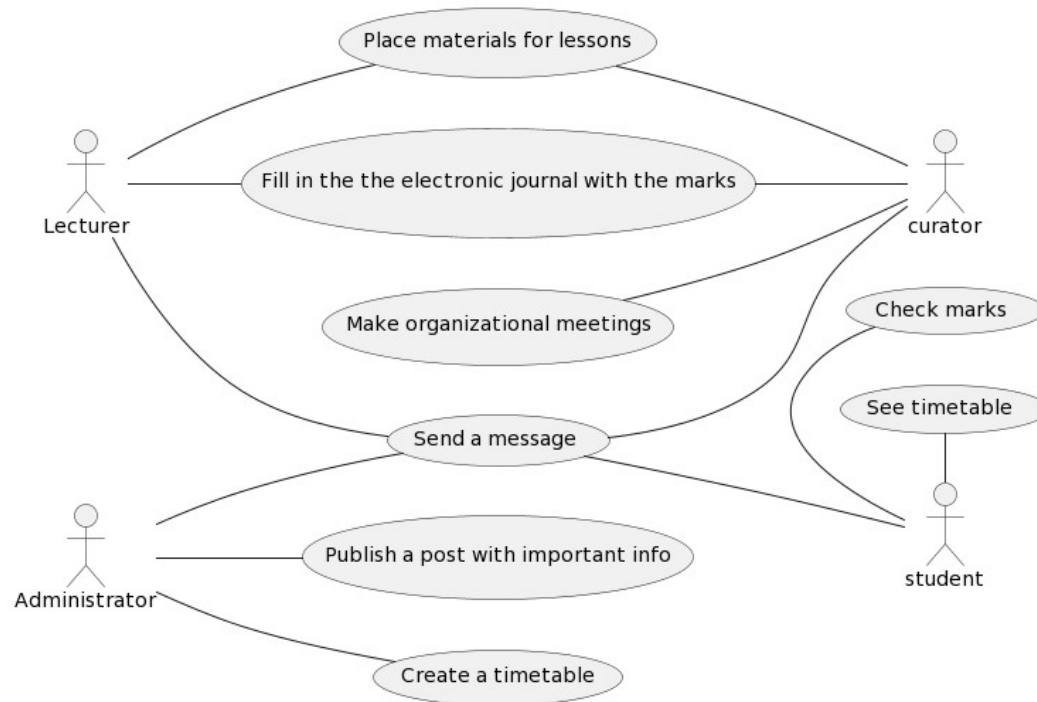


Приклад діаграми прецедентів/варіантів використання (use-case diagram)


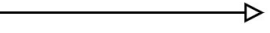


Use-case diagram

- ▶ **Діаграма варіантів використання** (англ. use-case diagram) - діаграма, що описує, як функціональні можливості програмної системи, що розробляється, доступний кожній групі користувачів.
- ▶ Для розглянемо спрощену діаграму для інформаційної системи університету
- ▶ У системі можна виділити такі групи користувачів:
 - Учні
 - Викладачі
 - Куратори
 - Адміністратори

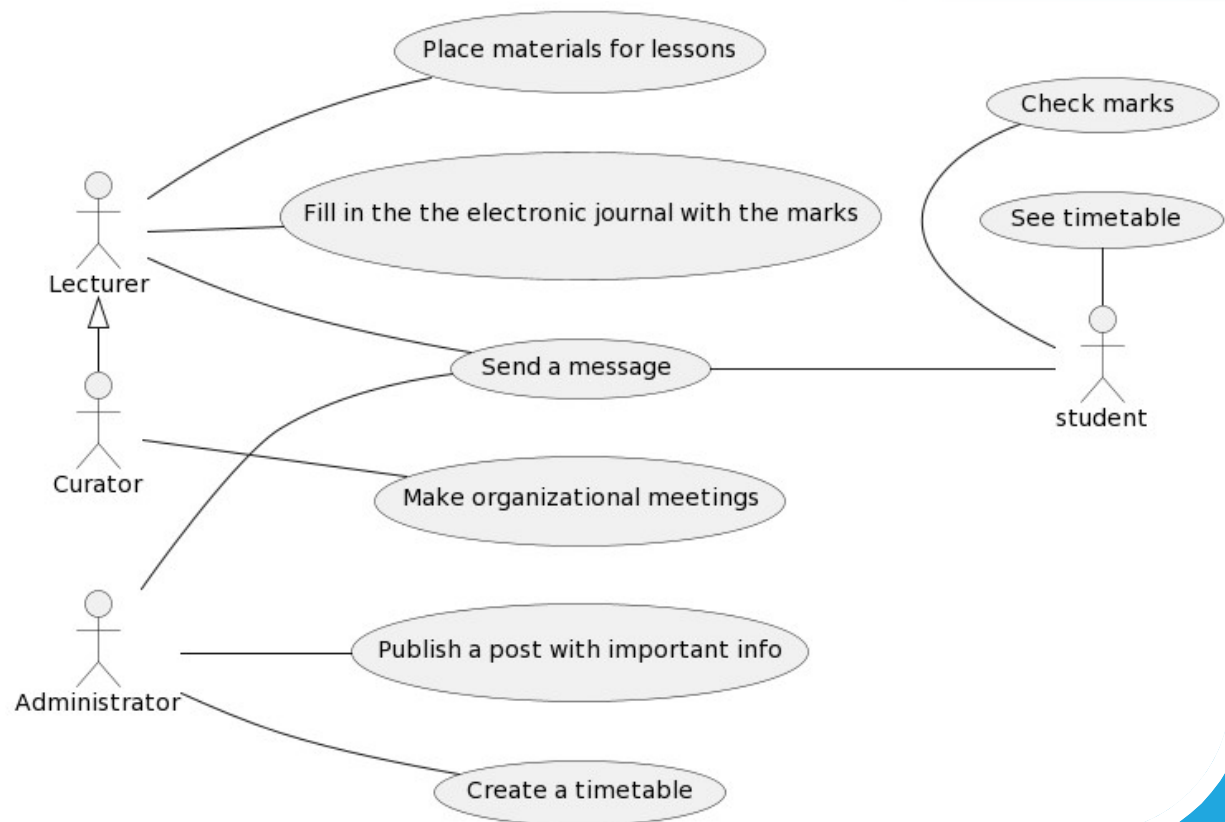


Позначення на Use-case diagram

Позначення	Опис
	Позначає можливість дії
	Відношення асоціації ("association relationship")
	Відношення узагальнення ("generalization relationship")
	Відношення включення ("include relationship")
	Відношення розширення ("extend relationship")
	Актор означає будь-яку сутність, що використовує систему.

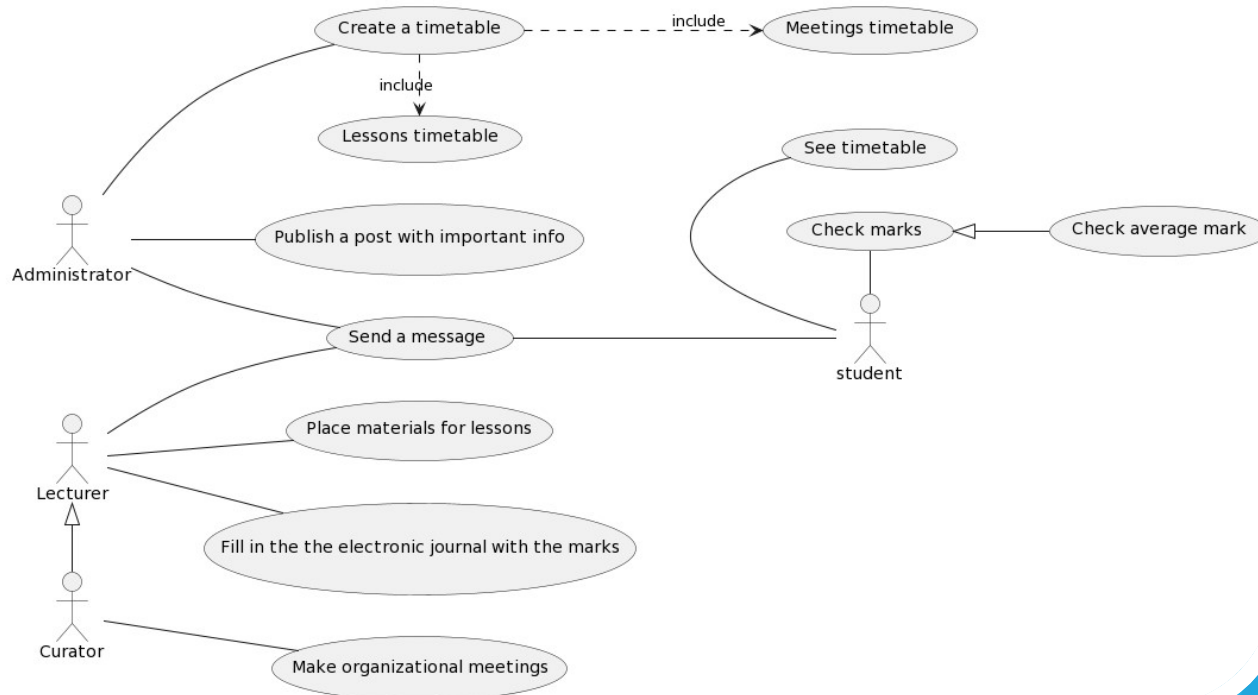
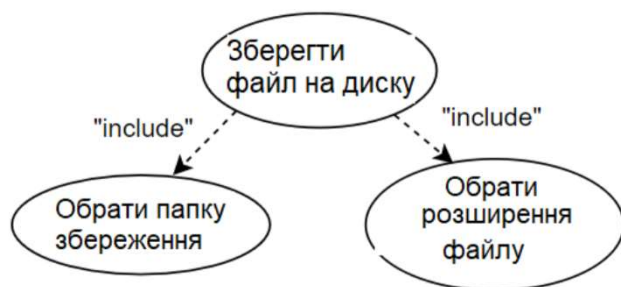
Відношення узагальнення

- ▶ Відношення узагальнення означає, що деякий актор (варіант використання) може бути узагальненням іншого актора (варіанта використання). Стрілка спрямована від окремого випадку (спеціалізації) до загального випадку.



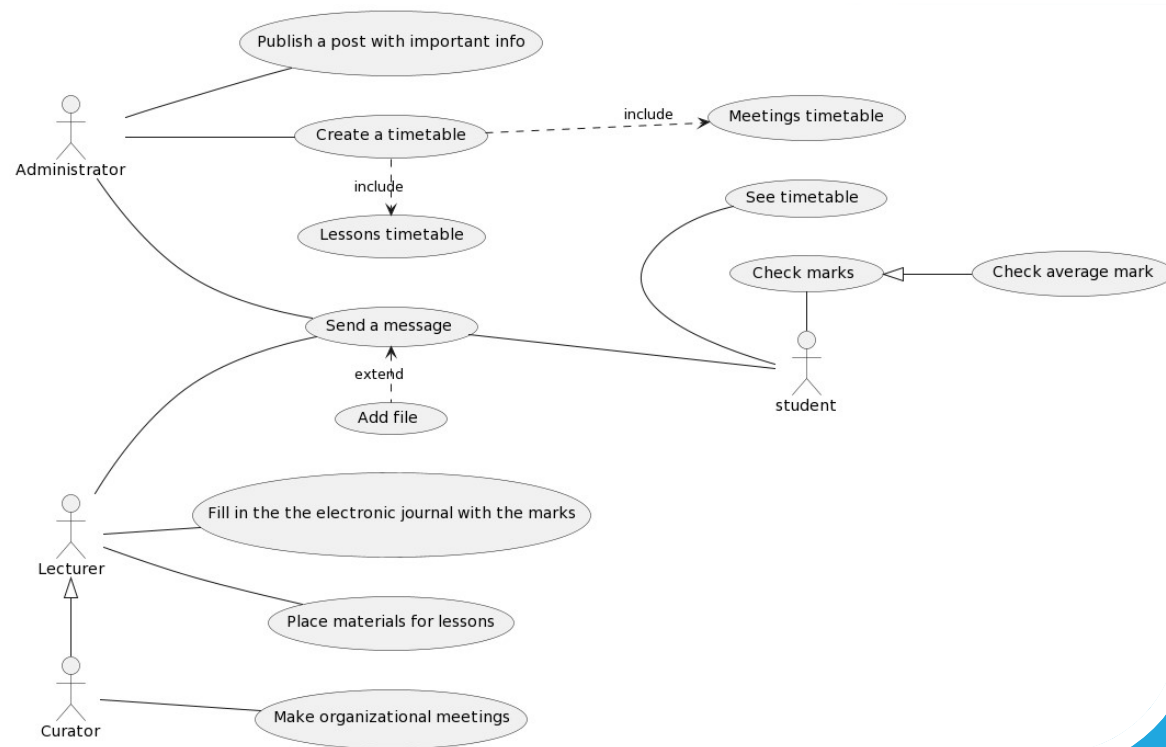
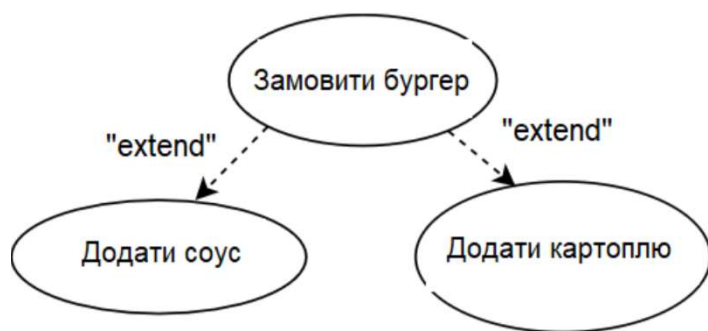
Відношення включення

- ▶ **Відношення включення** використовується, щоб показати, що деякий варіант використання включає інший варіант використання в якості складової частини.



Відношення розширення

- ▶ Якщо відношення включення означає, що елемент обов'язково включається до складу іншого елемента, то у разі відношення розширення це включення необов'язково.



Приклад use-case

Діючі особи	Користувач, Система
Мета	Користувач: авторизуватися в системі та почати працювати; Система: ідентифікувати користувача та його права.
Успішний сценарій: 1. Користувач запускає систему. Система відкриває сесію користувача, пропонує ввести логін та пароль. 2. Користувач вводить логін та пароль. 3. Система перевіряє логін та пароль. 4. Система створює запис в історії авторизацій (IP-адреса користувача, логін, дата, робоча станція). 5. Система видає користувачеві повідомлення щодо успішної авторизації (посилання на повідомлення).	
Результат	Користувач успішно авторизований і може працювати із системою.

Розширення	
*	Немає доступу до БД. Система видає повідомлення (<i>посилання на повідомлення</i>).Результат: користувач не може увійти.
1a	У налаштуваннях безпеки для цієї IP адреси існує заборона на вхід до системи.Результат: форма логіна не надається, система видає повідомлення користувачу (<i>посилання на повідомлення</i>).
2a	Користувач вибирає: «нагадати пароль». Викликається сценарій «нагадати пароль».
3a	Користувач із введеними логіном та паролем не знайдено.Результат: відмова в авторизації.Система видає повідомлення (<i>посилання повідомлення</i>).Перехід крок 2.
3б	Кількість невдалих спроб авторизуватися досягла максимального, встановленого в налаштуваннях.Результат: користувач не може увійти.Видається повідомлення: (<i>посилання повідомлення</i>).Вхід з IP-адреси Користувача заблоковано на час, встановлений у налаштуваннях.

Об'єктно-орієнтований аналіз

- ▶ Об'єктно-орієнтований підхід передбачає представлення ключових понять за допомогою об'єктів.
- ▶ Використовуючи об'єктно-орієнтований підхід у програмуванні, код залишається **організованим, гнучким і придатним для повторного використання**.
 - Об'єкти зберігають код упорядкованим, розміщуючи пов'язані деталі та конкретні функції в чітких місцях, які легко знайти. Деталі об'єктів залишаються пов'язаними з самими об'єктами.
 - Об'єкти зберігають код гнучким, тому деталі можна легко змінювати модульним способом в межах об'єкта, не впливаючи на решту коду.
 - Об'єкти дозволяють повторно використовувати код, оскільки вони зменшують обсяг коду, який необхідно створити, і зберігають простоту програм.

Об'єктно-орієнтований аналіз

- Об'єкти відносяться до наступних категорій:
- • **об'єкти сутності (entity objects)**, де початковий фокус під час проєктування зосереджено на проблемному просторі
- • **керуючі об'єкти (control objects)**, які отримують події та координують дії, коли процес переміщується до простору рішення
- • **граничні об'єкти (boundary objects)**, які з'єднують зовнішні служби з вашою системою.

Принципи об'єктно-орієнтованого проектування

- Абстракція
- Інкапсуляція
- Декомпозиція
- Узагальнення.



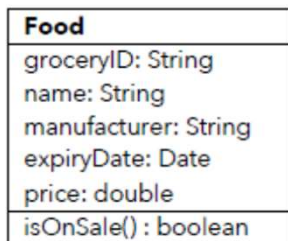
Abstraction

- ▶ **Абстракція** — теоретичний прийом дослідження, який дозволяє відсторонитися від деяких несуттєвих, у певному сенсі, властивостей досліджуваних явищ і виокремити суттєві та визначальні властивості.
- ▶ Наприклад, клас Student в програмі обліку студентів університету, крім загальних полів, таких як ім'я, прізвище, дата народження і т.д., міститиме поля, які відображають інформацію про номер залікової книжки, статус студента (дійсний, академічна відпустка, відраховано), факультет, номер його групи, оцінки за семестр і т. ін. Але для такого ж класа Student в програмі обліку студентів у тренінг-центрі з програмування така інформація буде неактуальна: клас міститиме поля, які відображають навчальний проєкт, на який був розподілений студент, рівень його англійської мови за результатами останнього тестування, кількість відвіданих заходів та ін.
- ▶ При використанні абстракції розробник фокусується на конкретних властивостях об'єкта залежно від тих задач, які повинен вирішувати об'єкт.

Abstraction summary

- ▶ Абстракція :
- ▶ Дозволяє краще зрозуміти концепцію, розбиваючи її на спрощений опис, який ігнорує неважливі деталі.
- ▶ Вилучає несуттєві деталі.
- ▶ Відноситься до певного контексту.
- ▶ Визначає деякі властивості об'єкта.
- ▶ Описує базову поведінку об'єкта.

UML Class Diagram:



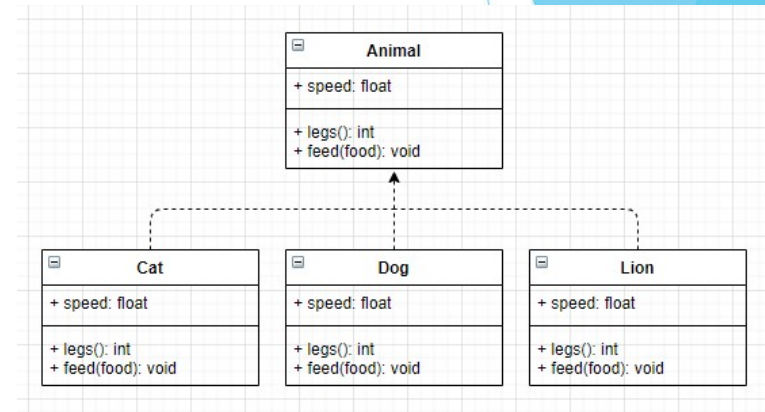
Назва класу

Властивості:

<variable name>:<variable type>

Дії

<name>(<parameter list>) : <return type>



Java code:

```
public class Food {
    public String groceryID;
    public String name;
    public String manufacturer;
    public Date expiryDate;
    public double price;

    public boolean isOnSale( Date date ) {
    }
}
```

Encapsulation

- ▶ **Інкапсуляція** при проєктуванні компонентів програми дозволяє реалізувати приховування внутрішніх даних компоненту і деталей його реалізації від інших компонентів програми та надання набору методів для взаємодії з ним (API).
- ▶ **Правильна інкапсуляція має велике значення з багатьох причин:**
- ▶ 1. Вона сприяє повторному використанню компонентів: оскільки в цьому випадку компоненти взаємодіють між собою лише через їх API і нечутливі до змін внутрішньої структури, вони можуть використовуватись в більш широкому контексті.
- ▶ 2. Інкапсуляція пришвидшує процес розробки: слабко пов'язані один з одним компоненти (тобто компоненти, чий код якомога менше звертається або використовує код інших компонентів) можуть розроблятися, тестуватися та доповнюватися незалежно.
- ▶ 3. Правильно інкапсульовані компоненти більш зрозумілі та легше налагоджуються, що спрощує підтримку програми.
- ▶ У мові Java, C# інкапсуляція реалізована за допомогою системи класів, які дозволяють зібрати інформацію про об'єкт в одному місці; пакетів, які групують класи по певному критерію, і модифікаторів доступу, якими можна позначити весь клас або його поле чи метод.

Encapsulation

► Існує чотири основні модифікатори доступу:

- + **public** - повний доступ до сутності (полю або методу класу) з будь-якого пакету;
- # **protected** - доступ до сутності лише для класів свого пакету і нащадків класу;
- - **private** - доступ тільки всередині класу, в якому оголошена сутність;
- **неявний модифікатор за замовчуванням** (за відсутності трьох явних) - доступ до сутності лише для класів свого пакету.

Encapsulation summary

- ▶ Збір значень атрибутів або даних або функцій, які маніпулюють цими значеннями разом у самодостатній об'єкт.
- ▶ Надання певних даних та функції об'єкта, до яких можна отримати доступ з інших об'єктів - забезпечення інтерфейсу.
- ▶ Можливість обмеження доступу до певних даних і функцій лише в межах цього об'єкта.
- ▶ Зменшення складності для користувача класу.
- ▶ Забезпечення модульності програмного забезпечення та полегшення роботи.
- ▶ Поведінка класів за моделлю «чорного ящика».
- ▶ Спрощення рефакторингу з правильною інкапсуляцією.

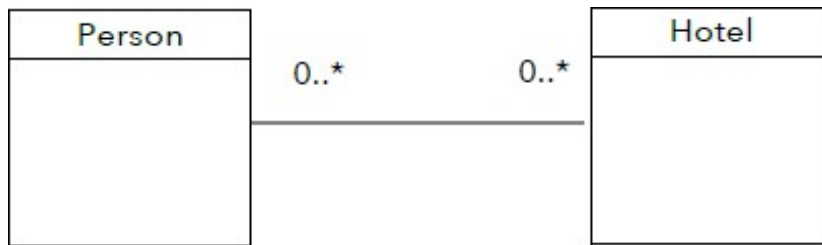
Decomposition

- ▶ **Декомпозиція:**
- ▶ Дозволяє у цілому виділити складові, які мають свої особливості.
- ▶ Дозволяє розбивати проблеми на частини, які легше зрозуміти та вирішити.
- ▶ Має 3 типи відносин:
 - Асоціація
 - Агрегація
 - Композиція



Association

- ▶ **Асоціація** - існує слабкий зв'язок між двома об'єктами. Ці об'єкти можуть деякий час взаємодіяти один з одним.
- ▶ Позначення у UML Class Diagram



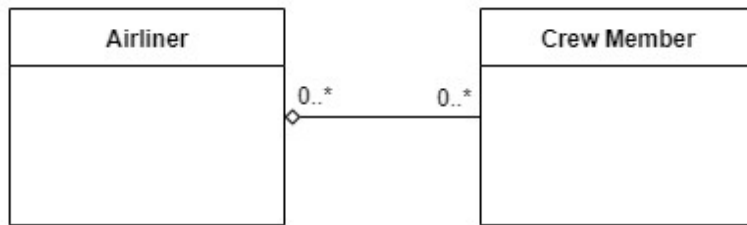
Person пов'язана з 0 або більше об'єктами Hotel, а Hotel може бути пов'язаний з 0 або більше Person

- ▶ Приклад у Java:

```
public class Student {
    public void play( Sport sport ){
        execute.play( sport );
    }
    ...
}
```

Aggregation

- **Агрегація** - це зв'язок «складається з», коли ціле має частини, які йому належать. У цьому відношенні може бути спільне використання частин між цілими. Від цілого до частин зв'язок вважається **слабким**. Це означає, що хоча частини можуть належати до цілого, вони також можуть існувати незалежно.
- Позначення у UML Class Diagram



Агрегація. Порожній ромб (символ агрегації) вказує на те, який об'єкт вважається цілим, а не частиною у відносинах.

- Приклад у Java:

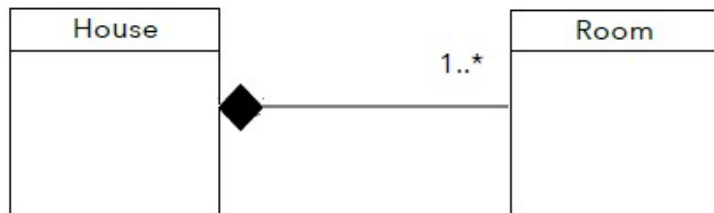
```
public class Airliner {
    private ArrayList<CrewMember> crew;

    public Airliner() {
        crew = new ArrayList<CrewMember>();
    }

    public void add( CrewMember crewMember ) {
        ...
    }
}
```


Composition

- ▶ **Композиція** - має сильний зв'язок «складається з». Це означає, що ціле не може існувати без своїх частин. Якщо руйнується ціле, то руйнуються і всі його частини. Зазвичай доступ до частин можливий лише через ціле.
- ▶ Позначення у UML Class Diagram



Композиція. Об'єкт House має 1 або більше об'єктів Room. Зафарбований ромб означає міцний зв'язок (складається з) і вказує на об'єкт, що береться за ціле. Два пов'язаних об'єкти не можуть існувати один без одного

- ▶ Приклад у Java:

```
public class House {
    private Room room;

    public House() {
        room = new Room();
    }
}
```

Generalisation

- ▶ **Принцип узагальнення** бере повторювані, загальні або спільні характеристики між двома або більше класами та виділяє їх в окремий клас, щоб код можна було використовувати повторно, а характеристики можна було успадковувати підкласами.
- ▶ Допомагає зменшити кількість надмірностей при вирішенні
- ▶ Прикладом узагальнення є успадкування, де можуть бути спільні характеристики в суперкласі. Наприклад: Тварина — це суперклас, а кішка/собака/білка — підкласи
- ▶ Код можна повторно використовувати, що означає, що нам не потрібно повторно впроваджувати вже існуючі поведінки.
- ▶ D.R.Y. (Не повторюйся).
- ▶ Допомагає створювати програмне забезпечення, яке легше розширювати та реалізовувати зміни.

Generalisation

► Приклад у Java

```
public abstract class Animal {
    protected int numberOfLegs;
    protected int numberOfTails;
    protected String name;

    public Animal( String petName, int legs, int
tails ) {
        this.name = petName;
        this.numberOfLegs = legs;
        this.numberOfTails = tails;
    }

    public void walk() { ... }
    public void run() { ... }
    public void eat() { ... }
}

public class Lion extends Animal {
    public Lion( String name, int legs, int tails
) {
        super( name, legs, tails );
    }

    public void roar() { ... }
}
```

Позначення у UML Class Diagram

