Лекція 17.
Відношення між типами.

FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

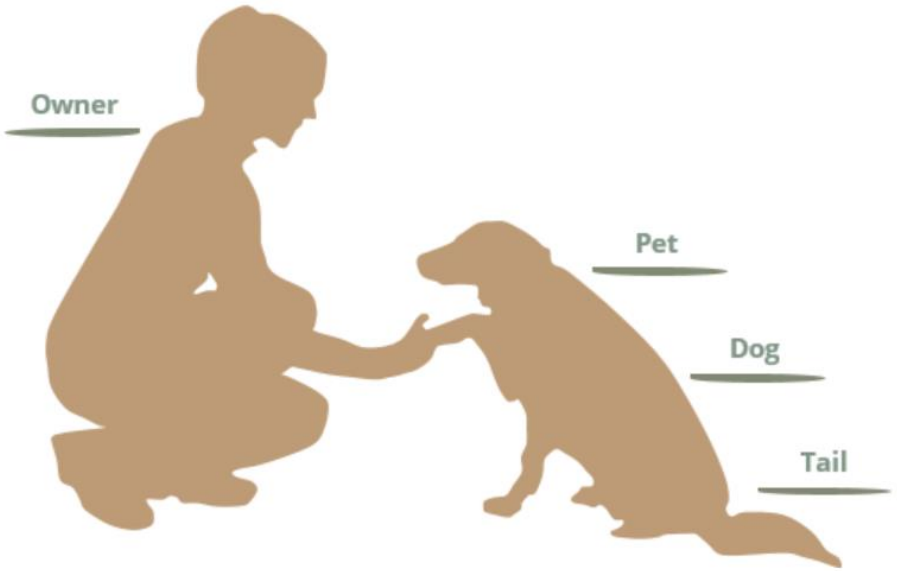
План на сьогодні

1. Композиція
2. Агрегація
3. Асоціація
4. Залежність
5. Принципи SOLID
6. Шаблон проектування "Стратегія"
7. UML

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Типи відношень між об’єктами

- **Наслідування** (“Є видом” - “Is-A”) - собака **“Є”** різновидом домашньої тварини.
- **Композиція** (“Містить, не може існувати без” - “Has-A”) - Хвіст є частиною собаки.
- **Агрегація** (“Має, але не керує”) – Факультет має студентів, але вони можуть перевестись на інший факультет.
- **Асоціація** (“Відношення між незалежними об’єктами” - “Uses-A”) - Власники годують домашніх тварин, а тварини радують власників.



Композиція

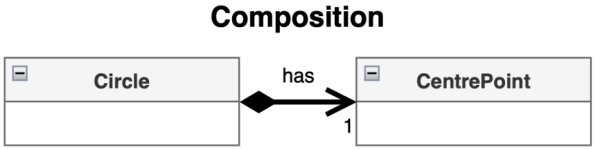
FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

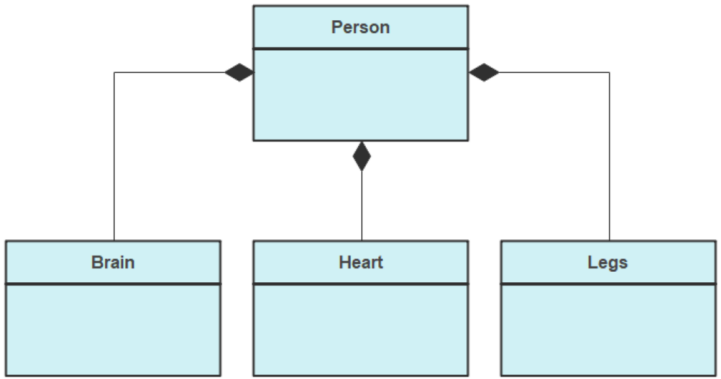
Композиція

Композиція – це механізм, який дозволяє створювати складні об'єкти з простіших частин, об’єднуючи їх у єдине ціле.

Ключові особливості:

- Один об'єкт **"має"** інший об'єкт.
- Частина **не може існувати самостійно** без цілого.
- Життєвий цикл частини **залежить** від життєвого циклу головного об'єкта.
- Частина **не знає** про об'єкт, якому вона належить.





Приклад з життя:

- Людина має серце.
- Автомобіль має двигун.
- Будинок має двері та вікна.

Приклад композиції

- Клас **Car** містить **Engine** як свою частину.
- Двигун не існує самостійно – він є частиною автомобіля.
- Коли **Car** створюється, **Engine** також створюється.
- Коли **Car** знищується, **Engine** теж знищується.

```
class Engine {
public:
    Engine() { cout << "Двигун створено\n"; }
    ~Engine() { cout << "Двигун знищено\n"; }
};
class Car {
private:
    Engine engine; // Композиція: двигун
належить автомобілю
public:
    Car() { cout << "Автомобіль створено\n"; }
    ~Car() { cout << "Автомобіль знищено\n"; }
};
```

Агрегація

Агрегація

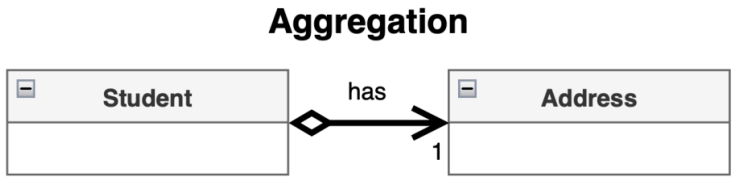
Агрегація – це підтип композиції, який описує відносини "частина-ціле", але з менш жорстким зв’язком між об'єктами.

Ключові особливості:

- Об’єкт **"має"** інший об'єкт, але **не керує** його життєвим циклом.
- Частина **може належати кільком** об’єктам одночасно.
- Частина **існує незалежно** від об'єкта-цілого.
- Частина **не знає** про існування об'єкта, якому вона належить.

Приклад з життя:

- Людина має адресу, але одна адреса може належати кільком людям.
- Відділ має працівників, але працівник може працювати у кількох відділах.



Приклад агрегації

- Об’єкт **Department** містить лише вказівник, але не керує життєвим циклом **Worker**.
- Коли **Department** знищується, **Worker** залишається в пам’яті, поки ми вручну не видалимо його.

```
class Worker {
public:
    std::string workerName;
    Worker(std::string name) : workerName(name) {}
};
class Department {
public:
    Worker* departmentWorker; // Агрегація: Відділ має вказівник на зовнішнього працівника
    Department(Worker* worker) : departmentWorker(worker) {}
};
```

Асоціація

Асоціація

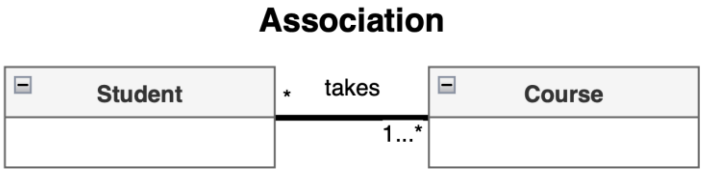
Асоціація – це відносини між двома незалежними об’єктами, які можуть взаємодіяти між собою, але не перебувають у відносинах "частина-ціле".

Ключові особливості:

- Об’єкти **не є частинами** один одного.
- Об’єкт **може належати кільком** іншим об’єктам одночасно.
- Об’єкт **існує незалежно** та не керується іншим об’єктом.
- Зв’язок може **може знати або не знати** про існування іншого об’єкта.

Приклад з життя:

- Лікар має пацієнтів, а пацієнти можуть звертатися до кількох лікарів.
- Викладач навчає студентів, а студенти можуть навчатися у кількох викладачів.



Приклад асоціації

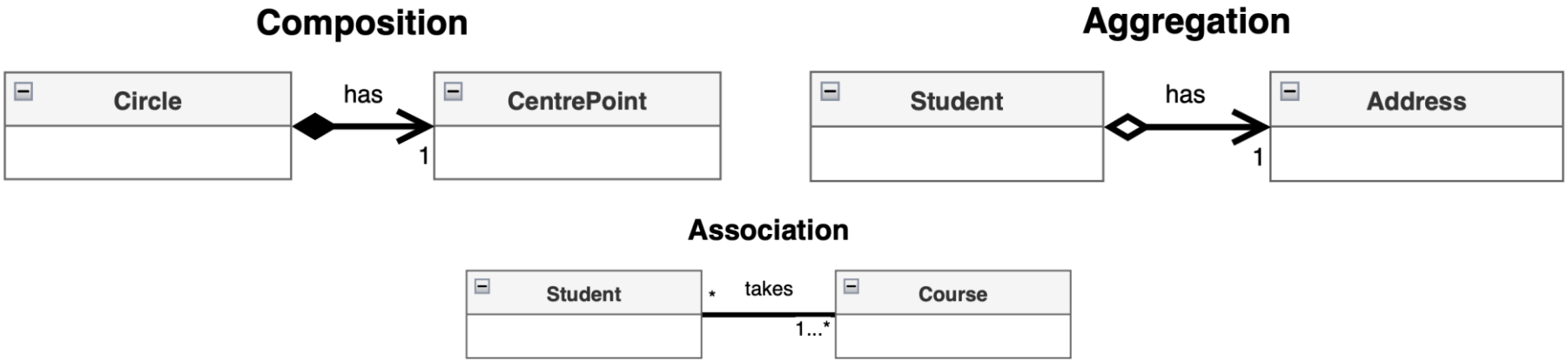
```
class Doctor; // Попереднє оголошення
class Patient {
public:
    string patientName;
    Doctor** doctors; // Динамічний масив вказівників лікарів
    int doctorCount;
    Patient(string name) : patientName(name), doctorCount(0) {
        doctors = new Doctor*[5]; // Виділяємо пам’ять для 5 вк. лікарів
    }

    void addDoctor(Doctor* doctor); // Додаємо лікаря
};
class Doctor {
public:
    string doctorName;
    Patient** patients; // Динамічний масив вказівників пацієнтів
    int patientCount;
    Doctor(string name) : doctorName(name), patientCount(0) {
        patients = new Patient*[5]; // Виділяємо пам’ять для 5 вк. пацієнтів
    }
    void addPatient(Patient* patient);
};
```

- Використовуються динамічні масиви (**Doctor**** у **Patient** та **Patient**** у **Doctor**), що дозволяє зберігати масив асоційованих об’єктів.
- При знищенні об'єкта масив очищується (delete[]), але самі об'єкти не видаляються автоматично, що підкреслює незалежність об'єктів у асоціації.
- Зв’язок створюється за допомогою методу **addPatient()** у **Doctor** та **addDoctor()** у **Patient**, що забезпечує двосторонню асоціацію.

Композиція vs. Агрегація vs. Асоціація

- **Композиція** – сильний зв’язок між класами. Об’єкт володіє частинами, і вони знищуються разом.
- **Агрегація** – слабший зв’язок, частини існують незалежно. Об’єкт може мати кілька посилань.
- **Асоціація** – просто зв’язок між об’єктами, які можуть взаємодіяти, але не є частинами один одного.



Порівняльна таблиця

Властивості	Композиція	Агрегація	Асоціація
Відносини	Частин-Цілого (частини є частиною цілого)	Частин-Цілого (але частини можуть існувати окремо)	Об’єкти не зв’язані між собою
Члени можуть належати відразу декільком класам	Ні	Так	Так
Існування членів керується класами	Так	Ні	Ні
Вид відносин	Односпрямовані	Односпрямовані	Односпрямовані або Двонаправлені
Тип відносин	“Частина чогось”	“Має”	“Використовує”

Залежність

Залежність

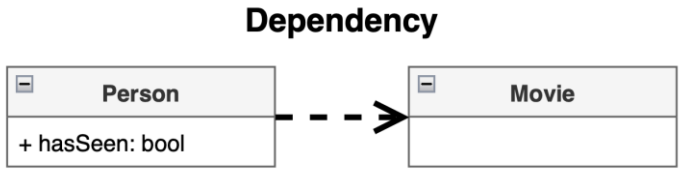
Залежність – це найслабший тип зв’язку між об’єктами, коли один об'єкт тимчасово використовує інший для виконання певного завдання.

Ключові особливості:

- Об'єкт не зберігає посилання чи вказівник на інший об’єкт.
- Використання іншого об’єкта відбувається тимчасово (наприклад, через параметри функції).
- Залежність завжди односпрямована – один об'єкт використовує інший, але не навпаки.

Приклад з життя:

- Людина залежить від автомобіля для поїздки, але автомобіль не залежить від конкретної людини.
- Квітка залежить від бджіл, щоб бути запиленою, але бджоли не залежать від конкретної квітки.



Приклад залежності

- **Report** не зберігає **Printer**, а лише отримує його у методі **generateReport().**
- **Printer** використовується лише на час виконання функції, тому зв’язок є тимчасовим.
- Клас **Printer** існує незалежно, і **Report** не керує його життєвим циклом.

```
class Printer {
public:
    void printMessage(const string& message) { // Залежність від рядка
        cout << "Printing: " << message << "\n";
    }
};
class Report {
public:
    void generateReport(Printer& printer) { // Передача Printer як залежності
        printer.printMessage("Report generated successfully!");
    }
};
```

Принцип єдиної відповідальності (SRP)

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Принцип відкритості/закритості

Принцип єдиної відповідальності (**Single Responsibility Principle, SRP**) – один із п’яти основних принципів **SOLID**.

Кожен клас повинен мати лише одну причину для зміни, тобто виконувати одну конкретну задачу. Це допомагає зробити код більш зрозумілим, модульним і легким для тестування, оскільки клас не буде виконувати кілька різних ролей.

Приклад порушення SRP

Уявімо, що ми створюємо клас Employee, який одночасно відповідає за зберігання інформації про працівника, а також за виведення цієї інформації на екран та збереження даних в базу даних. Це порушує принцип SRP, оскільки клас має кілька причин для зміни (зміна логіки представлення або зміна логіки зберігання даних).

Проблеми цього підходу:

- **Зміни в одному класі** можуть призвести до непередбачуваних наслідків у різних частинах програми. Якщо змінюється логіка збереження в базі даних, це може вплинути на виведення інформації.
- Клас має **три причини для зміни**: змінити інформацію про працівника, змінити спосіб збереження в базу даних і змінити спосіб виведення інформації.

Приклад дотримання SRP

Розділимо обов'язки класів, щоб кожен клас мав лише одну причину для зміни. Окремо створимо клас для зберігання даних в базу даних і клас для представлення інформації:

- **Клас Employee**: Тепер цей клас лише містить дані про працівника і має відповідні методи для доступу до цих даних. Його задача — лише зберігати інформацію.
- **Клас EmployeePrinter**: Цей клас відповідає лише за виведення інформації про працівника.
- **Клас EmployeeDatabase**: Цей клас відповідає лише за збереження даних про працівника в базу даних.

Переваги цього підходу:

- Якщо змінюється логіка виведення інформації про працівника, змінюється лише клас **EmployeePrinter**.
- Якщо змінюється логіка збереження в базу даних, змінюється лише клас **EmployeeDatabase**.

Принцип відкритості/закритості (OCP)

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Принцип відкритості/закритості

Принцип відкритості/закритості (Open Closed Principle, **OCP**) – один із п’яти основних принципів **SOLID**.

Програмні сутності (класи, модулі, функції) мають бути **відкритими для розширення**, але **закритими для змін**.

- **Закритість для змін** – не потрібно змінювати існуючий код при додаванні нового функціоналу.
- **Відкритість для розширення** – поведінку системи можна модифікувати шляхом додавання нових класів або функцій.

Принцип відкритості/закритості

Принцип відкритості/закритості Мейєра

- Бертран Мейєр вперше сформулював OCP у книзі *Object-Oriented Software Construction*.
- Використання **успадкування реалізації** – новий функціонал додається через підкласи.

Поліморфний принцип відкритості/закритості

- Запропонований Робертом Мартіном.
- Базується на **абстрактних інтерфейсах** – нові класи реалізують існуючі інтерфейси, не змінюючи базові класи.
- Використовується в **об'єктно-орієнтованому програмуванні** для досягнення гнучкості коду.

Приклад відкритості/закритості

- Додавання нового типу повідомлення (наприклад, Push-сповіщення) потребує **редагування методу** Send**()**.
- Це порушує принцип OCP, оскільки доводиться змінювати вже написаний код.

```
class Notification {
public:
    void Send(int type, const string& message) {
        if (type == 1) {
            cout << "Email: " << message << "\n";
        } else if (type == 2) {
            cout << "SMS: " << message << "\n";
        }
    }
};
```

Приклад відкритості/закритості

- Використовується **базовий клас** Notification із віртуальним методом Send().
- Додаючи новий клас (PushNotification), ми **не змінюємо** існуючий код, а лише розширюємо його.

```cpp
class Notification {
public:
    virtual void send() const = 0;  // Абстрактний метод
    virtual ~Notification() = default;
};

class EmailNotification : public Notification {
public:
    void send() const override {
        std::cout << "Sending Email notification!" << std::endl;
    }
};

class SMSNotification : public Notification {
public:
    void send() const override {
        std::cout << "Sending SMS notification!" << std::endl;
    }
};

class PushNotification : public Notification {
public:
    void send() const override {
        std::cout << "Sending Push notification!" << std::endl;
    }
};

void sendNotification(const Notification& notification){
    notification.send();
};
```

Принципи LSP та ISP

FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Принципи LSP та ISP

- **Принцип підстановки Ліскoв (Liskov Substitution Principle, LSP)** – один із п’яти основних принципів **SOLID**.
  Об'єкти похідного класу повинні замінювати об'єкти базового класу без зміни коректності роботи програми. Тобто підклас має розширювати поведінку батьківського класу, а не змінювати його суть.

- **Принцип Інтерфейсної сегрегації SOLID (Interface Segregation Principle, ISP)** говорить про те, що краще мати багато спеціалізованих інтерфейсів, ніж один великий інтерфейс з непотрібними методами.

Приклад порушення LSP та ISP

Уявімо, що у нас є базовий клас **Bird** (птах), і у нього є метод **fly()**. Ми також створюємо підкласи **Sparrow** (горобець) і **Penguin** (пінгвін). Але тут виникає проблема: не всі птахи можуть літати! Отже, якщо ми спробуємо викликати **fly()** у **Penguin**, це буде нелогічно:

Проблеми цього підходу:

- ❏ Порушення LSP та ISP: Метод **fly()** визначений у базовому класі **Bird**, але підклас **Penguin** змушений його перевизначати некоректно (викидати виняток).
- ❏ Використання Penguin замість **Bird** призводить до проблеми, що руйнує логіку програми (порушення LSP).

Приклад дотримання LSP та ISP

Щоб виправити ситуацію, розділимо поведінку птахів на **тих, хто літає** та **тих, хто не літає**.

- Bird тепер просто визначає базову поведінку птахів (наприклад, звуки).
- FlyingBird містить метод fly(), і тільки ті птахи, які можуть літати, наслідують його.
- SwimmingBird містить метод swim(), і тільки ті птахи, які можуть плавати, наслідують його.
- FlyingSwimmingBird - птахи які вміють літати та плавати.
- **Penguin** наслідує лише SwimmingBird, але не FlyingBird, бо **йому не потрібно мати метод fly()**. (Interface segregation principle)
- Функція makeBirdFly() приймає тільки FlyingBird, тому неможливо передати туди пінгвіна.
- Функція makeBirdSwim() приймає тільки SwimmingBird, тому неможливо передати туди горобця.
- Duck наслідує FlyingSwimmingBird оскільки вміє літати та плавати.

ДЗ: Намалювати UML діаграму згідно даного прикладу

Наслідування vs Агрегацію

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Наслідування vs Агрегація

- ❏ Дозволяє використовувати **поліморфізм** (Car* myCar = new ElectricCar();).
- ❏ Проста логічна структура, якщо Car справді залежить від типу двигуна.

```
class Car {
public:
    virtual void drive() = 0; // Абстрактний метод
};
class ElectricCar : public Car {
public:
    void drive() override {
        std::cout << "Їде на електродвигуні\n";
    }
};
class GasolineCar : public Car {
public:    void drive() override {
        std::cout << "Їде на бензиновому двигуні\n";
    }
};
```

Наслідування vs Агрегація

- **Гнучкість:** можна легко змінити двигун (Car може мати будь-який Engine).
- **Зменшена залежність:** Car не змінюється при додаванні нового типу двигуна.
- **Менше дублювання коду** у порівнянні з наслідуванням.

```
class Engine {
public:
    virtual void start() = 0;
    virtual ~Engine() {}
};
class ElectricEngine : public Engine {
public:
    void start() override {
        std::cout << "Запуск електродвигуна\n";
    }
};
class GasolineEngine : public Engine {
public:
    void start() override {
        std::cout << "Запуск бензинового двигуна\n";
    }
};
```

```
class Car {
private:
    Engine* engine; // Агрегація: Car "має" Engine
public:
    Car(Engine* eng) : engine(eng) {}
    void drive() {
        engine->start();
        std::cout << "Автомобіль рухається\n";
    }
};
```

Шаблон проектування "Стратегія"

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Шаблон проектування "Стратегія"

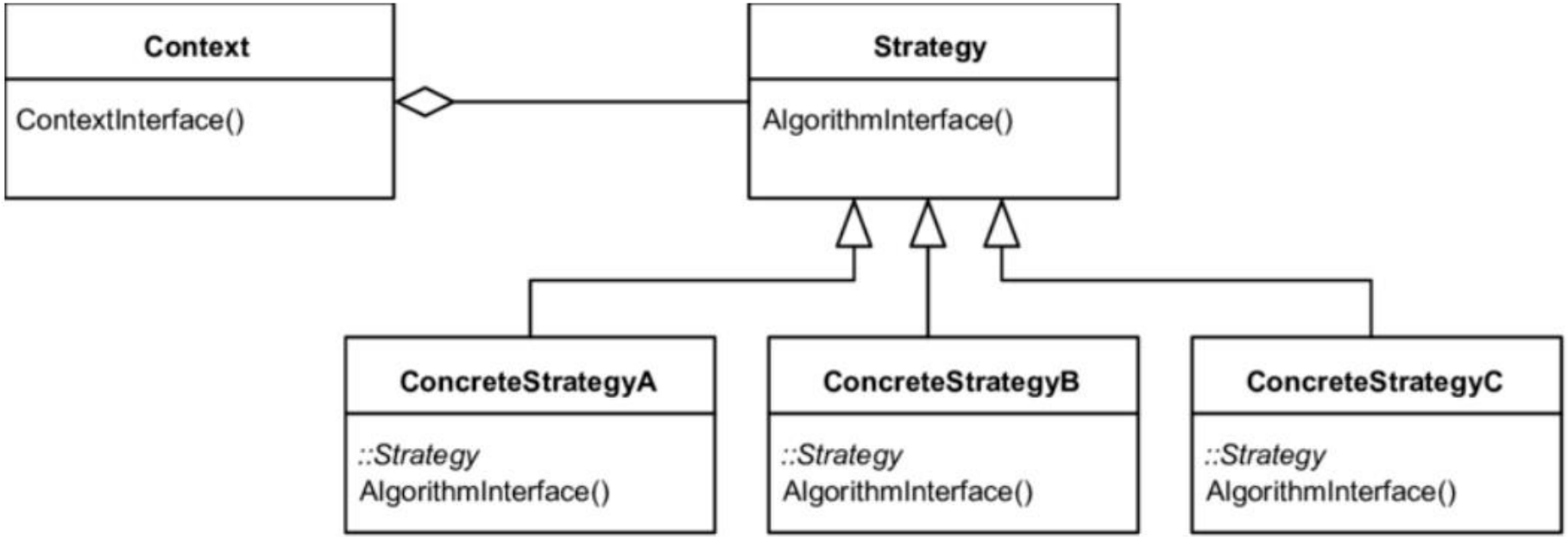
Стратегія – це поведінковий шаблон проектування, який дозволяє інкапсулювати алгоритми у вигляді окремих класів та зробити їх взаємозамінними.

- Контекст (Context) – об’єкт, який містить посилання на об’єкт стратегії.
- Стратегії (Strategies) – різні варіанти алгоритмів, які можуть бути підставлені у контекст.

Навіщо це потрібно?

- Змінювати поведінку об'єкта без зміни його коду.
- Уникати великої кількості умовних операторів (if-else, switch).
- Робити код більш гнучким і розширюваним.

Шаблон проектування "Стратегія"



Приклад "Стратегії"

```
// Інтерфейс стратегії
class IStrategy {
public:
    virtual void Execute(int a, int b) = 0;
    virtual ~IStrategy() {}
};
// Конкретна стратегія: додавання
class Addition : public IStrategy {
public:
    void Execute(int a, int b) override {
        cout << "Результат додавання: " << (a + b) << "\n";
    }
};
// Конкретна стратегія: множення
class Multiplication : public IStrategy {
public:
    void Execute(int a, int b) override {
        cout << "Результат множення: " << (a * b) << "\n";
    }
};
```

```
// Контекст, що використовує стратегію
class Context {
private:
    IStrategy* strategy;
public:
    Context(IStrategy* strategy = nullptr) : strategy(strategy) {}

    void SetStrategy(IStrategy* newStrategy) {
        strategy = newStrategy;
    }

    void ExecuteStrategy(int a, int b) {
        if (strategy) {
            strategy->Execute(a, b);
        }
    }
};
```

UML

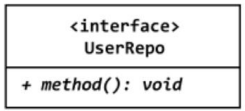
FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

UML

UML (Unified Modeling Language) — уніфікована мова моделювання, яка використовується для візуального представлення об'єктно-орієнтованих систем. **UML** допомагає розробникам аналізувати, проектувати та документувати програмне забезпечення.

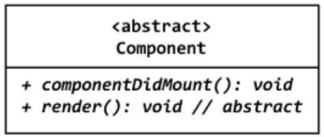
- Полегшує розуміння архітектури програмного забезпечення
- Сприяє стандартизації проектної документації
- Допомагає ефективно взаємодіяти між членами команди
- Підтримується багатьма інструментами моделювання

Основні елементи UML



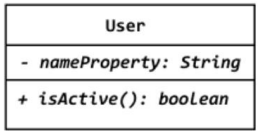
Interface

Interface name written underneath the <interface> annotation. Methods underneath.



Abstract class

Same as the interface shape. Abstract methods marked as abstract with comments or “abstract methodName(): returnType”.



Class

Properties or attributes sit at the top, methods or operations at the bottom + indicates public, - indicates private, and # indicates protected

These should be drawn vertically

Inheritance

B inherits from A. Creates an “is-a” relation-ship. A is a generalization.

Implementation/realization

B is a concrete implementation/realization of A.

Association

A and B call each other.

One way association

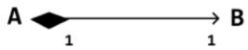
A can call B’s properties/methods, but not vice versa.

Aggregation

A has 1 or more instances of B. B can survive if A is disposed.

Ex: Professor (1) “has-many” classes (0..) to teach.*

Ex: Pond (0..1) “has-many” ducks (0..). Ducks can survive if the pond is destroyed.*



Composition

A has 1 or more instances of B. B cannot survive if A is disposed.

Ex: User (1) “has a” UserName (1). UserNames can’t exist as separate parts in away from a User in our application.

Дякую!

FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY