**Лабораторна робота №10-3. Розробка програм із використанням множинного наслідування**

**Мета:** Набуття навичок в розробці програм, де використовується множинне наслідування.

**Завдання**

1. Розробити програму, яка б демонструвала дії Пегасу — чарівного крилатого коня (див. лекцію ). Клас опису об’єкту Пегас отримати від двох базових класів - Кінь та Птах. Обидва класи Кінь та Птах відносяться до класу Тварини.
2. Мінімальні вимоги до членів класу Тварини, Кінь, Птах, Пегас: один член – ім’я.
3. Мінімальні вимоги до власних методів похідних класів: Кінь – методи: скакати, лежати; Птах – літати, сидіти; Пегас – рух, скакати, лежати, літати. Тварини можуть мати метод – рух. Метод має виражатися виведенням на консоль відповідного повідомлення щодо дій об’єкту.
4. В головній програмі створити об’єкти класів Кінь, Птах, Пегас, для яких послідовно викликаються відповідні методи. Перед викликом методу на консоль виводиться ім’я об’єкту.
5. В разі обрання свого підходу до демонстрації множинного наслідування за допомогою класу Пегас надайте відповідні пояснення.
6. Оформити однією програмою, до якої включити опис класу. В першому рядку програми та заголовкового файлу повинні бути записаними в коментарі номер групи та прізвище, а також номер ЛР
7. Продовжити працювати над програмою лабораторної роботи №11, закінчити попередні завдання з урахуванням наданих зауважень (для тих, хто не здав). Результати надсилати разом зі сформованими файлами.

Результати надсилати на електронну адресу викладача

[**t.i.lumpova@gmail.com**](mailto:t.i.lumpova@gmail.com)у вигляді заголовкового файлу та cpp-файлу з іменем у форматі

Для заголовкового файлу

**<Прізвище англійською>.h**

Наприклад, Ivanov.h

Для cpp-файлу:

**<Номер групи> <Номер лабораторної><Прізвище англійською>.cpp**

Наприклад, 31-01Ivanov.cpp.

Тему в заголовку листа записати

**ООП<Номер групи>-ЛР <Номер лабораторної>-<Прізвищеанглійською>**

**Строк відсилки ЛР ІПЗ-31 09.05.2024**

**ІПЗ-32 10.05.2024**

Всі запитання, що виникнуть, надсилайте на електронну адресу викладача, тему в заголовку листа записати

**ООП<Номер групи>-Запитання-<Прізвище англійською>**.

**Теоретичні відомості**

Адекватною метафорою для ілюстрації суті множинного спадкування є спадкування ознак і поводження батьків (матері і батька) дитиною в біологічній популяції. Як відомо, деякі біологічні ознаки, риси характеру, особливості поведінки дитина успадковує від батька, інші - від матері. Разом з тим вона може має і свої власні особливості, що з’явилися у результаті виховання, інших обставин. З погляду теорії, множинне спадкування є адекватним відображенням існування декількох незалежних класифікацій на множині різноманітних об'єктів, визначених в системі на етапі її аналізу.

Хоча множинне наслідування має ряд переваг над одинарним, але програмісти рідко його використовують. Основна проблема полягає в тому, що не всі компілятори С++ підтримують множинне наслідування. Рекомендується використовувати множинне наслідування, коли новий клас потребує функції та можливості більш ніж з одного базового класу. Оголошуйте базовий клас віртуальним, якщо він буде загальним для декількох похідних класів.

**Класи можливостей**

Проміжним рішенням між одиночним та множинним наслідуванням може бути використання класів можливостей. *Клас можливостей додає лише нові функціональні можливості, але не включає ніяких даних.* Методи класу можливостей передаються в похідні класи з допомогою звичайного наслідування.  Наприклад, клас Кінь походить від двох класів – Тварин та Зображення, при цьому останній не включає ніяких даних, а лише включає методи зображення об’єкту на екрані. В такому випадку всі наслідувальні дані класу Кінь походять лише від класу Тварини, а методи наслідуються від обох класів.

**Узагальнимо наші знання про класи**.

* Можливість використовувати готовий модуль як базу для власних розробок, не переробляючи його код називається повторним використанням коду.
* В основі механізму, що дозволяє створювати ієрархії класів, лежить принцип успадкування.
* Клас, що лежить в основі ієрархії, називається базовим.
* Класи, що успадковують властивості базового класу, називаються похідними.
* Похідні класи, у свою чергу, можуть бути базовими стосовно своїх спадкоємців, що в результаті приводить до ланцюжка успадкування. Процес утворення похідного класу на основі базового називається виводом класу. З одного базового класу можна вивести декілька похідних. Крім того, похідний клас може бути спадкоємцем декількох базових класів.
* Успадкування буває одиночним і множинним. При одиночному успадкуванні в кожного похідного класу є лише один базовий клас, а при множинному — декілька.
* Усі члени базового класу автоматично стають членами похідного. Керуючись оголошенням похідного класу, компілятор спочатку він бере усі властивості базового класу, а потім додає до них нові функціональні можливості похідного. Для цього використовується наступна синтаксична конструкція.

class ім'я\_похідного\_класу:специфікатор\_доступу ім'я\_базового\_класу

{

// тіло класу

};

* Хоча всі члени базового класу автоматично стають членами похідного класу, однак доступ до цих членів визначається видом успадкування. У залежності від специфікатора доступу, зазначеного при оголошенні похідного класу, розрізняють відкрите, закрите і захищене успадкування. За замовчуванням використовується закрите успадкування (специфікатор private).
* При відкритому успадкуванні всі відкриті і захищені члени базового класу стають відкритими і захищеними членами похідного класу відповідно.
* При захищеному успадкуванні всі відкриті і захищені члени базового класу стають захищеними членами похідного класу.
* При закритому успадкуванні всі відкриті і захищені члени базового класу стають закритими членами похідного класу відповідно.
* При успадкуванні рівень доступу до членів похідного класу може лише зменшитись. “Все можна ще більше закрити, але не відкрити”.
* Закриті члени базового класу є доступними лише членам базового класу. Успадкування не може суперечити принципу приховання даних.
* При створенні об'єктів похідного класу необхідно спочатку створити проміжний об'єкт базового класу. Отже, спочатку викликається конструктор базового класу, а потім — конструктор похідного класу. Якщо базових класів декілька, вони викликаються в порядку їх перерахування в списку наслідуваних класів.
* Деструктори викликається у порядку, зворотному відносно до конструкторів.
* Множинне успадкування часте приводить до неоднозначностей. Однак для вирішення неоднозначностей можна використати віртуальні базові класи. Для того щоб запобігти дублювання, у списку успадкування перед ім'ям базового класу варто вказати ключовому слову virtual, передбачивши конструктор за замовчуванням.
* Оскільки перевантаження функцій і операторів визначається в ході компіляції, такий вид поліморфізму називається статичним. (Усі процеси, виконувані компілятором, називаються статичними.) Процедура зв'язування типів з різними версіями функції на етапі компіляції називається раннім зв*'*язуванням. Однак ми хотіли б, щоб об'єкт реагував на контекст програми в ході її виконання. Це явище називається динамічним поліморфізмом, чи пізнім зв*'*язуванням.
* В основі механізму динамічного поліморфізму лежить механізм успадкування і віртуальних функцій. Віртуальна функція з'являється в базовому класі за допомогою ключового слова virtual. У похідних класах слово virtual можна не вживати. Програміст може перевизначити цю функцію в кожнім з похідних класів, настроївши її на рішення нової задачі. Якщо функція викликається з об'єкта відповідного класу, що входить в ієрархію, її поводження нічим не відрізняється від звичайної функції-члена. Віртуальність виявляється лише при виклику через вказівник.
* На противагу перевантаженню, при перевизначенні в похідних класах прототипи віртуальних функцій повинні точно збігатися з прототипом віртуальної функції в базовому класі. Такий процес називається заміщенням.
* На віртуальні функції накладаються наступні обмеження.
* Вони не можуть бути статичними.
* Вони не можуть бути дружніми.
* Конструктори не можуть бути віртуальними.
* Оскільки об'єкти похідного класу, по суті, являють собою модифіковані об'єкти базового класу, на них у мові С++ можна посилатися за допомогою вказівника на об*'*єкти базового класу. Якщо вони містять віртуальні функції, їх вибір буде виконаний на етапі виконання програми. Ієрархія класів, зв'язаних визначеною віртуальною функцією, називається поліморфічним кластером.
* Якщо віртуальна функція не заміщується у якомусь похідному класі, викликається її попередня заміщена версія. Отже, віртуальні функції утворять ієрархію*,* тобто успадковуються*.*
* Якщо віртуальна функція зовсім не заміщується у похідних класах — це помилка.
* Якщо базовий клас настільки абстрактний, що визначити заздалегідь, як повинна виглядати його віртуальна функція-член, неможливо, його конкретне втілення конкретизується лише в похідних класах на основі додаткової інформації. Такій класи називаються абстрактними. Для вирішення цієї проблеми в мові С++ реалізований механізм суто віртуальних функцій.
* Суто віртуальною функцією називається віртуальна функція-член базового класу, що не має визначення. Оголошення суто віртуальної функції виглядає в такий спосіб.

virtual тип ім*'*я*\_*функції(параметри) = 0;

* Об'єкт класу розміщається в суцільній області пам'яті, адреса якої зберігається в неявному вказівнику this. При виклику звичайної функції-члена цей вказівник передається їй як додатковий аргумент.
* Створюючи об'єкт похідного класу, компілятор поєднує його поля і поля базового класу в одне ціле. Якщо базовий клас містить віртуальні функції, їх адреси заносяться в таблицю віртуальних функцій. Усі класи, що утворюють поліморфічний кластер, містять вказівник на цю таблицю, у якій зберігаються вказівники на усі віртуальні функції-члени класів.
* Якщо вказівник базового класу посилається на об'єкт похідного класу, необхідно застосовувати віртуальний деструктор.
* Оператор dynamic\_cast має два операнди: у кутових дужках указується тип, а в круглих дужках — вказівник. Наприклад, в операторі

dynamic\_cast<TBase\*>(pObj)

TBase\* — тип, а pObj — вказівник.

* Якщо вказівник pObj посилається на об'єкт базового класу TBase чи похідного, результатом застосування оператора dynamic\_cast буде вказівник на базовий клас TBase.
* За допомогою оператора dynamic\_cast віртуальні базові класи можна перетворювати в похідні (понижуюче приведення), похідні — у базовий (підвищувальне приведення), а також один похідний клас — в іншій. До класів, що не містять віртуальних функцій, оператор dynamic\_cast не застосовується.
* Оператор typeid повертає посилання на об'єкт класу type\_info, визначеного в заголовку <typeinfo>. Клас typeinfo, поряд з ім'ям типу, містить визначення операцій порівняння, тому його можна використовувати в логічних вираженнях.

**Агрегація та композиція**

Асоціація – це коли один клас включає в себе інший клас в якості одного з полів. Асоціація описується словом «має» (has). Автомобіль має двигун, відповідно він не буде нащадком двигуна (хоча така архітектура теж можлива в окремих випадках).

Виділяють два окремих випадки асоціації: композицію та агрегацію.

Композиція – це коли двигун не існує окремо від автомобіля. Він створюється при створенні автомобіля и повністю керується автомобілем. В типовому прикладі, екземпляр двигуна буде створюватися в конструкторі автомобіля.

class Engine

{

int power;

public: Engine(int p)

{

power = p;

}

}

class Car

{

string model = "Porshe";

Engine engine;

public: Car()

{

this.engine = new Engine(360);

}

}

```

Агрегація – це коли екземпляр двигуна буде створюватися десь в іншому місті коду, и передаватися в конструктор автомобіля в якості параметра.

class Engine

{

int power;

public: Engine(int p)

{

power = p;

}

}

class Car

{

string model = "Porshe";

Engine engine;

public: Car(Engine someEngine)

{

this.engine = someEngine;

}

}

Engine goodEngine = new Engine(360);

Car porshe = new Car(goodEngine);

### Спадкування vs Композиція vs Агрегація

Асоціація використовується на ранніх етапах дизайну, щоб показати, що існує залежність між класами.

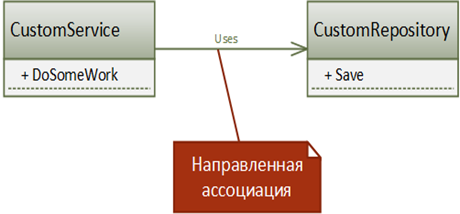
[](http://lh6.ggpht.com/-m4ZAymMANaE/ULzqPNMdWmI/AAAAAAAAD1E/MCznN_aFibU/s1600-h/image%5B45%5D.png)

Рисунок 1. Відношення асоціації

Відношення відкритого спадкування (IS A Relationship) говорить, що все, що справедливо для базового класу справедливо і для його нащадка. За допомогою цього відношення ми отримуємо поліморфну поведінку, абстрагуємося від конкретної реалізації класів, працюючи лише с абстракціями (інтерфейсами або базовими класами), не звертаючи уваги на деталі реалізації.

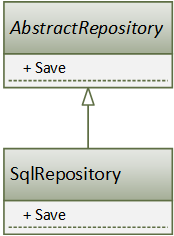
[](http://lh6.ggpht.com/-_HKlZxN-ZNY/ULzqRP2d1kI/AAAAAAAAD1Q/VhPv1s1QkxA/s1600-h/image%5B15%5D.png)

Рисунок 2. Відношення наслідування

Але: 1) не всі відношення між класами визначаються відношення «is a»,

2) наслідування є самим сильним зв’язком між двома класами, яку неможливо розірвати під час виконання (це відношення є статичним та, в строготипизованих мовах визначається під час компіляції).

Тут можна застосувати пару: **композиція** (composition) и **агрегація** (aggregation). Вони моделюють відношення «є частиною» (HAS-A Relationship). Клас цілого містить поля (властивості) своїх складових.

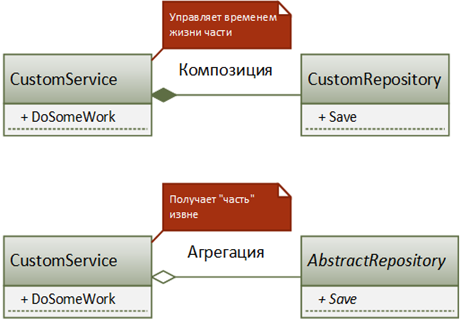
[](http://lh5.ggpht.com/-AXVlmpOuyYc/ULzqShNWgVI/AAAAAAAAD1g/fkMykO8SHYg/s1600-h/image%5B32%5D.png)

Рисунок 3. Відношення композиції та агрегації

**HINT**  
Щоб легше запам’ятати візуальну нотацію: (1) *ромбик завжди знаходиться з боку цілого, а проста лінія з боку складової частини*; (2) *замальований ромб означає більш сильний зв’язок – композицію, незамальований ромб показує більш слабкий зв’язок – агрегацію.*

Різниця між композицією та агрегацією полягає в тому, що **у випадку композиції** **ціле явно контролює час життя своєї складової частини** (частина не існує без цілого), а **у випадку агрегації ціле хоч і містить свою складову, їх час життя не зв’язаний** (наприклад, складова передається через параметри конструктора).

class CompositeCustomService

{  
    // Композиция

    private readonly CustomRepository \_repository

          = new CustomRepository();

    public void DoSomething()

    {

        // Используем \_repository

    }

}

class AggregatedCustomService

{   
    // Агрегация

    private readonly AbstractRepository \_repository;

    public AggregatedCustomService(AbstractRepository repository)   
    {

        \_repository = repository;

    }

    public void DoSomething()

    {

        // Используем \_repository

    }

}

**CompositeCustomService** для керування своїми складовими використовує композицію, а **AggregatedCustomService** – агрегацію. Явний контроль часу життя приводить до більш високої зв’язності між цілим та частиною, оскільки використовується конкретний тип, який тісно зв’язує учасників між собою.

Такий жорсткий зв’язок є доцільним, коли залежність є стабільною. Але ми можемо використовувати композицію та контролювати час життя об’єкта, не зав’язуючись на конкретні типи. Наприклад, за допомогою абстрактної фабрики:

internal interface IRepositoryFactory

{   
    AbstractRepository Create();

}

class CustomService

{   
    // Композиция

    private readonly IRepositoryFactory \_repositoryFactory;

    public CustomService(IRepositoryFactory repositoryFactory)

    {        \_repositoryFactory = repositoryFactory;

    }

    public void DoSomething()

    {

        var repository = \_repositoryFactory.Create();

        // Используем созданный AbstractRepository

    }

}

В даному випадку ми не позбавляємося композиції (**CustomService** все ще контролює час життя **AbstractRepository**), але робить це не напряму, а за додаткової абстракції – абстрактної фабрики. Такий підхід вимагає подвоєння кількості класів наших залежностей, тому його доцільно використовувати, коли явний контроль часу життя є необхідною умовою.

Логічність використання різних відношень між класами залежить від точки зору проектувальника, одну задачу можна вирішити декількома способами: можемо отримати сильно зв’язаний дизайн з великою кількістю наслідування та композиції, а в іншому випадку – ця ж задача буде розбита на більш автономні будівельні блоки, з’єднані між собою за допомогою агрегації.

Наприклад, нашу задачу с сервісами и репозитаріями можна рішити різними способами. Можна зробити **SqlCustomService** нащадком від **AbstractCustomService**; а можна сказати, що такий підхід невірний, оскільки **CustomService** у нас один, а ієрархія повинна бути у репозитарієв.

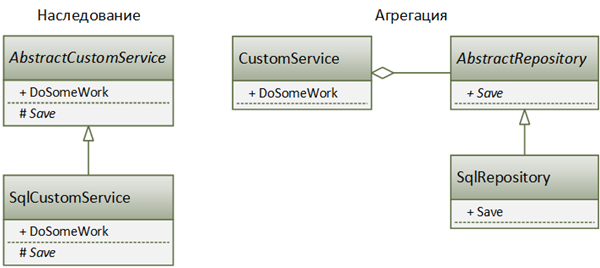
[](http://lh5.ggpht.com/-pHLT2gHv-Fo/ULzqUrZaj3I/AAAAAAAAD10/uXmCvfefbbQ/s1600-h/image%5B42%5D.png)

Рисунок 4. Наслідування vs Агрегація

Кожний варіант приводить до одного і того ж кінцевого результату, при цьому зв’язність змінюється від дуже високої (при наслідуванні) до дуже слабкої (при агрегації).