9.1.3. Зв’язки між класами в об’єктно-орієнтованому програмуванні: асоціація, агрегація, композиція, спадкування, залежність, реалізація

У об’єктно-орієнтованому програмуванні існує дві ієрархії: класів та об’єктів.

За допомогою об’єктно-орієнтованого програмування є можливість описати різні об’єкти, їх характеристики та властивості. Об’єкти описуються за допомогою класів. Об’єкт це сутність зі своїми властивостями та характеристиками, яка може реагувати на повідомлення за допомогою своїх існуючих даних та з’являється при створення екземпляра класу або прототипу. Клас – це список характеристик та можливостей об’єкта. Наприклад, людина – це клас: у неї одна голова, дві руки, дві ноги, вона вміє ходити, розмовляти, бігати. А дівчина Олена це об’єкт класу людина, вона та інші люди схожі один до одного тільки тому що їх описує один загальний клас – людина. Але при цьому об’єкти відрізняються один від одного, тому що з впливом часу вони приймають унікальні риси. Класи та об’єкти це зовсім різні поняття, але дуже тісно зв’язані між собою. Припустимо, нам в програмі необхідно працювати з країнами. Країна – це абстрактне поняття. У неї є такі характеристики, як назва, населення, площа, прапор та інше. Для опису такої країни буде використовуватися клас з відповідними полями даних. Такі країни, як Росія і Україна будуть вже об’єктами (конкретними представниками типу країна).

У об’єктно-орієнтованому програмуванні існують таки зв’язки (відношення) між класами та об’єктами:

- асоціація;
- успадкування;
- агрегація;
- використання;
- інстанціювання;
- метаклас.

Успадкування класів

Успадкування класів – це дуже потужна можливість, яка дозволяє створювати похідні класи. Один клас може успадковувати властивості та характеристики іншого класу. При цьому клас який успадковує ці властивості називається дочірнім (похідним), а клас, який саме містить ці властивості – батьківським (базовим). Таким чином буде зекономлено маса часу на написання та налагодження коду нової програми. Об’єкти похідного класу вільно можуть використовувати все, що створено і налагоджено в базовому класі. При цьому, є можливість в похідний клас дописати необхідний код для

удосконалення програми: додати нові елементи, методи тощо. Базовий клас залишиться недоторканим. Наприклад, у нас є клас автомобіль, в описанні вказано, що в нього чотири колеса, двигун, гальмо тощо. У нього ж є два дочірніх класи – це легковий автомобіль та вантажівка, вони успадкували базові властивості батьківського класу, але при цьому придбали свої унікальні риси. Також прикладом може буди два незалежних класи роботи та автомобілі, що успадкували свої властивості від класу технологічні пристрої, але кожен з похідних класів буде мати свої особливості. Автомобілі без управління людиною функціонувати не можуть, коли роботам втручання людини в управління не потрібно.

В об'єктно-орієнтованому програмуванні **асоціація** (англ. association) визначає зв'язок між класами об'єктів, який дозволяє одному екземпляру об'єкта змусити інший виконувати дію від його імені. Цей зв'язок є структурним, оскільки він визначає, що об'єкти одного виду пов'язані з об'єктами іншого і не представляють поведінку.

У загальних термінах причинно-наслідковий зв'язок зазвичай називається «надсиланням повідомлення», «викликом методу» або «викликом функції-члена» до керованого об'єкта. Конкретна реалізація зазвичай вимагає, щоб запитуючий об'єкт викликав метод або функцію-член, використовуючи посилання або вказівник на розташування пам'яті керованого об'єкта.

Вважається, що об'єкти, пов'язані через асоціацію, виконують роль стосовно асоціації, якщо поточний стан об'єкта в активній ситуації дозволяє іншим пов'язаним об'єктам використовувати об'єкт у спосіб, визначений роллю. Роль може бути використана для розрізнення двох об'єктів одного класу при описі його використання в контексті асоціації. Роль описує загальнодоступні аспекти об’єкта щодо асоціації.

Кінці об'єднання можуть мати всі характеристики власності:

- можуть мати кратність, виражену нижньою та верхньою межею у вигляді «lowerlimit..upperlimit»;
- можуть мати ім'я;
- можуть оголосити видимість;
- можуть вказати, чи є кінець асоціації впорядкованим та/або унікальним.

Агрегація у програмуванні – це вид композиції.

Композиція у програмуванні або ж Об'єктна композиція, також Агрегація та включення – це створення об'єктів існуючих класів як елементів інших класів. Про композицію також часто говорять як про «відношення приналежності» за принципом у «у машини є корпус, колеса і двигун». Агрегація або включення організоване за принципом «у машини є модель».

Вкладені об'єкти нового класу зазвичай оголошуються закритими, що робить їх недоступними для прикладних програмістів, що працюють з класом. З іншого боку, творець класу може змінювати ці об'єкти, не порушуючи роботи існуючого клієнтського коду. Крім того, заміна вкладених об'єктів на стадії виконання програми дозволяє динамічно змінювати її поведінку. Механізм успадкування такої гнучкості не має, оскільки для похідних класів встановлюються обмеження, що перевіряються на стадії компіляції.

На відміну від успадкування, в композиції тип відносин є Has-a тобто має (машина має двигун). В наслідуванні ж тип відносин між породженим об'єктом і батьківським є Is-a зв'язком, тобто якщо об'єкт кішка породжено від тварина, то кішка є тварина (cat is a pet).

Різниця між агрегацією і композицією полягає в тому, що у зв'язку композиція життя об'єкта контролюється його «володарем», а під час агрегації – не контролюється.

Приклади коду:

```
// Composition
class Car
{
  private:
    Carburetor* itsCarb;
  public:
    Car() {itsCarb=new Carburetor();}
    virtual ~Car() {delete itsCarb;}
};
```

```
// Aggregation
class Pond
{
  private:
    vector<Duck*> itsDucks;
};
```

Композиція та агрегування.

Класи та об'єкти можуть бути пов'язані один з одним. Спадкування описує зв'язок «є» (або англійською «IS A»). Лев є Тваринним. Таке ставлення легко висловити за допомогою успадкування, де Animalбуде батьківським класом, а Lionнащадком. Однак не всі зв'язки в світі описуються таким чином. Наприклад, клавіатура безумовно якось пов'язана з комп'ютером, але вона не є комп'ютером . Руки якось пов'язані з людиною, але вони не є людиною. У цих випадках в його основі лежить інший тип відношення: не є, а є частиною (HAS A). Рука не є людиною, але є частиною людини. Клавіатура не є комп'ютером, але є частиною комп'ютера. Відносини HAS A можна описати в коді, використовуючи механізми композиції таагрегування .

Різниця між ними полягає у «суворості» цих зв'язків. Наведемо простий приклад: У нас є наша Carмашина. Кожна машина має двигун. Крім того, кожна машина має пасажирів усередині. У чому принципова різниця між полями Engine enginei Passenger [] passengers? Якщо у машини всередині сидить пасажир A, це не означає, що в ній не можуть перебувати Впасажири C. Одна машина може відповідати кільком пасажирам.

Крім того, якщо всіх пасажирів висадити із машини, вона продовжить спокійно функціонувати. Зв'язок між класом Carта масивом пасажирів Passenger [] passengersменш суворий. Вона називається агрегацією . Є непогана стаття з цієї теми: Відносини між класами (об'єктами). У ній наведено ще один добрий приклад агрегації. Припустимо, у нас є клас Student, який позначає студента, та клас StudentsGroup(група студентів). Студент може входити і до клубу любителів фізики, і до студентського фан-клубу «Зоряних війн» або команди КВК.

Композиція - суворіший тип зв'язку. При використанні композиції об'єкт не тільки є частиною якогось об'єкта, але й не може належати до іншого об'єкта того ж типу. Найпростіший приклад – двигун автомобіля. Двигун є частиною автомобіля, але може бути частиною іншого автомобіля. Як бачиш, їх зв'язок набагато суворіший, ніж у Cari Passengers.

Виділяють два окремі випадки асоціації: композицію та агрегацію.

Композиція – це коли двигун не існує окремо від автомобіля. Він створюється під час створення автомобіля і повністю керується автомобілем. У 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 створюється десь в іншому місці коду, і передається в конструктор автомобіля як параметр.

```
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);

Зв'язність

Зв'язність (англ. coupling) чи залежність (англ. dependency) – це міра, у якій модуль (компонент) програми залежить від кожного іншого модуля (використовує якусь інформацію про нього).

Зв'язність зазвичай протиставляється пов'язаності. Метрики програмного забезпечення зв'язність та пов'язаність, винайдені Ларрі Констянтином, першим розробником структурного проєктування,[1] який також був першим їхнім прихильником (див. також SSADM). Слабка зв'язність часто є ознакою добре структурованої

комп'ютерної системи, та гарної архітектури, і в поєднанні з високою пов'язаністю дозволяє досягнути гарної прочитності та підтримуваності коду.

Зв'язність може бути «слабкою» (чи «низькою») або «сильною» («високою»).

Процедурне програмування:

Зв'язність за вмістом (сильна). З'являється, коли один модуль модифікує або залежить від внутрішнього вмісту іншого модуля (наприклад, використовує його змінні). Тому зміна способу, яким другий модуль обробляє дані, вимагатиме зміни залежного модуля.

Зв'язність за спільністю даних. Коли два модулі мають спільні глобальні дані (глобальні змінні). Зміна спільного ресурсу передбачає зміну всіх модулів що його використовують.

Зовнішня зв'язність. З'являється коли два модулі поділяють нав'язаний ззовні формат даних, протокол комунікації чи інтерфейс пристрою. Зазвичай це пов'язане з взаємодією з зовнішніми інструментами чи апаратурою.

Зв'язність контролю. З'являється коли один модуль контролює хід роботи іншого, передаючи йому інформацію про те що робити.

Залежність-штамп (Залежність від структурованих даних) - коли модулі мають спільну складну структуру даних, і використовують лише її частини, можливо різні (наприклад функції передається запис, хоча вона потребує лише його частину).

Зв'язність даних виникає коли модулі діляться спільними даними через, наприклад, параметри. Кожне дане є елементарним, і єдиним яке ділиться (наприклад передача числа функції що обчислює квадратний корінь).

Зв'язність через повідомлення (слабка). Найслабший тип зв'язності. Досягається за допомогою децентралізації стану (в об'єктах). Взаємодія компонентів проводиться через параметри та обмін повідомленнями.

Відсутня. Модулі взагалі не взаємодіють між собою.

Об'єктно-орієнтоване програмування:

Зв'язність підкласу. Описує взаємозв'язок між предком та нащадком. Нащадок прив'язаний до предка, а предок ні.

Тимчасова зв'язність. Коли дві дії упаковані в один модуль лише тому, що вони можуть відбуваються одночасно.

У недавній праці інші концепції зв'язності були вивчені та використані як ознаки різних принципів модульності, що використовуються на практиці.

Недоліки. Сильно зв'язні системи зазвичай демонструють наступні характеристики розробки, які часто розглядаються як **недоліки**:

- зміна одного модуля зазвичай викликає хвилю змін в інших модулях;
- збирання модулів до купи та стиковка вимагатиме більше зусиль та часу, через збільшення міжмодульних залежностей;
- конкретний модуль важче буде повторно використати та/або тестувати, тому що потрібно включити залежні модулі.

Проблеми продуктивності:

Незалежно від сили зв'язності, продуктивність системи зменшується через створення повідомлень та параметрів, їх передавання, трансляцію та інтерпретацію. Дивіться подійно-орієнтоване програмування.

Витрати на створення повідомлень. Створення будь-якого повідомлення вимагає додаткових витрат як процесора так і пам'яті. Створення цілочисельного повідомлення (яке може бути посиланням на рядок, масив чи іншу структуру даних) вимагає менших витрат, ніж створення складного повідомлення такого як наприклад у SOAP. Для оптимізації потрібно зменшувати довжину повідомлення та збільшувати міру вмісту, яка в нього вкладається.

Витрати на передачу повідомлень. Оскільки для отримання змісту, повідомлення потрібно передати повністю, потрібно це оптимізувати. Коротші повідомлення передаються і приймаються швидше.

Витрати на трансляцію повідомлень. Протоколи повідомлень, і вони самі часто містять надлишкову інформацію (наприклад опис пакету і його структури). Тому отримувач часто потребує перетворення повідомлення в простішу форму, видаленням додаткових символів та інформації про структуру та/або приведенням типу значень до потрібного. Для оптимізації процесів трансляції повідомлення знову ж таки має мати якомога меншу довжину, за якомога більшого вмісту.

Витрати на інтерпретацію повідомлень. Всі повідомлення мають інтерпретуватися приймачем. Прості повідомлення, такі як цілі, можуть не вимагати додаткової обробки. Проте, складні повідомлення, наприклад у SOAP, потребують парсера та перетворення рядків для відтворення змісту.

Способи вирішення проблем. Одним з підходів до зменшення зв'язності є функціональне проєктування, яке намагається обмежити відповідальності модулів у функціональності.

Слабка зв'язність виникає тоді, коли один модуль взаємодіє з іншим через простий та стабільний інтерфейс і його не хвилює як реалізований інший модуль. (дивіться: Інкапсуляція).

Такі системи як CORBA чи COM дозволяють об'єктам взаємодіяти між собою без необхідності знати що-небудь про реалізацію один одного. Обидві системи навіть дозволяють взаємодію між об'єктами написаними різними мовами.

Зв'язність та пов'язаність. Зв'язність та пов'язаність – два терміни, які дуже часто вживаються разом. Вони описують якості, які повинні мати модулі. Зв'язність характеризує взаємозв'язки між модулями, а пов'язаність описує зв'язок функцій усередині модуля. Низька пов'язаність спричиняє ситуацію, коли модуль виконує різні непов'язані завдання, і починає створювати проблеми, коли модуль стає великим.

Кожен об'єкт є **реалізацією** класу.

У моделюванні UML зв'язок реалізації – це зв'язок між двома елементами моделі, в якому один елемент моделі (клієнт) реалізує (впроваджує або виконує) поведінку, яку визначає інший елемент моделі (постачальник).

Асоціація – це ставлення, у якому об’єкти одного типу якимось чином пов’язані з об’єктами іншого типу. Наприклад, об’єкт одного типу містить або використовує об’єкт іншого типу. Наприклад, гравець грає у певній команді:

```
classTeam
{

}
classPlayer
{
     publicTeam Team { get; set; }
}
```

Клас Player пов’язаний з ставленням асоціації з класом Team.

Композиція визначає відношення HAS A , тобто відношення “має”. Наприклад, клас автомобіля містить об’єкт класу електричного двигуна:

```
public class ElectricEngine{ }
public class Car
{
    ElectricEngine engine;
    public Car()
    {        engine = new ElectricEngine();    }
}
```

У цьому клас автомобіля повністю управляє життєвим циклом об’єкта двигуна. При знищенні об’єкта автомобіля в області пам’яті разом з ним буде знищено об’єкт двигуна. І в цьому плані об’єкт автомобіля є головним, а об’єкт двигуна – залежним.

Від композиції слід відрізняти **агрегацію**. Вона також передбачає відношення HAS A , але вона реалізується інакше:

```
publicabstractclassEngine{ }
publicclassCar
{
     Engine engine;
     publicCar(Engine eng)
     {     engine = eng;     }
}
```

Спадкування є базовим принципом ООП і дозволяє одному класу (спадкоємцю) успадкувати функціонал іншого класу (батьківського). Нерідко відносини успадкування ще називають генералізацією чи узагальненням. Спадкування визначає відношення IS A , тобто “є”. Наприклад:

```
classUser
{
     publicintId { get; set; }
     publicstringName { get; set; }
}
classManager : User
{
     publicstringCompany{ get; set; }
}
```

Реалізація передбачає визначення інтерфейсу та її реалізація у класах. Наприклад, є інтерфейс IMovable з методом Move, який реалізується у класі Car:

```
publicinterfaceIMovable
{
     voidMove();
}
publicclassCar : IMovable
{
```

```
        publicvoidMove()
        {     Console.WriteLine("Машина едет");     }
}
```

Діаграма класів

Асоціація (англ. association) показує, що об'єкти однієї сутності (класу) пов'язані з об'єктами іншої сутності. Якщо між двома класами визначена асоціація, то можна переміщатися від об'єктів одного класу до об'єктів іншого. Цілком припустимі випадки, коли обидва кінці асоціації відносяться до одного і того ж класу. Це означає, що з об'єктом деякого класу дозволено зв'язати інші об'єкти з того ж класу. Асоціація, що зв'язує два класи, називається бінарною. Можна, хоча це рідко буває необхідним, створювати асоціації, що зв'язують відразу кілька класів; вони називаються n-арними. Графічно асоціація зображується у вигляді лінії, що з'єднує клас сам з собою або з іншими класами.

Асоціації може бути присвоєно ім'я, яке описує природу відносин. Зазвичай ім'я асоціації не вказується, якщо тільки ви не хочете явно задати для неї рольові імена або у вашій моделі настільки багато асоціацій, що виникає необхідність посилатися на них і відрізняти один від одного. Ім'я буде особливо корисним, якщо між одними і тими ж класами існує кілька різних асоціацій.

Клас, що бере участь в асоціації, грає в ній деяку роль. По суті, це «обличчя», яким клас, що знаходиться на одній стороні асоціації, звернений до класу з іншого її боку. Ви можете явно позначити роль, яку клас грає в асоціації.

Часто при моделюванні буває важливо вказати, скільки об'єктів може бути пов'язано допомогою одного примірника асоціації. Це число називається кратністю (Multiplicity) ролі асоціації та записується або як вираз, значенням якого є діапазон значень, або в явному вигляді. Вказуючи кратність на одному кінці асоціації, ви тим самим говорите, що на цьому кінці саме стільки об'єктів повинно відповідати кожному об'єкту на протилежному кінці. Кратність можна задати рівною одиниці (1), можна вказати діапазон: «нуль або одиниця» (0..1), «багато» (0 .. *), «одиниця або більше» (1 .. *). Дозволяється також вказувати певне число (наприклад, 3). За допомогою списку можна задати і більш складні кратності, наприклад 0. . 1, 3..4, 6 .. *, що означає «будь-яке число об'єктів, крім 2 і 5». Спрямована асоціація (англ. Navigable Association) - відносини направлені тільки в одному напрямку.

Агрегація (англ. aggregation)– проста асоціація між двома класами, де один з класів має вищий ранг (ціле) і складається з декількох менших за рангом (частин).

Ставлення такого типу також називають слабкою агрегацією (англ. weak aggregation); воно зараховане до відносин типу «має» (з урахуванням того, що об'єкт-ціле має кілька об'єктів-частин). Агрегація є окремим випадком асоціації і її зображують як просту асоціацію з незафарбованим ромбом з боку «цілого».

За потреби можна вказувати кратність агрегації.

Агрегація не накладає жорстких умов на термін існування об'єктів. Наприклад, «частини» можуть існувати тоді, коли «ціле» зникає.

Графічно агрегація представлена порожнім ромбом на блоці «цілого», і лінією, яка проведена від цього ромба до класу, що міститься в ньому («частин»).

Композиція (англ. composition) – більш строгий варіант агрегації (англ. strong aggregation). Відома також як агрегація за значенням. Композиція має жорстку залежність

часу існування екземплярів класу контейнера та екземплярів класів що містяться в ньому. Якщо контейнер буде знищений, то весь його вміст буде також знищено.

Графічно представляється як і агрегація, але з зафарбованим ромбиком.

Відмінності між композицією і агрегацією. Відносини між класом Виш і класами Студент і Факультет злегка відрізняються один від одного, хоча обидва є відносинами агрегування. У виші може бути будь-яка кількість студентів (включаючи нуль), і кожен студент може навчатися в одному або декількох вишах; виш може складатися з одного або декількох факультетів, але кожен факультет належить одному і лише одному вишу. Відношення між класами Виш і Факультет називають композицією, оскільки при знищенні моделі Виш моделі факультетів, що належать цьому ВНЗ, також будуть знищені. Студент і Виш пов'язані агрегацією тому, що Студента не можна видалити при знищенні Вишу.

Відносини композиції

1. При спробі представити реальні відношення «ціле-частина», наприклад, двигун є частиною автомобіля.

2. Коли руйнується контейнер, руйнується і його вміст, наприклад, університет та його факультети.

Агрегаційне відношення

1. При представленні відношення програмного забезпечення або бази даних, наприклад, двигун моделі автомобіля ENG01 є частиною моделі автомобіля CM01, оскільки двигун, ENG01, може також бути частиною іншої моделі автомобіля[9].

2. Коли контейнер знищується, його вміст зазвичай не знищується, наприклад, у професора є студенти; коли професор залишає університет, студенти не йдуть разом з ним.

Таким чином, зв'язок агрегації часто називають «каталожним» утриманням, щоб відрізнити його від «фізичного» утримання композиції.

Залежність може бути слабшою формою зв'язку, яка вказує на те, що один клас залежить від іншого, оскільки він використовує його в певний момент часу. Один клас залежить від іншого, якщо незалежний клас є змінною-параметром або локальною змінною методу залежного класу. Іноді зв'язок між двома класами дуже слабкий. Вони взагалі не реалізуються за допомогою змінних-членів. Замість цього вони можуть бути реалізовані як аргументи функцій-членів.

Графічне представлення **Реалізації** в UML – це порожнистий трикутник на інтерфейсному кінці пунктирної лінії (або дерева ліній), яка з'єднує її з одним або декількома реалізаторами. На інтерфейсному кінці пунктирної лінії, що з'єднує її з користувачами, використовується звичайна стрілка. На діаграмах компонентів використовується графічна конвенція «куля-розетка» (реалізатори показують кулю або льодяник, тоді як користувачі показують розетку). Реалізації можуть бути показані тільки на діаграмах класів або компонентів. Реалізація – це зв'язок між класами, інтерфейсами, компонентами та пакетами, який з'єднує елемент-клієнт з елементом-постачальником. Зв'язок реалізації між класами/компонентами та інтерфейсами показує, що клас/компонент реалізує операції.