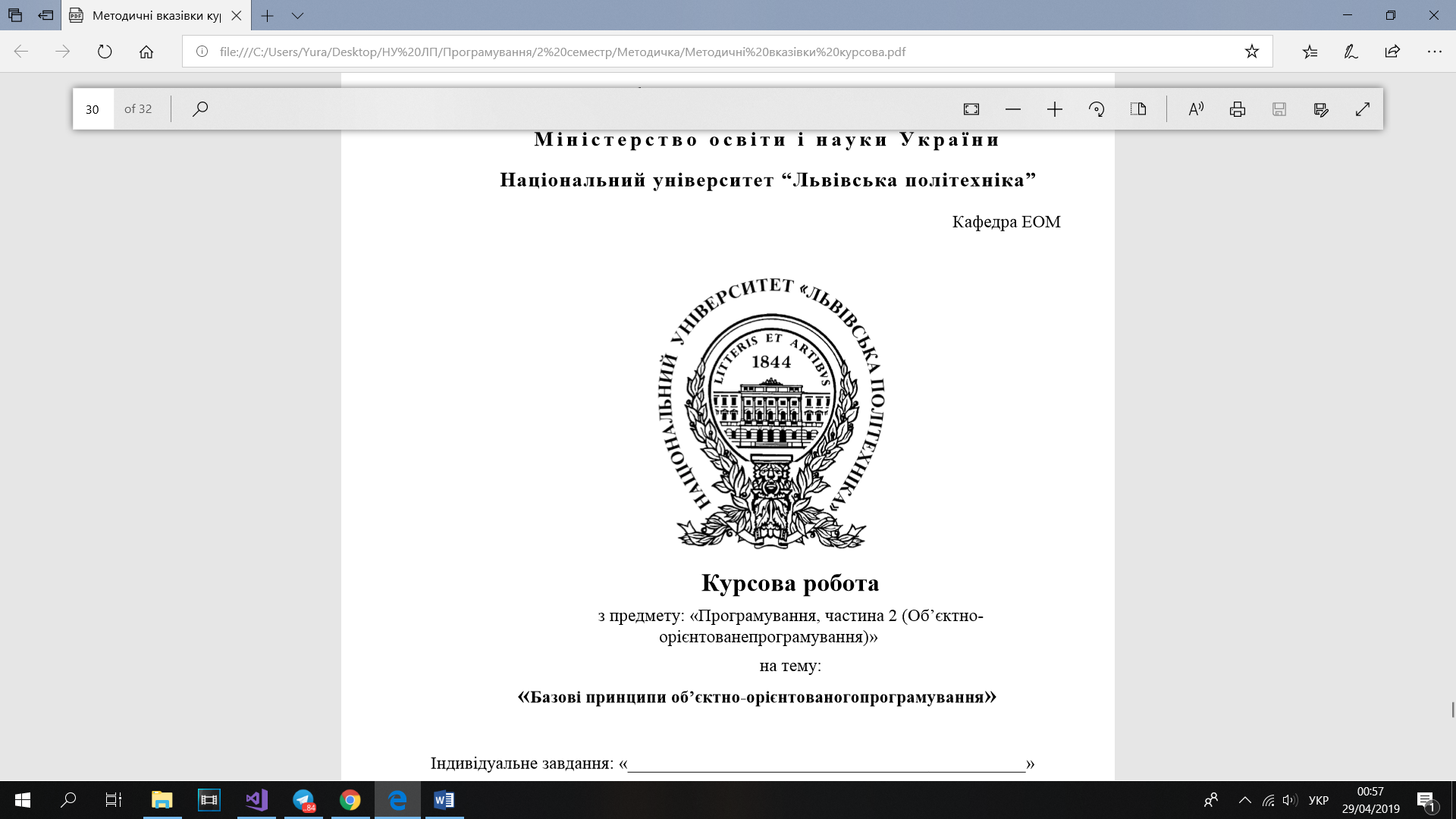
Міністерство освіти і науки України

Національний університет “Львівська політехніка”

Кафедра ЕОМ



**Курсова робота**

з предмету: «Програмування, частина 2 (Об’єктно-орієнтоване програмування)» на тему: «Базові принципи об’єктно-орієнтованого програмування»

**Індивідуальне завдання:   
«Нічний клуб»**

Виконав:

ст. гр. КІ-15

Гербей О.М.

Прийняв:

Козак Н.Б.

Львів-2019

**Анотація**

Метою курсової роботи є створення моделі функціонування нічного клуба. Написання програмного забезпечення було виконано мовою програмування C++ із використанням об’єктно-орієнтованого підходу для формування програми у вигляді набору сутностей та об’єктів, що взаємодіють. Програма включає у собі можливість роботи із текстовими файлами, а саме: запис та зчитування даних. Інтерфейс програми написано українською мовою.

**Зміст**

**Завдання на курсову роботу....................................................................................4**

**Вступ............................................................................................................................5**

**Огляд та обґрунтовування вибору технологій об'єктно-орієнтованого програмування...................................................…………………………………….......6**

**Теорія про діаграми.................................................................................................10**

**Реалізація.................................................................................................……….....14**

**Відлагодження та тестування реалізованої програми згідно індивідуального завдання.....................................................................................................................19**

**Висновки...................................................................................................................22**

**Завдання на курсову роботу**

**Індивідуальне завдання: «Нічний клуб».**

Згідно із отриманим індивідуальним завданням метою курсової роботи є реалізація мовою С++ із застосуванням об’єктно-орієнтованого підходу предметної галузі “ **Нічний клуб** ”. Програма повинна підтримувати можливості роботи з файлами: запис, зчитування.

***Вступ***

**Об’єктно-орієнтоване програмування** (ООП) — одна з парадигм програмування, яка розглядає програму як множину об'єктів, що взаємодіють між собою. В основі ООП покладено три основні положення: інкапсуляцію, успадкування та поліморфізм. Однією з переваг ООП є краща модульність програмного забезпечення, що дозволяє групам розробників працювати незалежно один від одного, розробляючи окремі функціональні компоненти програми . Попри те, що парадигма ООП з'явилась в 1960-тих роках, вона не мала широкого застосування до 1990-тих, коли розвиток комп'ютерів та комп'ютерних мереж дозволив писати надзвичайно об'ємне і складне програмне забезпечення, що змусило переглянути підходи до написання програм та покращити їх модульність. Однією з таких мов програмування стала мова С++. У 1990-х р. вона стала однією із найпопулярніших мов програмування загального призначення. Розробником цієї мови став професор Техаського університету Б’єрн Страуструп. На початку свого існування ця мова дістала назву «Сі з класами». Пізніше у 1983 році Страуструп перейменував мову на C++. У 1998 році було ратифіковано міжнародний стандарт мови С++: ISO/IEC 14882 «Standard for the C++ Programming Language». Сьогодні цю мову використовують для розробки програмного забезпечення, написання драйверів, системного програмування, потужних серверних та клієнтських програм. Мова C++ мала значний вплив на інші мови програмування такі як Java, C# та ін.

**Метою виконання** курсової роботи є засвоєння основних методів та алгоритмів об’єктно-орієнтованого програмування мови С++ та їх практичне застосування у описі сутностей конкретної сфери.

**Огляд та обґрунтовування вибору технологій об'єктно-орієнтованого програмування.**

**1.1Визначення ООП**

Об'єктно-орієнтоване програмування — це метод програмування, заснований на поданні програми у вигляді сукупності взаємодіючих об'єктів, кожен з яких є екземпляром певного класу, а класи є членами певної ієрархії наслідування[[6]](https://uk.wikipedia.org/wiki/%D0%9E%D0%B1%27%D1%94%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D1%96%D1%94%D0%BD%D1%82%D0%BE%D0%B2%D0%B0%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F" \l "cite_note-6). Програмісти спочатку пишуть клас, а на його основі при виконанні програми створюються конкретні об'єкти (екземпляри класів). На основі класів можна створювати нові, які розширюють базовий клас і таким чином створюється ієрархія класів.

На думку Алана Кея, розробника мови Smalltalk, якого вважають одним з «батьків-засновників» ООП, об'єктно-орієнтований підхід полягає в наступному наборі основних принципів:

* Все є об'єктами.
* Всі дії та розрахунки виконуються шляхом взаємодії (обміну даними) між об'єктами, при якій один об'єкт потребує, щоб інший об'єкт виконав деяку дію. Об'єкти взаємодіють, надсилаючи і отримуючи повідомлення. Повідомлення — це запит на виконання дії, доповнений набором аргументів, які можуть знадобитися при виконанні дії.
* Кожен об'єкт має незалежну пам'ять, яка складається з інших об'єктів.
* Кожен об'єкт є представником (екземпляром, примірником) класу, який виражає загальні властивості об'єктів.
* У класі задається поведінка (функціональність) об'єкта. Таким чином усі об'єкти, які є екземплярами одного класу, можуть виконувати одні й ті ж самі дії.
* Класи організовані у єдину деревоподібну структуру з загальним корінням, яка називається ієрархією успадкування. Пам'ять та поведінка, зв'язані з екземплярами деякого класу, автоматично доступні будь-якому класу, розташованому нижче в ієрархічному дереві.

Таким чином, програма являє собою набір об'єктів, що мають стан та поведінку. Об'єкти взаємодіють використовуючи повідомлення. Будується ієрархія об'єктів: програма в цілому — це об'єкт, для виконання своїх функцій вона звертається до об'єктів що містяться у ньому, які у свою чергу виконують запит шляхом звернення до інших об'єктів програми. Звісно, щоб уникнути безкінечної рекурсії у зверненнях, на якомусь етапі об'єкт трансформує запит у повідомлення до стандартних системних об'єктів, що даються мовою та середовищем програмування. Стійкість та керованість системи забезпечуються за рахунок чіткого розподілення відповідальності об'єктів (за кожну дію відповідає певний об'єкт), однозначного означення інтерфейсів міжоб'єктної взаємодії та повної ізольованості внутрішньої структури об'єкта від зовнішнього середовища (інкапсуляції).

**1.2Фундаментальні поняття**

В результаті дослідження Дебори Дж. Армстронг (англ. Deborah J. Armstrong) комп'ютерної літератури, що була видана протягом останніх 40 років, вдалось відокремити фундаментальні поняття (принципи), використані у переважній більшості визначень об'єктно-орієнтованого програмування. До них належить:

Клас

Клас визначає абстрактні характеристики деякої сутності, включаючи характеристики самої сутності (її атрибути або властивості) та дії, які вона здатна виконувати (її поведінки, методи або можливості). Наприклад, клас Собака може характеризуватись рисами, притаманними всім собакам, зокрема: порода, колір хутра, здатність гавкати. Класи вносять модульність та структурованість в об'єктно-орієнтовану програму. Як правило, клас має бути зрозумілим для не-програмістів, що знаються на предметній області, що, у свою чергу, значить, що клас повинен мати значення в контексті. Також, код реалізації класу має бути досить самодостатнім. Властивості та методи класу, разом називаються його членами.

***Об'єкт***

Окремий екземпляр класу (створюється після запуску програми і ініціалізації полів класу). Клас Собака відповідає всім собакам шляхом опису їхніх спільних рис; об'єкт Сірко є одним окремим собакою, окремим варіантом значень характеристик. Собака має хутро; Сірко має коричнево-біле хутро. Об'єкт Сірко є екземпляром (примірником) класу Собака. Сукупність значень атрибутів окремого об'єкта називається станом. На основі класу Собака можна, також, створити інший об'єкт Дружок, який відрізнятиметься від об'єкта Сірко своїм станом (наприклад кольором хутра). Обидва об'єкта (Сірко і Дружок) є екземплярами класу Собака.

***Метод***

Можливості об'єкта. Оскільки Сірко — Собака, він може гавкати. Тому гавкати() є одним із методів об'єкта Сірко. Він може мати й інші методи, зокрема: місце(), або їсти(). В межах програми, використання методу має впливати лише на один об'єкт; всі Собаки можуть гавкати, але треба щоб гавкав лише один окремий собака.

***Обмін повідомленнями***

«Передача даних від одного процесу іншому, або надсилання викликів методів.»

***Абстрагування***

Спрощення складної дійсності шляхом моделювання класів, що відповідають проблемі, та використання найприйнятнішого рівня деталізації окремих аспектів проблеми. Наприклад Собака Сірко більшу частину часу може розглядатись як Собака, а коли потрібно отримати доступ до інформації специфічної для собак породи коллі — як Коллі і як Тварина (можливо, батьківський клас Собака) при підрахунку тварин Петра.

**1.3Об’єктно-орієнтовані особливості С++**

Сі++ додає до Сі об'єктно-орієнтовані можливості. Він вводить класи, які забезпечують три найважливіші властивості ООП: інкапсуляція, поліморфізм, успадкування.

**Інкапсуляція**

Основним способом організації інформації в С++ є класи. На відміну від типу, структура (struct) мови С, що складається тільки з полів, клас (class) С++ складається з полів і функцій-членів або методів (англ. member functions). Поля бувають публічними (public), захищеними (protected) і приватними (private). У С++ тип структура аналогічний типу клас, відмінність в тому, що за умовчанням поля і функції-члени у структури публічні, а у класу — приватні.

З публічними полями можна робити зовні класу все, що завгодно. До захищених і приватних полів не можна звертатися ззовні класу, щоб не порушити цілісність даних класу. Спроба такого звернення викличе помилку компіляції. До таких полів можуть звертатися тільки функції-члени класу (а також так звані функції-друзі і функції-члени класів-друзів; про поняття друзів в C++ дивись нижче.) Поза тілом функцій-членів (а також друзів) захищені і власні поля недоступні навіть для читання. Такий захист полів називається інкапсуляциєю.

Використовуючи інкапсуляцію, автор класу може захистити свої дані від некоректного використання. Крім того, вона замислювалася для полегшення сумісної розробки класів. Малося на увазі, що зміна способу зберігання даних, якщо вони оголошені як захищені або приватні, не вимагає відповідних змін в класах, які використовують змінений клас. Наприклад, якщо в старій версії класу дані зберігалися у вигляді лінійного списку, а в новій версії — у вигляді дерева, ті класи, які були написані до зміни формату зберігання даних, переписувати не буде потрібно, якщо дані були приватними або захищеними (у останньому випадку — якщо використовуючі класи не були класами-нащадками), оскільки жоден з них цих класів не міг би безпосередньо звертатися до даних, а тільки через стандартні функції, які в новій версії мають вже коректно працювати з новим форматом даних. Навіть оператор доступу operator [] може бути визначений як така стандартна функція.

Функції-члени, як і поля, можуть бути публічними, захищеними і приватними. Публічні функції може викликати будь-хто, а захищені і власні — тільки функції-члени і друзі.

**Успадкування**

Для створення класів з доданою функціональністю вводять успадкування. Клас-нащадок має поля і функції-члени базового класу, але не має права звертатися до приватних (private) полів і функцій базового класу. У цьому і полягає різниця між приватними і захищеними членами.

Клас-нащадок може додавати свої поля і функції або перевизначати функції базового класу.

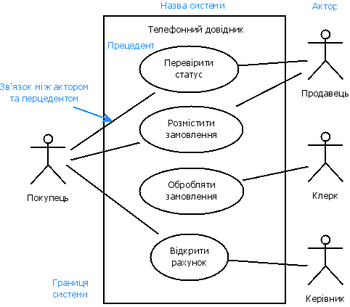
За умовчанням, конструктор нащадка без параметрів викликає конструктор базового класу, а потім конструктори доданих елементів. Деструктор працює в зворотному порядку. Інші конструктори доводиться визначати щоразу наново. На щастя, це можна зробити викликом конструктора базового класу.

**Поліморфізм**

Поліморфізмом в програмуванні називається перевизначення нащадком функцій-членів базового класу

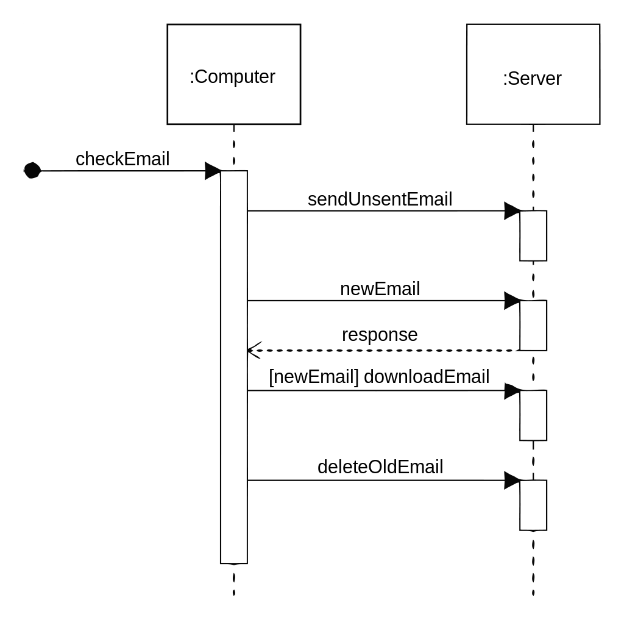
**Теорія про діаграми**

**Діаграма прецедентів** — в UML, діаграма, на якій зображено відношення між акторами та прецедентами в системі. Також, перекладається як діаграма варіантів використання. Діаграма прецедентів є графом, що складається з множини акторів, прецедентів (варіантів використання) обмежених границею системи (прямокутник), асоціацій між акторами та прецедентами, відношень серед прецедентів, та відношень узагальнення між акторами.



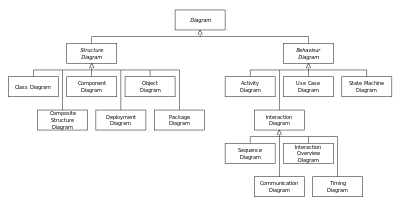
Приклад use case diagram

**Діаграма послідовності** (англ. sequence diagram) — різновид діаграми в UML. Діаграма послідовності відображає взаємодії об'єктів впорядкованих за часом. Зокрема, такі діаграми відображають задіяні об'єкти та послідовність відправлених повідомлень На діаграмі послідовностей показано у вигляді вертикальних ліній різні процеси або об'єкти, що існують водночас. Надіслані повідомлення зображуються у вигляді горизонтальних ліній, в порядку відправлення. Визначені стандартом UML 2.0 діаграми послідовностей мають ті ж можливості що і визначені стандартом UML 1.x, та підтримують додаткові можливості зміни стандартного порядку повідомлень.



Приклад sequence diagram

**Діагра́ма кла́сів** (англ. class diagram) — статичне представлення структури моделі. Відображає статичні (декларативні) елементи, такі як: класи, типи даних, їх зміст та відношення. Діаграма класів, також, може містити позначення для пакетів та може містити позначення для вкладених пакетів. Також, діаграма класів може містити позначення деяких елементів поведінки, однак їх динаміка розкривається в інших типах діаграм. Діаграма класів служить для представлення статичної структури моделі системи в термінології класів об'єктно-орієнтованого програмування. На цій діаграмі показують класи, інтерфейси, об'єкти й кооперації, а також їхні відносини.



Приклад class diagram

#### Асоціації

Асоціація 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-арнимі. 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 відносини. 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».

#### Агрегація

[KP-UML-Aggregation-20060420.svg](https://uk.wikipedia.org/wiki/%D0%A4%D0%B0%D0%B9%D0%BB:KP-UML-Aggregation-20060420.svg)

Агрегація — проста асоціація між двома класами, яка відображає структурне відношення між рівноправними сутностями, коли обидва класи знаходяться на одному концептуальному рівні, і жоден з них не важливіший за решту. Але іноді доводиться моделювати відношення типу «частина/ціле», в якому один з класів має вищий ранг (ціле) і складається з декількох менших за рангом (частин). Ставлення такого типу називають агрегацією; 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 зображуєть як просту асоціацію з незафарбованим ромбом з боку «цілого».

Графічно агрегація представлена порожнім ромбом на блоці класу, і лінією, яка проведена від цього ромба до класу, що міститься в ньому.

#### Композиція

Композиція — більш суворий варіант агрегації. Відома також як агрегація за значенням.

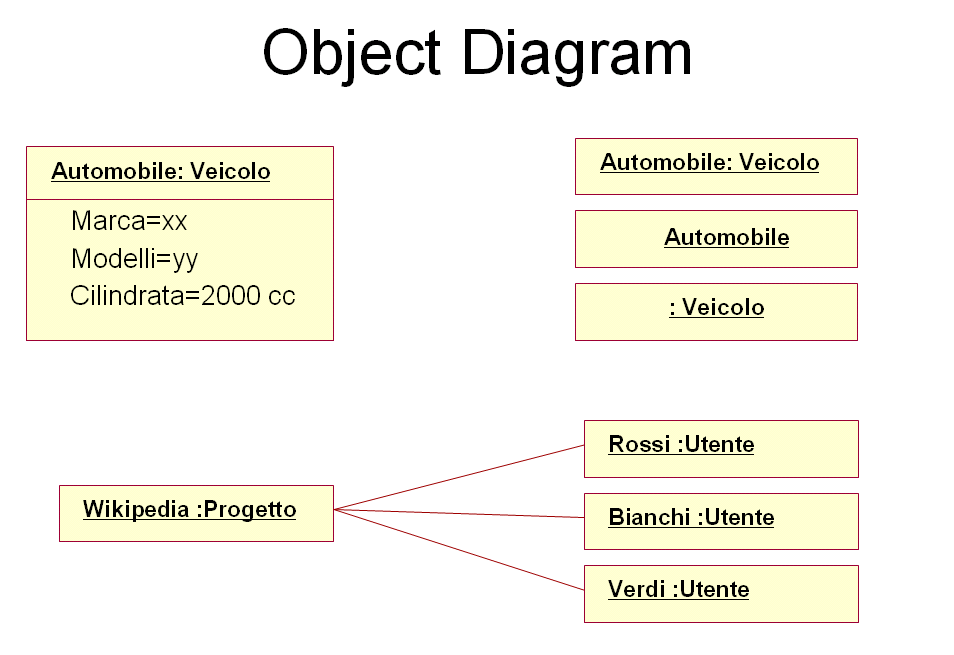
Композиція має жорстку залежність часу існування екземплярів класу контейнера та примірників містяться класів. Якщо контейнер буде знищений, то весь його вміст буде також знищено.

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. В 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, так як при знищенні моделі Виш моделі факультетів, що належать цьому ВНЗ, також повинні бути знищені. Студент і Виш пов'язані агрегацією тому, що Студента не можна видалити при знищенні Вишу.

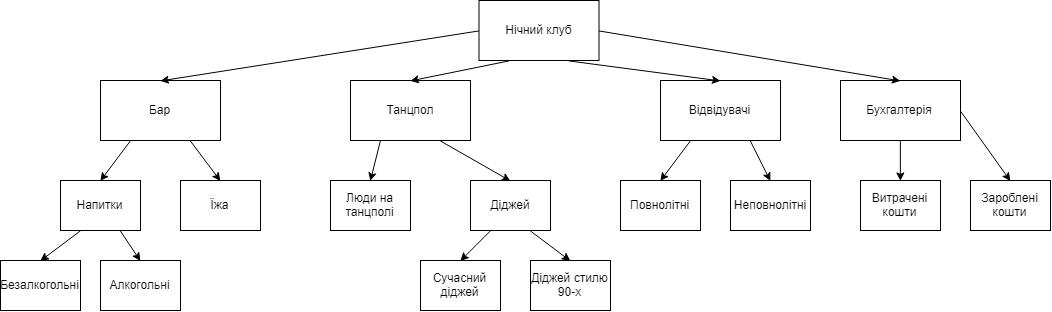
**Діаграма об'єктів —** в UML, діаграма, що відображає об'єкти та їх зв'язки в певний момент часу. Діаграма об'єктів може розглядатись як окремий випадок діаграми класів, на якій можуть бути представлені як класи, так і екземпляри (об'єкти) класів. Схожою за змістом є діаграма взаємодії (англ. collaboration diagram). Діаграми об'єктів не мають власної нотації. Оскільки діаграми класів можуть відображати об'єкти, то діаграма класів, на якій відображено лише об'єкти, та не відображено класи, може вважатись діаграмою об'єктів.



Приклад object diagram

**Реалізація**

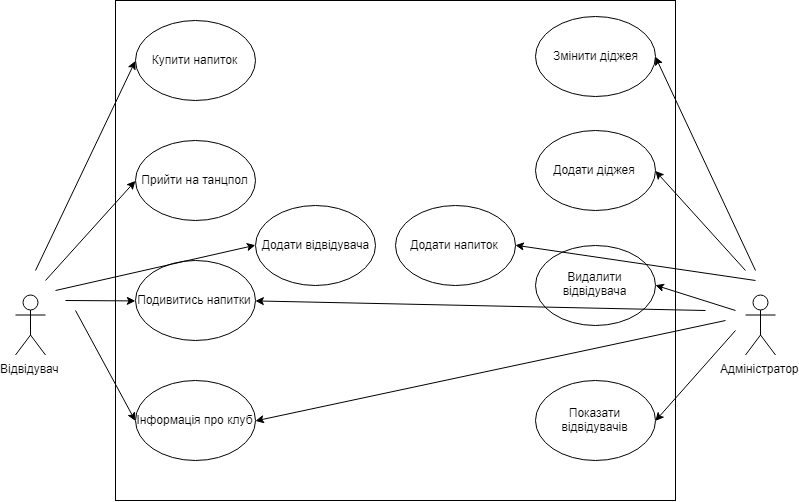
1. **Аналіз об'єктно-орієнтованої моделі проетку**
   1. **Аналіз структури предметної області**



Структура предментої області

«Нічний клуб»

* 1. **Аналіз функціонального наповнення проетку**



* 1. **Аналіз об'єктно-орієнтованої моделі проетку**

**\*Діаграма класів\***

**2.Огляд структури меню**

В алгоритмі наяві 3 підменю, шо утворюють меню консольної програми. Перше розгалуження відповідає за головне меню програми, що включає такі можливості:

а) отримати інформацію про нічний клуб

б) відвідувачі;

в) танцпол;

г) бар;

д) закрити меню.

При виконанні переходу до пункту «отримати інформацію про нічний клуб» перед користувачем з'являється вся потрібна інформація про даний нічний клуб.

При виконанні переходу до пункту «відвідувачі», користувач поптрапляє у меню «відвідувачі»:

а) показати відвідувачів;

б) додати відвідувачів;

в) видалити відвідувачів

г) закрити меню.

При виконанні переходу до пункту «танцпол», користувач потрапляє у меню «танцпол»:

а) змінити діджея;

б) додати людей на танцпол;

в) показати діджея;

г) закрити меню;

При виконанні переходу до пункту «бар», користувач потрапляє у меню «бар»:

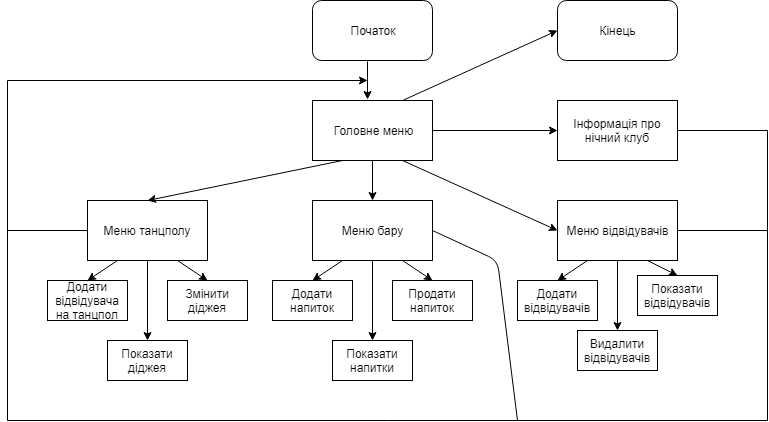
а) показати напої;

б) додати напої;

в) продати напої;

г) закрити меню;

При виконанні переходу до пункту «закрити меню», користувач закриває програму.



**Рис.1 блок-схема програми**

**Програмна реалізація алгоритму згідно індивідуального завдання.**

В програмі наявні змінні:

а) типу int, float, double, string, vector для виконання обчислень та виведення в консоль;

б) vector об’єктів класу для виконання методів.

**Порядок викликів функцій:**

* Виклик головного методу «Головне меню» (run();)
* Виклик методу «Меню відвідувачів» (visitorMenu();)
* Виклик методу «Меню танцполу» (danceFloorMenu();)
* Виклик методу «Меню бару» (barMenu();

**Опис реалізації основних функцій/методів:**

1. Функція зчитування файлу

void Bar::readFileBar() {

drinkName.open("drinkName.txt");

drinkPrice.open("drinkPrice.txt");

string str, strTemp;

int prc, prcTemp = 0;

if (drinkName.is\_open()) {

while (true) {

drinkName >> str;

if (str == strTemp)

break;

else {

drink.push\_back(str);

strTemp = str;

}

drinkPrice >> prc;

if (prc == prcTemp)

break;

else {

price.push\_back(prc);

prcTemp = prc;

}

}

}

else

cout << "File not found!" << endl;

drinkName.close();

drinkPrice.close();

}

1. Функція додавання нового напою

void Bar::addDrink() {

drinkName.open("drinkName.txt", ios::app);

drinkPrice.open("drinkPrice.txt", ios::app);

int choice = 1;

string nameTemp;

int priceTemp;

while (choice == 1) {

cout << "What drink would you like to add?" << endl;

cin >> nameTemp;

cout << "Enter " << nameTemp << " price:" << endl;

cin >> priceTemp;

drinkName << nameTemp << endl;

drinkPrice << " " << priceTemp << " ";

cout << "Add more ?" << endl << "1 - yes" << endl << "0 - no" << endl;

cin >> choice;

}

drinkName.close();

drinkPrice.close();

readFileBar();

}

1. Функція продажу напою

void Bar::sellDrinks() {

if (visitorCount != 0) {

int choice = pickVisitor(),

choice1;

cout << "What drink would you like to buy ?" << endl;

showDrinks();

cin >> choice1;

choice1--;

visitors[choice].setMoney(visitors[choice].getMoney() - price[choice1]);

cout << drink[choice1] << " successfully bought !" << endl <<

"Money left: " << visitors[choice].getMoney() << "$" << endl;

setBarProfit(getBarProfit() + price[choice1]);

}

else

cout << "There is no visitors !" << endl;

}

1. Функція повторного зчитування файлу

void Bar::rewriteFileBar() {

for (int i = 0; i < drink.size(); i++) {

drink.erase(drink.begin(), drink.end());

price.erase(price.begin(), price.end());

}

readFileBar();

}

1. Функція додавання людей на танцпол

void DanceFloor::addPeopleToFloor() {

int choice = pickVisitor();

if (visitors[choice].getDancing() == true) {

cout << "Visitor already is in dance floor !" << endl;

}

else {

visitors[choice].setDancing(true);

cout << visitors[choice].getName() << " added to dance floor" << endl;

}

}

1. Функція додання відвідувача

void Human::addVisitor() {

Human \*t;

t = new Human;

t->setAllVisitor();

visitors.push\_back(\*t);

visitorCount++;

}

1. Функція показу відвідувачів

bool Human::showVisitors() {

if (visitorCount != 0) {

int count = 1;

cout << "Pick visitor: " << endl;

for (it = visitors.begin(); it != visitors.end(); it++) {

cout << setw(5) << count << ": ";

string name = it->getName();

cout << name << endl;

count++;

}

return true;

}

else {

cout << "No visitors, add some!" << endl;

return false;

}

}

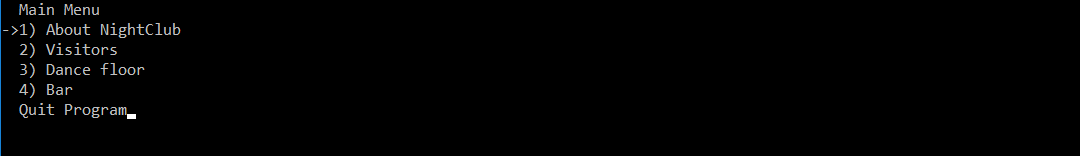
**Відлагодження та тестування реалізованої програми згідно індивідуального завдання.**

Під час тестування окремих частин програм з’являлись певні недоліки та помилки. Щоб все правильно працювало потрібно було не дати можливості користувачу змушувати програму некоректно працювати. Тож всі можливі неполадки були ізольовані і опрацьовані так, щоб з’являлось певне повідомлення з певною інформацією, звідки користувач дізнавався як слід користуватися програмою, щоб уникати таких невизначеностей.

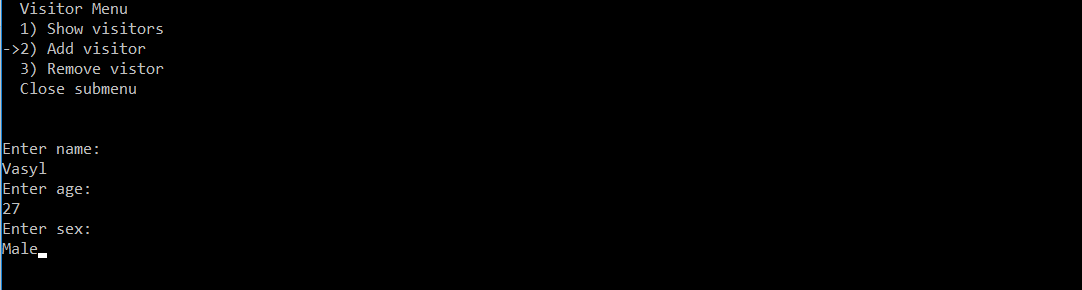
**Відлагодження та тестування програми в цілому:**

Подібні кроки були виконані і з перевіркою всієї програми. Нижче наведені скріншоти, які запевнюють в коректній роботі програми:

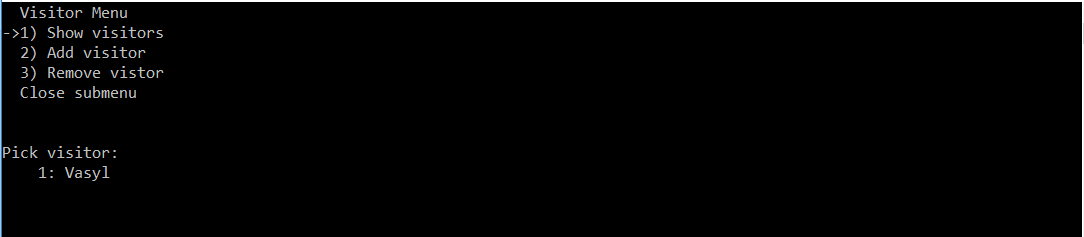
1)



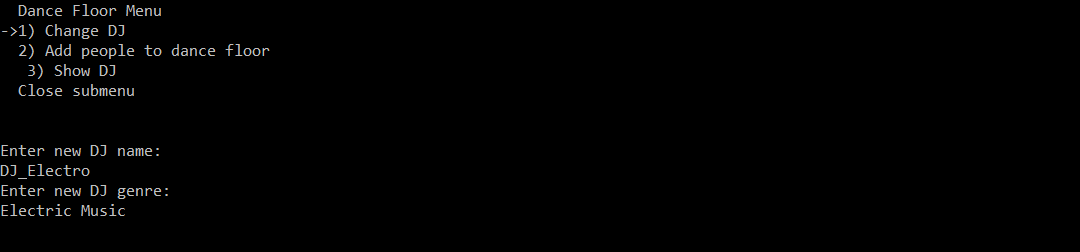
2)



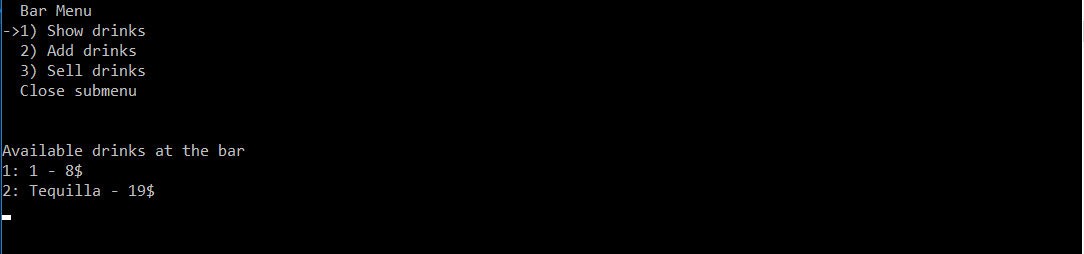
3)



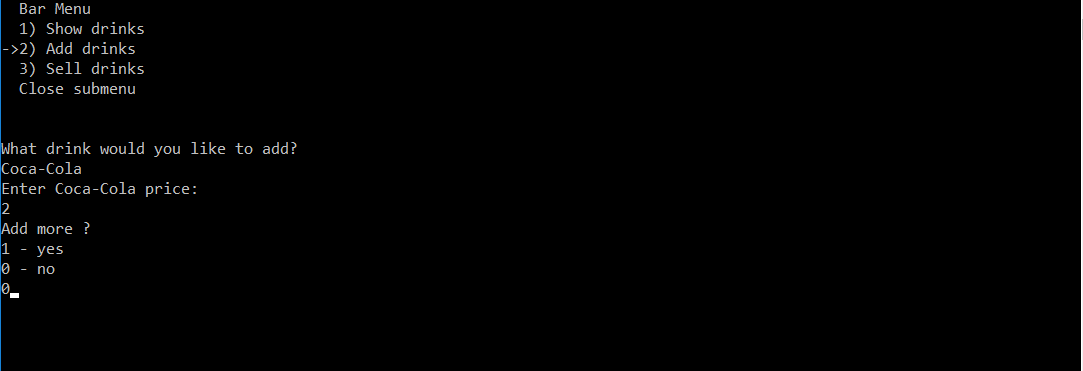
4)



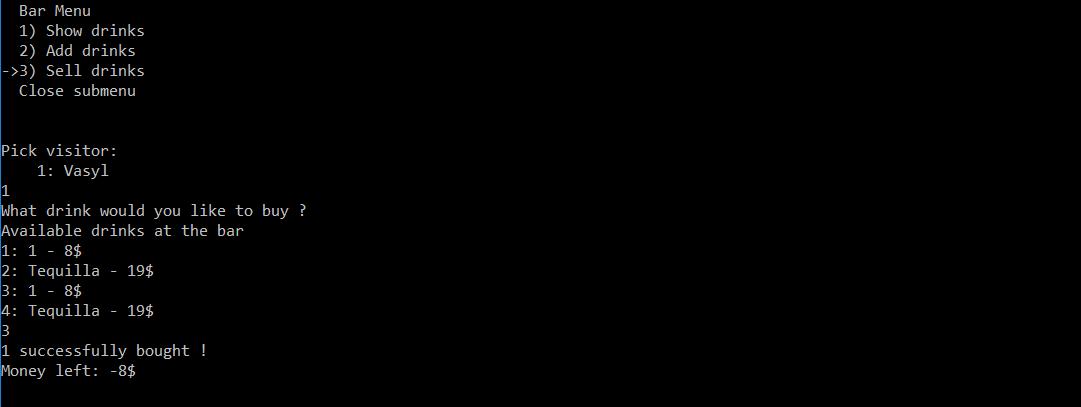
5)



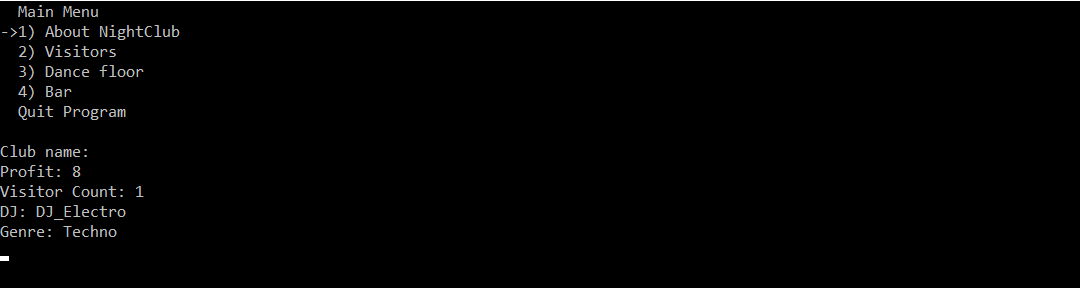
6)



7)



8)



**Аналіз структурної складності виконання програми**

Оцінюючи структуру програми, потрібно наголосити, що наявні 3 конструкції switch, які полегшують не тільки користування програмою в ролі користувача, але як на мене, полегшують власне кодування. Ці конструкції начебто уявно відділяють блок коду від усієї програми, що покращує сприйняття й розуміння виконаних та запланованих цілей (функцій, операцій). Як висновок, можу сказати що, виконання програми на структурному рівні не є складним.

**Аналіз часу виконання програми в залежності від об'єму вхідних даних**

Всі операції всередині програми виконуються миттєво, немає затримки між переходами по меню та виведенням інформації на консоль. Читання даних з файлу та виконання операцій з числами виконується бездоганно.

**Висновки**

Використання ООП для вирішення задач такого типу значно полегшує програмісту роботу, робить код більш структурованим та дозволяє легше його читати. Для себе можу зробити важливий висновок, що ООП — гнучкий інструмент який ідеально підходить для написання складних програм які потребують описання деяких об’єктів та їх поведінки.