{PROGRAMMING}

STEP computer ACADEMY

UNITIED MODELING LANGUAGE

Урок №1

Введення в об'єктно-орієнтований аналіз та проектування

Зміст

1. Складнощі розробки програмного забезпечення

На наш час дуже зросла складність створюваних систем. Проявами цієї складності є не тільки збільшення розмірів та функціональності. Значна кількість проблем в чстою зміною потреб користувачів і зростанням вимог до якості систем.

Традиційні способи вирішення складнощів, такі як: збільшення складу розробників, їх спеціалізація, розподіл задач, ведуть до ще більших труднощів узгодження результатів та складання готових систем. Серйозним кроком до перемоги над складністю виникла поява об'єктно-орієнтованого програмування (ООП). Але лише кроком. Поява ООП не усунула складнощі недостатнього взаєморозуміння розробників і користувачів, неефективного управління розробкою в умовах мінливих вимог, не контрольованості змін у процесі виконання робіт, суб'єктивності щодо оцінки якості продуктів розробки тощо.

2. Причини виникнення ООП

На перших комп'ютерах для написання програм спочатку застосовувалися машинні коди, а згодом мова асемблера. Але ця мова не відповідає сьогоднішнім стандартам. У міру підвищення складності програм виявилося, що розробники неспроможні пам'ятати всю інформацію, необхідну для налагодження та вдосконалення їх програм. Які значення зберігаються у регістрах? Чи є змінна із цим ім'ям? Які змінні треба ініціалізувати перед тим, як передати управління наступному коду?

Частково ці проблеми вирішили перші мови високого рівня: Фортран, Кобол, Алгол. Але зростання складності програм тривало, і з'явилися проєкти, в яких жоден програміст не міг утримати у голові усі деталі. Над проєктами почали працювати команди програмістів. Значна взаємозалежність елементів ПЗ заважає створювати ПЗ на кшталт конструювання матеріальних об'єктів. Наприклад, будинок, автомобіль та електроприлади зазвичай збирають з готових компонентів, для яких не потрібна розробка «з нуля». Багаторазове використання ПЗ — мета, до якої постійно прагнуть, але і якої рідко досягають. З програмної системи важко вилучити незалежні фрагменти. ООП полегшує це завдання.

3. Базові поняття

Саме застосування об'єктно-орієнтованої мови не змушує до написання об'єктно-орієнтованих програм, хоч і спрощує їхню розробку. Щоб ефективно використовувати ООП, потрібно розглядати завдання іншим способом, ніж це закладено в процедурному програмуванні.

Відомо твердження, застосовне до природних мов, що мова, якою висловлюється ідея, спрямовує мислення. Комп'ютерним і природним мовам притаманне твердження: «Мова спрямовує думки, але не встановлює їх».

І за аналогією — об'єктно-орієнтована техніка не забезпечує програміста новою обчислювальною потужністю, яка дозволила б вирішити проблеми, недоступні іншим засобам. Але об'єктно-орієнтований підхід полегшує завдання і зводить її до більш природньої форми. Це дозволяє підходити до проблеми у такий спосіб, який сприяє управлінню великими програмними системами.

ООП часто називають новою парадигмою програмування. Парадигма програмування — спосіб концептуалізації, який визначає, як проводити обчислення і як робота, яку виконує комп'ютер, повинна бути структурована і організована.

Процес розподілу завдання на окремі, структурно пов'язані, частини, називають **декомпозицією**. При процедурній декомпозиції у задачі виділяються алгоритми і оброблювані ними структури даних, при логічній декомпозиції — правила, пов'язані з окремими поняттями.

При об'єктно-орієнтованій декомпозиції в завданні виділяються класи і способи взаємодії об'єктів цих класів один з одним.

Головний спосіб боротьби зі складністю ПЗ — абстрагування. Тобто, здатність відокремити логіку фрагменту програми від проблеми його реалізації.

Абстрагування — це виділення таких суттєвих характеристик об'єкта, які відрізняють його від усіх інших видів об'єктів. В такий спосіб чітко визначаються особливості даного об'єкта з погляду подальшого розгляду. Абстрагування дозволяє відокремити найістотніші особливості поведінки від несуттєвих особливостей. Абстракція визначає суттєві характеристики деякого об'єкта, які відрізняють його від інших видів об'єктів, і чітко окреслює концептуальний кордон об'єкта з погляду спостерігача.

Модульність — поліпшений метод створення та управління сукупностями імен і пов'язаними з ними значеннями. Суть модуля полягає у розподілі простору імен на дві частини: відкрита (public) частина є доступною ззовні модуля, закрита (private) частина доступна лише всередині модуля.

Ієрархія — ранжована (упорядкована) система абстракцій.

4. Огляд існуючих методологій

4.1. Методологія процедурно-орієнтованого програмування

Поява перших електронних обчислювальних машин і комп'ютерів посприяла новому етапу у розвитку техніки обчислень. Здавалося, достатньо розробити послідовність елементарних дій, кожну з яких перетворити на зрозумілі комп'ютеру інструкції, і будь-яке обчислювальне завдання буде вирішене. Ця ідея виявилася настільки життєздатною, що довго домінувала над усім процесом розробки програм. З'явилися спеціальні мови програмування, які дозволили перетворювати окремі обчислювальні операції на відповідний програмний код. До таких мов належать Assembler, C, Pascal та інші.

Основою даної методології розробки програм була процедурна чи алгоритмічна організація структури програмного коду. Це було настільки очевидно для вирішення обчислювальних завдань, що ні в кого не викликало сумніву щодо доцільності такого підходу. Вихідним поняттям цієї методології було поняття **алгоритму**, під яким розуміється деяке розпорядження виконати точно певну послідовність дій, вкладених у досягнення заданої мети чи рішення поставленої задачі.

Прийнято вважати, що термін «алгоритм» походить ще з Середньовіччя від імені математика Аль-Хорезмі,

який у 825 р. виділив правила виконання арифметичних дій у десятковій системі числення.

Вся історія математики тісно пов'язана з розробкою тих чи інших алгоритмів вирішення завдань, притаманних для своєї епохи. Більше того, саме поняття алгоритму стало предметом відповідної теорії — **теорії алгоритмів**, яка займається вивченням загальних властивостей алгоритмів. Згодом, зміст цієї теорії став настільки абстрактним, що відповідні результати були зрозумілі лише фахівцям. Тому, якийсь час мови програмування називалися алгоритмічними, а перший графічний засіб документування програм отримав назву **блок-схеми алгоритму**.

Потреби у практиці не завжди вимагали встановлення підрахування конкретних функцій чи розв'язання окремих завдань. У мовах програмування виникло і закріпилося нове поняття **процедури**, яке конкретизувало загальне поняття алгоритму стосовно вирішення завдань на комп'ютерах. Так само, як і алгоритм, процедура є закінченою послідовністю дій або операцій, спрямованих на вирішення окремого завдання. У мовах програмування з'явилася спеціальна синтаксична конструкція, яка отримала назву процедури.

Згодом, розробка великих програм перетворилася на серйозну проблему і потребувала їх розподілу на дрібніші фрагменти. Основою для такого розбиття якраз і стала процедурна декомпозиція, при якій окремі частини програми або модулі являли собою сукупність процедур для вирішення деякої сукупності задач. Головна особливість процедурного програмування полягає в тому, що програма завжди має початок у часі, або початкову

процедуру та закінчення. При цьому, візуально, вся програма може мати вигляд спрямованої послідовності графічних примітивів або блоків.

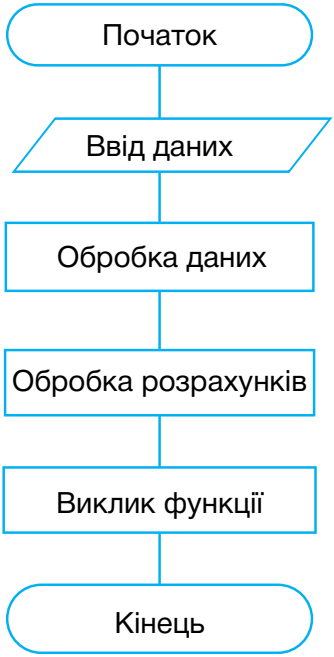

Рисунок 1. Графічне уявлення програми у вигляді послідовних процедур

Важливою властивістю таких програм є завершення всіх дій попередньої процедури для початку дій наступної процедури.

Зміна порядку виконання цих дій навіть у межах однієї процедури вимагала включення до мов програмування спеціальних умовних операторів типу if-else і goto для реалізації розгалуження обчислювального процесу в залежності від проміжних результатів розв'язання завдання.

Поява та інтенсивне використання умовних операторів, і оператора безумовного переходу, стали предметом гострих дискусій серед спеціалістів із програмування. Справа в тому, що безконтрольне застосування у програмі оператора безумовного переходу goto здатне серйозно ускладнити розуміння коду. Відповідні програми стали порівнювати зі спагеті, маючи на увазі численні переходи від одного фрагмента програми до іншого, або (що ще гірше) повернення кінцевих операторів програми до її початкових операторів. Ситуація здавалася настільки драматичною, що у літературі лунали заклики видалити оператор goto з мов програмування. Саме з того часу прийнято вважати програмування гарним стилем без використання goto.

Розглянуті ідеї сприяли становленню певної системи поглядів на процес розробки програм та написання програмних кодів, яка отримала назву **методології структурного програмування**.

Основою такої методології є процедурна декомпозиція програмної системи і організація окремих модулів у вигляді сукупності виконуваних процедур. В рамках даної методології набуло розвитку низхідне проектування програм, або програмування «зверху-вниз». Період найбільшої популярності ідей структурного програмування припадає на кінець 70-х — початок 80-х років. Як допоміжний засіб структуризації програмного коду, було рекомендоване використання відступів на початку кожного рядка, які мають виділяти вкладені цикли та умовні оператори. Все це покликане сприяти розумінню або читабельності самої програми. Ось приклад програми мовою С, що ілюструє цю особливість написання програм:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;
void main()
{
    const int n=20;
    int Mas[n];
    srand(time(0));

    for(int i=0;i<n;i++)
    {
        Mas[i] = rand()%100;
        cout<<Mas[i]<<" ";
    }
    for(int i=0;i<n;i++)
    {
        bool sort = true;
        for(int k=0;k<n-1;k++)
        {
            if(Mas[k]>Mas[k+1])
            {
                int n = Mas[k];
                Mas[k] = Mas[k+1];
                Mas[k+1]=n;
                sort=false;
            }
        }
        if(sort==true) break;
    }
    cout<<endl<<endl;
    for(int i=0;i<n;i++)
    {
        cout<<Mas[i]<<” “;
    }
}
```

У цей час основним показником складності розробки програм вважали розмір самої програми. Цілком серйозно обговорювали такі оцінки складності програм, як кількість рядків програмного коду. Щоправда, робилися деякі припущення щодо синтаксису самих рядків, які мали задовольняти певним правилам. Загальна трудомісткість розробки програм оцінювалася спеціальною одиницею виміру: «людино-місяць» або «людино-рік». А професіоналізм програміста був безпосередньо пов'язаний з кількістю рядків програмного коду, який він міг написати та налагодити, скажімо, протягом місяця.

4.2. Методологія об'єктно-орієнтованого програмування

Згодом ситуація стала суттєво змінюватися. Виявилося, що трудомісткість розробки програмних додатків на початкових етапах програмування оцінювалася значно нижче реальних зусиль. Це слугувало причиною додаткових витрат і затягування остаточних термінів готовності програм. У процесі розробки додатків змінювалися функціональні вимоги замовника, що ще більше віддаляло етап закінчення роботи програмістів. Збільшення розмірів програм призводило до необхідності залучення більшої кількості програмістів, що, своєю чергою, потребувало додаткових ресурсів для організації їх узгодженої роботи.

Але не менш важливими виявилися якісні зміни, пов'язані з усуненням акценту використання комп'ютерів. В епоху «великих машин» основними споживачами

програмного забезпечення були великі підприємства, компанії та установи, але згодом з'явилися персональні комп'ютери, як основа продажу дрібного й середнього бізнесу. Обчислювальні та розрахунково-алгоритмічні завдання у цій галузі традиційно займали другорядне місце, а на перший план виступили завдання обробки та маніпулювання даними.

Стало очевидним, що традиційні методи процедурного програмування не здатні впоратися ні зі зростанням складності програм і їх розробки, ні з необхідністю підвищення їхньої надійності. У другій половині 80-х років виникла нагальна потреба у новій методології програмування, яка була б здатна вирішити весь цей комплекс проблем. Такою методологією стало **об'єктно-орієнтоване програмування (ООП)**.

Поняття класу та об'єкта є фундаментальними поняттями ООП. При цьому, під класом розуміють деяку абстракцію сукупності об'єктів, які мають загальний набір властивостей і мають однакову поведінку. При цьому, кожен об'єкт розглядається як екземпляр відповідного класу. Об'єкти, які не мають повністю однакових властивостей або мають різну поведінку, за визначенням, не можуть належати до одного класу. Основними принципами ООП є успадкування, інкапсуляція та поліморфізм.

Наслідування — це принцип, відповідно до якого знання про загальну категорію дозволяється застосовувати для вужчої категорії.

Інкапсуляція — це приховування окремих деталей внутрішнього пристрою класів від зовнішніх щодо нього об'єктів чи користувачів. Інкапсуляція виникла

від розподілу модуля на дві частини (секції) в деяких мовах програмування: інтерфейс і реалізацію. Інтерфейс містить всю інформацію, необхідну для взаємодії з іншими об'єктами. Реалізація приховує чи маскує з інших об'єктів усі деталі, які мають відношення до процесу взаємодії об'єктів.

Поліморфізм — це здатність деяких об'єктів набувати різних зовнішніх форм залежно від обставин. Стосовно ООП поліморфізм означає, що дії, які виконуються однойменними методами, можуть відрізнятися в залежності від того, якому з класів відноситься той чи інший метод. Поліморфізм об'єктно-орієнтованих мов пов'язаний з навантаженням функцій, але не тотожний їй. Важливо пам'ятати, що імена методів і властивостей тісно пов'язані з класами, де вони описані. Це забезпечує певну надійність роботи програми, оскільки виключає випадкове застосування методу на вирішення невластивих йому завдань. Поява об'єктно-орієнтованих мов програмування була пов'язана з необхідністю реалізації концепції класів і об'єктів на синтаксичному рівні. Застосування класів у відомих мовах програмування, таких як C і Pascal, призвело до появи мов C++ і Object Pascal.

Широке поширення методології ООП вплинуло на процес розробки програм. Зокрема, процедурно-орієнтована декомпозиція програм поступилася об'єктно-орієнтованій декомпозиції, при якій окремими структурними одиницями програми виявилися не процедури і функції, а класи та об'єкти з відповідними властивостями і методами. Як наслідок, програма перестала бути послідовністю дій, визначених на етапі кодування, а стала подієво-керованою.

Остання обставина стала домінуючою під час розробки значної кількості сучасних програм. У цьому випадку кожна програма є нескінченним циклом очікування деяких, наперед визначених, подій. Ініціаторами подій можуть бути інші програми чи користувачі.

Найбільш істотною обставиною у розвитку методології ООП стало усвідомлення факту, що процес написання програмного коду може бути відділений від процесу проектування структури програми. Дійсно, перш ніж розпочати програмування класів, їх властивостей і методів, необхідно визначити, чим є самі ці класи. Більш того, потрібно дати відповіді на такі питання: скільки і які класи потрібно визначити для вирішення поставленого завдання? Які властивості і методи необхідні для надання класам необхідної поведінки? А також встановити взаємозв'язки між класами.

Ця сукупність завдань не так пов'язана з написанням коду, як із загальним аналізом вимог до майбутньої програми, а також з аналізом конкретної предметної області, для якої розробляється програма. Усі ці обставини призвели до появи спеціальної методології об'єктно-орієнтованого аналізу та проектування (ООАП).

4.3. Методологія об'єктно-орієнтованого аналізу та проектування

Необхідність аналізу предметної галузі на початок написання програми було встановлено давно, починаючи з розробки масштабних проєктів.

Виділення вихідних або базових компонентів предметної області, необхідних для вирішення того чи іншого

завдання являє нетривіальну проблему. Складність цієї проблеми проявляється у неформальному характері технік виконання або правил, які можна використовувати для цієї мети. Більше того, така робота повинна виконуватися спільно з фахівцями або експертами, яким добре відома предметна область.

Для виділення або ідентифікації компонентів предметної області було запропоновано декілька способів і правил. Сам цей процес отримав назву концептуалізації предметної галузі. При цьому, під компонентом вважають деяку абстрактну одиницю, яка наділена функціональністю, тобто, яка може виконувати певні дії, пов'язані з вирішенням поставлених завдань.

Поява методології ООАП вимагала, з одного боку, розробки різних засобів концептуалізації предметної області, а з іншого — відповідних фахівців, які володіли б цією методологією. На цьому етапі з'являється відносно новий тип спеціаліста, — аналітика (архітектора).

Поряд із фахівцями з предметної області, аналітик бере участь у побудові концептуальної схеми майбутньої програми, яка потім перетворюється програмістами у код. При цьому, окремі компоненти вибираються таким чином, щоб при наступній розробці їх було зручно уявити у формі класів і об'єктів. У цьому випадку важливе значення набуває і сама мова уявлення інформації про концептуальну схему предметної галузі.

Розподіл процесу розробки складних програм на окремі етапи сприяв становленню концепції життєвого циклу програми. Під життєвим циклом (ЖЦ) програми розуміють сукупність взаємопов'язаних та послідовних

етапів, починаючи від розробки вимог до неї і закінчуючи повною відмовою від її використання.

Згідно зі встановленими принципами, ЖЦ програми складається з наступних етапів:

- Аналіз предметної галузі та формулювання вимог до програми;
- Проектування структури програми;
- Реалізація програми у код;
- Впровадження програми;
- Супровід програми;
- Відмова від використання програми.

На етапі аналізу предметної області формулювання вимог здійснюється визначення функцій, які повинна виконувати розроблювана програма, а також концептуалізація предметної області. Цю роботу виконують аналітики разом із фахівцями предметної галузі. Результатом даного етапу має бути деяка концептуальна схема, що містить опис основних компонентів та тих функцій, які вони мають виконувати.

Етап проєктування структури програми полягає у розробці детальної схеми майбутньої програми, на якій вказуються класи, їх властивості та методи, а також різні взаємозв'язки між ними. Як правило, на цьому етапі можуть брати участь у роботі аналітики, архітектори та окремі кваліфіковані програмісти. Результатом даного етапу має стати деталізована схема програми, на якій вказуються всі класи та взаємозв'язки між ними у процесі функціонування програми. Відповідно до методології ООАП, саме така схема має бути вихідною інформацією для написання коду програми.

Результатом етапу програмування є програмний додаток, який має необхідний функціонал та вміє вирішувати потрібні завдання у конкретній предметній області.

Етапи впровадження та супроводження програми пов'язані з необхідністю налаштування та конфігурування середовища програмування, а також з усуненням помилок, які виникли у процесі її використання. Іноді, у якості окремого етапу, виділяють тестування програми, під яким розуміють перевірку роботи програми в деякій частині вихідних даних або в деяких спеціальних режимах експлуатації.

Результатом цих етапів є підвищення надійності програмного додатку, що виключає виникнення критичних ситуацій або заподіяння шкоди компанії, яка використовує цю програму.

Методологія ООАП тісно пов'язана з концепцією автоматизованої розробки програмного забезпечення (*Computer Aided Software Engineering, CASE*). Появу перших CASE-засобів зустріли з певною настороженістю. Згодом з'явилися як захоплені відгуки про їх застосування, так і критичні оцінки їх можливостей. Причин для таких суперечливих думок було декілька. Перша з них полягає в тому, що ранні CASE-засоби були простою надбудовою над деякою системою керування базами даних. Хоча візуалізація процесу розробки концептуальної схеми БД має важливе значення, вона не вирішує проблем розробки програм інших типів.

Друга причина має складнішу природу, оскільки пов'язана з графічною нотацією, реалізованою в тому чи іншому CASE-засобі. Якщо мови програмування мають строгий

синтаксис, то спроби запропонувати відповідний синтаксис для візуального представлення концептуальних схем БД сприйняли далеко неоднозначно. На цьому фоні поява уніфікованої мови моделювання (*Unified Modeling Language, UML*), орієнтована на вирішення завдань перших двох етапів ЖЦ програм була сприйнята з задоволенням.

4.4. Методологія системного аналізу та системного моделювання

Системний аналіз як науковий напрямок має більш давню історію, ніж ООП та ООАП, і власний предмет дослідження. Центральним поняттям системного аналізу є поняття системи, під якою розуміється сукупність об'єктів, компонентів або елементів довільної природи, що утворюють певну цілісність.

Визначальною передумовою виділення деякої сукупності, як системи, є виникнення у неї нових властивостей, яких не мають її складових елементів. Прикладів систем можна навести досить багато — це може бути персональний комп'ютер, автомобіль, людина, біосфера, програма та ін.

Найважливішими характеристиками будь-якої системи є її структура та процес функціонування. Під структурою системи розуміють стійку у часі сукупність взаємозв'язків між її елементами або компонентами. Саме структура пов'язує усі елементи та перешкоджає розпаду системи на окремі компоненти. Структура системи може відображати різні взаємозв'язки, зокрема і вкладені елементи однієї системи до іншої. У цьому випадку прийнято називати дрібнішу або вкладену систему підсистемою, а більшу -метасистемою. Процес функціонування системи

тісно пов'язаний зі зміною її властивостей чи поведінки у часі. При цьому, важливою характеристикою системи є її стан, під яким розуміється сукупність властивостей або ознак, які в кожний момент часу відображають найбільш суттєві особливості поведінки системи.

Процес функціонування системи відображає поведінку системи в часі і може бути представлений як послідовна зміна її станів. Якщо система змінює один свій стан на інший, то прийнято вважати, що система переходить з одного стану в інший. Сукупність ознак чи умов зміни стану системи в цьому випадку називається переходом. Методологія системного аналізу є концептуальною основою системно-орієнтованої декомпозиції предметної галузі. У цьому, випадку вихідними компонентами концептуалізації є системи та взаємозв'язки між ними. При цьому, поняття системи є більш загальним, ніж поняття класів та об'єктів у ООАП. Результатом системного аналізу є побудова деякої моделі системи чи предметної області. Важливість побудови моделей полягає у можливості їх використання для отримання інформації про властивості або поведінку системи-оригіналу. При цьому, процес побудови та подальшого застосування моделей для отримання інформації про систему-оригінал отримав назву моделювання. Найбільш загальною моделлю системи є, так звана, математична модель «чорного ящика». В цьому випадку, система представляється у вигляді прямокутника, внутрішній пристрій якого невідомий або приховано від аналітика. Проте, система не є повністю ізольованою від зовнішнього середовища, оскільки середовище накладає на систему деякі інформаційні чи матеріальні впливи.

Такі дії отримали назву вхідних впливів. У свою чергу, система також чинить на середовище (або інші системи) певні інформаційні або матеріальні впливи, які отримали назву вихідних дій. Графічно дана модель може бути зображена наступним чином:

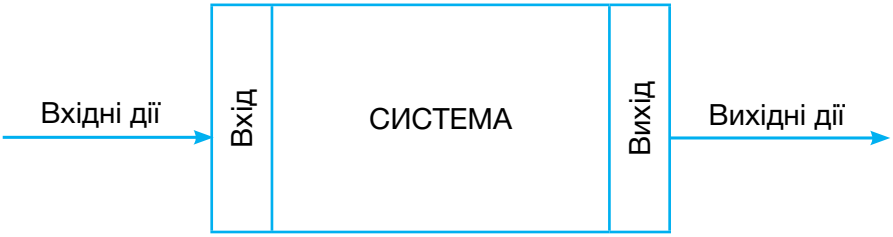

Рисунок 2.

Цінність моделей, подібних до моделі «чорного ящика», дуже умовна. Однак, загальна модель системи містить деяку важливу інформацію про функціональні особливості даної системи, які дають уявлення про її стан. Дійсно, крім найзагальнішої інформації про те, на які дії реагує система і як впливає ця реакція на навколишні об'єкти та системи, іншої інформації ми отримати не можемо.

Процес розробки адекватних моделей та їх подальшого конструктивного застосування вимагає не лише знання загальної методології системного аналізу, а також наявності відповідних образотворчих засобів або мов для фіксації результатів моделювання і їх документування. Вочевидь, природна мова не повністю підходить для цієї мети, оскільки наділена неоднозначністю та невизначеністю. Для побудови моделей були розроблені досить серйозні теоретичні методи, засновані на розвитку математичних і логічних засобів моделювання, а також запропоновані

різні формальні і графічні нотації, які відображають специфіку розв'язуваних задач. Важливо уявляти, що уніфікація будь-якої мови моделювання тісно пов'язана із методологією системного моделювання.

Складність системи та, відповідно, її моделі може бути розглянута з різних точок зору. Насамперед, можна виділити складність структури системи, яка характеризується кількістю елементів системи та різними типами взаємозв'язків між цими елементами. Другим аспектом складності є складність процесу функціонування системи. Це може бути пов'язано як з непередбачуваним характером поведінки системи, так і з неможливістю формального уявлення правил перетворення вхідних впливів у вихідні.

5. Класи та об'єкти

«Об'єкт являє собою особливий розпізнаваний предмет, блок або сутність (реальну або абстрактну), що має важливе функціональне призначення в даній предметній області» (Смітт, Токей).

За визначенням, називатимемо об'єктом поняття, абстракцію чи будь-яку річ із чітко окресленими межами, що має сенс у контексті аналізованої прикладної проблеми. Введення об'єктів переслідує дві мети:

- розуміння прикладного завдання (проблеми);
- запровадження основи реалізації на комп'ютері.

Об'єкт має стан, поведінку та індивідуальність. Об'єкти існують у часі, змінюються, мають внутрішній стан, і можуть створюватися, руйнуватися та розділятися.

Структура і поведінка подібних об'єктів визначає загальний їм клас.

Стан об'єкта характеризується переліком усіх можливих (зазвичай статичних) властивостей даного об'єкта і поточними значеннями (зазвичай динамічними) кожного з цих властивостей.

Поведінка визначається послідовністю вчинених над об'єктом дій. Поведінка характеризує те, як об'єкт впливає або піддається впливу інших об'єктів, з погляду зміни стану цих об'єктів, і передачі повідомлень. Об'єднання об'єктів у класи дозволяє ввести в завдання абстракцію і розглянути її у більш спільній постановці. Клас має ім'я, яке відноситься до всіх об'єктів цього класу. Крім

того, у класі вводяться імена атрибутів, визначених для об'єктів. У цьому сенсі, опис класу подібний до опису типу структури (запису); при цьому, кожен об'єкт має ту саму суть, як і екземпляр структури (змінна чи константа відповідного типу).

Атрибут — це значення, що характеризує об'єкт у його класі. Приклади атрибутів. Серед атрибутів різняться постійні атрибути (константи) і змінні атрибути. Постійні атрибути характеризують об'єкт у його класі (наприклад, номер рахунку, категорія, ім'я людини, тощо). Поточні значення змінних атрибутів характеризують поточний стан об'єкта (наприклад, баланс рахунку, вік людини і т.п.). Змінюючи значення цих атрибутів, ми змінюємо і стан об'єкта.

5.1. Операції над об'єктом

Операція — це функція (або перетворення), яку можна застосовувати до об'єктів класу.

Усі об'єкти даного класу використовують один і той же екземпляр кожної операції (тобто, збільшення кількості об'єктів деякого класу не призводить до збільшення кількості завантаженого коду програми). Об'єкт, із якого викликана операція, передається їй у вигляді її неявного аргументу (параметра this).

Одна й та сама операція може застосовуватися до об'єктів різних класів: така операція називається поліморфною, оскільки вона може мати різні форми для різних класів. Наприклад, для об'єктів класів вектор і комплексну кількість можна визначити операцією +. Така операція буде поліморфною, оскільки складання

векторів та складання комплексних чисел вважаються різними операціями.

Кожній операції відповідає метод — реалізація цієї операції для об'єктів цього класу. Отже, операція — це специфікація методу, а метод — реалізація операції.

Операція (і методи, які її реалізують) визначається своєю сигнатурою, яка включає, крім імені операції, типи (класи) усіх її аргументів, і тип (клас) результату (повертається значення). Усі методи, які реалізують операцію, повинні мати таку ж сигнатуру, як і реалізована ними операція.

При моделюванні системи, корисно розрізняти операції, які мають побічні ефекти (ці ефекти виражаються у зміні значень атрибутів об'єкта, тобто в зміні його стану), та операції, які видають потрібне значення, не змінюючи стан об'єкта. Ці операції називаються запитами.

Значення деяких атрибутів об'єкта можуть бути доступними лише для операцій цього об'єкта. Запити без аргументів (за винятком неявного аргументу — об'єкта, до якого застосовується операція) можуть розглядатися як похідні атрибути.

5.2. Види найпоширеніших операцій

1. **Модифікатор** — змінює стан об'єкта шляхом запису або доступу.
2. **Селектор** — дає доступ для визначення стану об'єкта без його зміни (операція зчитування).
3. **Ітератор** — можливість доступу до частин об'єкта у певній послідовності.

4. **Конструктор** — створення або ініціалізація об'єкта.
5. **Деструктор** — руйнування об'єкта або звільнення зайнятої ним пам'яті.

5.3. Зв'язки між об'єктами

Між об'єктами можна встановлювати залежності за даними. Ці залежності виражають зв'язки (або відносини) між класами зазначених об'єктів. Відносини двох будь-яких об'єктів ґрунтуються на знаннях, які вони мають один про одного: про методи, які можна виконувати, і про очікувану поведінку.

Типи відносин між об'єктами:

1. Зв'язки.
2. Агрегація.

У типі відносин «Зв'язки», об'єкти можуть виконувати різні ролі:

Актор, або активний об'єкт (*actor*) — об'єкт може впливати, але ніколи не схильний до впливу (активний об'єкт).

Сервер, або виконавець (*server*) — піддається впливу з боку інших об'єктів, але ніколи не активний.

Агент, або посередник (*agent*, *broker*) — виконує роль як актора, так і сервера, як правило, створюється на користь активного об'єкта.

Зв'язок типу «Агрегація» — це відношення типу частина-ціле (*A part-of*). Дозволяє, виходячи з цілого (агрегата), увійти до його частин (атрибутів). За змістом, агрегація може означати фізичне входження одного об'єкта до іншого (склад або структуру об'єкта), але може бути

й інша семантика, наприклад, приналежність одного об'єкта до іншого.

5.4. Клас

Клас — це множина об'єктів, пов'язаних спільністю структури і поведінки.

Як правило, виділяють інтерфейс (видимий для всіх зовнішній вигляд і набір підтримуваних методів) і реалізацію класу (внутрішній пристрій, захищений від інших).

Для завдання класу необхідно вказати ім'я цього класу, а потім перерахувати його атрибути та операції (або методи). З кожним об'єктом пов'язана структура даних, полями якої є атрибути цього об'єкта та покажчики функцій (фрагментів коду), що реалізують операції цього об'єкта (зазначимо, що покажчики функцій в результаті оптимізації коду зазвичай замінюються на звернення до цих функцій). Таким чином, об'єкт — це деяка структура даних, тип якої відповідає класу цього об'єкта.

Іноді у підкласі буває необхідно перевизначити операцію, визначену в одному із суперкласів. Для цього операція, яка може бути отримана з суперкласу в результаті успадкування, визначається і в підкласі. Це її повторне визначення «заступає» її визначення у суперкласі так, що у підкласі застосовується не успадкована, а перевизначена у ньому операція.

Нагадаємо, що кожна операція визначається своєю сигнатурою. Отже, сигнатура перевизначення операції має збігатися з сигнатурою операції з суперкласу, яка перевизначається цією операцією.

Перевизначення може переслідувати одну з таких цілей:

- **розширення**: нова операція розширює успадковану операцію з огляду на вплив атрибутів підкласу;
- **обмеження**: нова операція обмежується виконанням частини дій успадкованої операції, використовуючи специфіку об'єктів підкласу;
- **оптимізація**: використання специфіки об'єктів підкласу дозволяє спростити та прискорити відповідний метод;
- зручність.

Відносини між класами реалізовані у мовах програмування:

- **Асоціація** — смисловий або семантичний зв'язок між класами. Зв'язок один-до-одного, один-до-багатьох, багато-до-багатьох.
- **Узагальнення та успадкування** дозволяють виявити аналогії між різними класами об'єктів, що визначають багаторівневу класифікацію об'єктів. Так, у графічних системах можуть існувати класи, що визначають опис різних геометричних фігур: точок, ліній (прямих, дуг кіл і кривих, що визначаються сплайнами), багатокутників, кіл тощо.

Спосіб реалізації — делегування, коли об'єкти розглядаються як прототипи (зразки), які делегують свою поведінку іншим об'єктам, обмежуючи потребу у створенні нових класів. Реалізує відношення спільності та асоціативності. Виділяють просте та множинне успадкування.

6. Екскурс у діаграми

Під структурним системним аналізом заведено розуміти метод дослідження системи, який починається з найбільш загального опису з подальшою деталізацією уявлення окремих аспектів її поведінки та функціонування. При цьому загальна модель системи будується у вигляді деякої ієрархічної структури, яка відбиває різні рівні абстракції з обмеженою кількістю компонентів на кожному з рівнів. Одним із головних принципів структурного системного аналізу є виділення на кожному з рівнів абстракції лише найістотніших компонентів або елементів системи.

У рамках програмної інженерії прийнято розглядати три графічні нотації, що отримали назву діаграм: діаграми «сутність-зв'язок» (*Entity-Relationship Diagrams, ERD*), діаграми функціонального моделювання (*Structured Analysis and Design Techique, SADT*) та діаграми потоків даних (*Data Flow Diagrams, DFD*).

6.1. Діаграми «сутність-зв'язок»

Діаграми «сутність-зв'язок» призначені для графічного уявлення моделей даних розроблюваної програмної системи і пропонують деякий набір стандартних позначень для визначення даних і відносин між ними. За допомогою цього виду діаграм можна описати окремі компоненти концептуальної моделі даних і сукупність взаємозв'язків між ними, які мають важливе значення для розроблюваної системи.

Основними поняттями цієї нотації є поняття сутності та зв'язку. При цьому, під сутністю (*entity*) мають на увазі довільну безліч реальних або абстрактних об'єктів, кожен з яких має однакові властивості та характеристики. У цьому випадку кожен аналізований об'єкт може бути екземпляром однієї і тільки однієї сутності, повинен мати унікальне ім'я або ідентифікатор, а також відрізнятися від інших екземплярів такої сутності.

Прикладами сутностей можуть бути: банк, клієнт банку, рахунок клієнта, аеропорт, пасажир, рейс, комп'ютер, термінал, автомобіль, водій. Кожна з сутностей може розглядатися з різною мірою деталізації і на різному рівні абстракції, що визначається конкретною постановкою завдання. Для графічного уявлення сутностей використовуються спеціальні позначки:

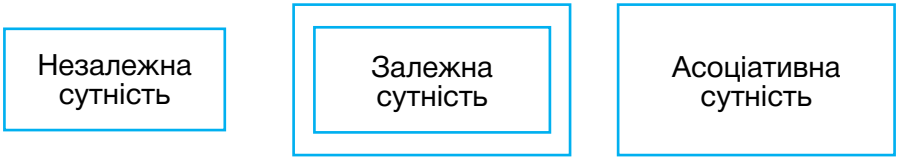

Рисунок 3.

Зв'язок (*relationship*) визначається як відношення або деяка асоціація між окремими сутностями. Прикладами зв'язків можуть бути родинні стосунки типу «батько-син» або виробничі відносини типу «начальник-підлеглий». Інший тип зв'язків задається відносинами «мати у власності» або «володіти властивістю». Різні типи зв'язків графічно зображуються у вигляді ромба з відповідним ім'ям цього зв'язку.

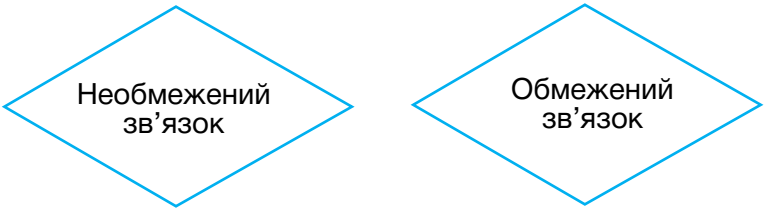

Рисунок 4.

Графічна модель даних будується таким чином, щоб зв'язки між окремими сутностями відображали не лише семантичний характер відповідного відношення, але й додаткові аспекти обов'язковості зв'язків, а також кратність екземплярів сутностей, що беруть участь у даних відносинах.

Розглянемо на простому прикладі ситуацію, яка описується двома сутностями: «Співробітник» і «Компанія». При цьому, в якості зв'язку можливе використання ставлення приналежності співробітника цієї компанії. Якщо враховувати те, що в компанії працюють декілька співробітників і вони не можуть бути працівниками інших компаній, то ця інформація може бути представлена графічно у вигляді наступної діаграми «сутність-зв'язок»:

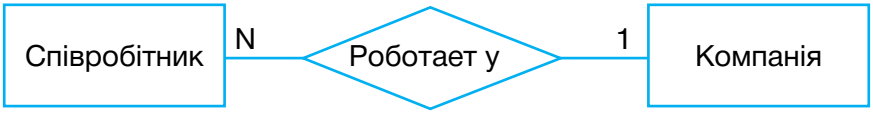

Рисунок 5.

На цьому зображенні літера «N» біля зв'язку означає те, що в компанії можуть працювати більше одного працівника, причому значення N не фіксується заздалегідь.

«1» з іншого кінця зв'язку означає, що співробітник може працювати лише в одній компанії. Тобто, не допускається прийом на роботу працівників за сумісництвом з інших компаній або установ.

Трохи інша ситуація складається у випадку розгляду сутностей «працівник» і «проєкт», і зв'язку «бере участь у роботі над проєктом».

Рисунок 6.

Один працівник може брати участь у розробці кількох проєктів, а в розробці одного проєкту можуть брати участь декілька працівників. Тобто, цей зв'язок є багатозначним. Цей факт спеціально відображається на діаграмі літерами «N» і «M» поряд з відповідними сутностями, причому вибір конкретних букв не є важливим.

Розглянуті дві діаграми можуть бути об'єднані в одну, де буде представлена інформація про співробітників компанії, які беруть участь у розробці проєктів даної компанії. У цьому випадку, може бути запроваджено додатковий зв'язок, що характеризує проєкти цієї компанії.

Обмеженість діаграм «сутність-зв'язок» проявляється при конкретизації концептуальної моделі в більш детальний прояв модельованої програмної системи. Такий прояв, крім статичних зв'язків, має містити інформацію про поведінку або функціонування окремих її компонентів. Для таких цілей використовується інший тип діаграм, які отримали назву діаграм потоків даних.

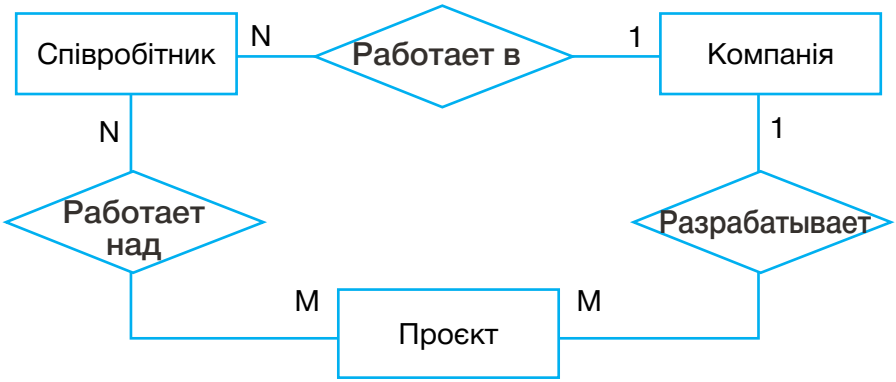

Рисунок 7.

6.2. Діаграми функціонального моделювання

Початок розробки діаграм функціонального моделювання належить до середини 1960-х, коли Дуглас Т. Рос запропонував спеціальну техніку моделювання, яка отримала назву SADT (*Structured Analysis & Design Technique*). Військово-повітряні сили США використали методику SADT як частину своєї програми інтеграції комп'ютерних і промислових технологій (*Integrated Computer Aided Manufacturing*, ICAM) і назвали її IDEF0 (*Icam DEFinition*). Метою програми ICAM було збільшення ефективності комп'ютерних технологій у сфері проєктування нових засобів озброєнь та ведення бойових дій. Одним із результатів цих досліджень був висновок про те, що описові мови не ефективні для документування і моделювання процесів функціонування складних систем. Подібні описи не забезпечують потрібного рівня несуперечності та повноти, які мають домінуюче значення при вирішенні завдань моделювання.

В рамках програми **ICAM** було розроблено декілька графічних мов моделювання, які отримали такі назви:

- Нотація **IDEF0** — для документування процесів виробництва і відображення інформації про використання ресурсів на кожному з етапів проєктування систем.
- Нотація **IDEF1** — для документування інформації про виробниче оточення систем.
- Нотація **IDEF2** — для документування поведінки системи у часі.
- Нотація **IDEF3** — спеціально для моделювання бізнес-процесів.

Нотація IDEF2 ніколи не була повністю реалізована. Нотація IDEF1 в 1985 році була розширена і перейменована на IDEFIX. Для методології IDEF-SADT знайшли застосування в урядових та комерційних організаціях, оскільки на той період часу цілком задовольняла різним вимогам, які представлені до моделювання широкого класу систем.

На початку 1990 року спеціально утворена група користувачів IDEF (*IDEF User Group*), у співпраці з Національним інститутом зі стандартизації і технології США (*National Institutes for Standards and Technology*, NIST), зробила спробу створення стандарту для IDEF0 і IDEFIX. Ця спроба виявилася успішною і завершилася прийняттям у 1993 році стандарту уряду США, відомого як FIPS для двох технологій: IDEF0 і IDEFIX. Протягом наступних років цей стандарт продовжував активно розвиватися та сприяв основою для реалізації в деяких перших CASE-засобах.

Методологія IDEF-SADT являє собою сукупність методів, правил та процедур, призначених для побудови функціональної моделі системи будь-якої предметної галузі. Функціональна модель SADT відображає структуру процесів функціонування системи і її окремих підсистем, тобто, виконувані ними дії та зв'язки між цими діями. З цією метою будуються спеціальні моделі, які дозволяють у наочному вигляді представити послідовність певних дій. Вихідними будівельними блоками будь-якої моделі IDEF0 процесу є активність (*activity*) та стрілки (*arrows*).

Розглянемо основні поняття методології IDEF-SADT, які застосовуються при побудові діаграм функціонального моделювання. Активність являє собою деяку дію або набір дій, які мають фіксовану мету і призводять до деякого кінцевого результату. Іноді активність називають процесом. Моделі IDEF0 відстежують різні види активності системи, їх опис і взаємодію з іншими процесами. На діаграмах активності або процес зображуються прямокутником, який називається блоком. Стрілка слугує для позначення деякого носія або впливу, які забезпечують перенесення даних чи об'єктів від однієї діяльності до іншої.

Стрілки також необхідні для опису того, що саме здійснює процес і які ресурси вона споживає.

Це так звані ролі стрілок — ICOM — скорочення перших літер від назв відповідних стрілок IDEF0.

При цьому, розрізняють стрілки чотирьох видів:

- **I** (*Input*) — вхід, тобто все, що надходить у процес або споживається процесом.

- **C** (*Control*)— керування або обмеження виконання операцій процесу.
- **O** (*Output*) — вихід або результат процесу.
- **M** (*Mechanism*) — механізм, який використовується для виконання процесу.

Методологія IDEF0 однозначно визначає, як зображуються на діаграмах стрілки кожного виду ICOM. Стрілка Вхід (*Input*) виходить із лівої сторони рамки робочого поля і входить зліва у прямокутник процесу. Стрілка Керування (*Control*) входить і виходить зверху. Стрілка Вихід (*Output*) виходить із правої сторони процесу та входить з правої сторони рамки. Стрілка Механізм (*Mechanism*) входить у прямокутник процесу знизу. Таким чином, базове уявлення процесу на діаграмах IDEF0 має такий вигляд:

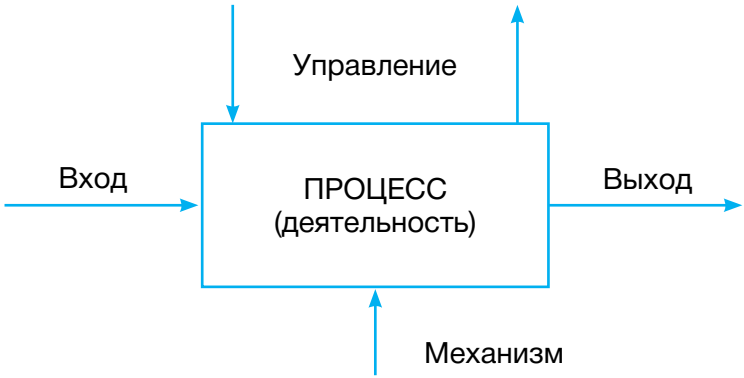

Рисунок 8.

Техніка побудови діаграм є головною особливістю методології IDEF-SADT. Місце з'єднання стрілки із блоком визначає тип інтерфейсу. При цьому, усі функції

модельованої системи та інтерфейси на діаграмах подаються у вигляді відповідних блоків процесів та стрілок ICOM. Керуюча інформація входить до блоку зверху, а оброблювана інформація, зображується зліва від блоку. Результати процесу зображуються як виходи процесу і позначаються з правої сторони блоку. Як механізм реалізації такої операції, можуть виступати людина або автоматизована система. Відповідний механізм на діаграмі позначається стрілкою, яка входить до блоку процесу знизу.

Однією з найважливіших особливостей методології IDEF-SADT є поступове введення все більш детальних уявлень моделі системи в ході розробки окремих діаграм. Побудова моделі IDEF-SADT починається з уявлення всієї системи у вигляді найпростішої діаграми, що складається з одного блоку процесу і стрілок ICOM, потрібних для зображення основних видів взаємодії з об'єктами поза системою. Оскільки вихідний процес представляє всю систему як єдине ціле, таке уявлення є найбільш загальним та підлягає подальшій декомпозиції.

Для ілюстрації основних ідей методології IDEFSADT розглянемо такий простий приклад. В якості процесу уявімо діяльність з оформлення кредиту в банку. Входом даного процесу є заявка від клієнта на отримання кредиту, а виходом — відповідний результат, тобто — кредит. При цьому, керуючими факторами є правила оформлення кредиту, які регламентують умови отримання відповідних фінансових коштів в кредит. Механізмом даного процесу є працівник банку, який уповноважений виконати усі операції щодо оформлення кредиту.

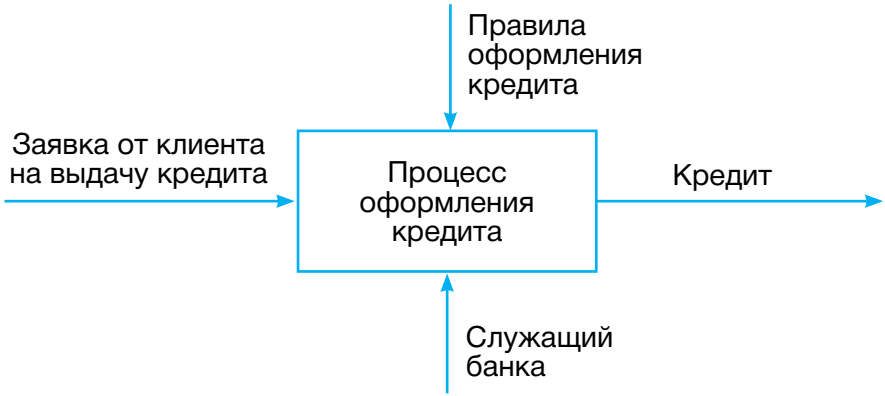

Рисунок 9.

Зрештою, модель IDEF-SADT являє собою серію ієрархічно взаємопов'язаних діаграм із супровідною документацією, яка розподіляє початкове уявлення складної системи на окремі складові частини. Деталі кожного з основних процесів видаються у вигляді більш детальних процесів в інших діаграмах. У цьому випадку кожна діаграма нижнього рівня є декомпозицією деякого процесу більш загальної діаграми. Тому, на кожному етапі декомпозиції більш загальна діаграма конкретизується на більш детальних діаграмах.

В даний час, діаграми структурного системного аналізу IDEF-SADT продовжують використовуватися цілою низкою організацій для побудови та детального аналізу функціональної моделі існуючих на підприємстві бізнес-процесів, а також для розробки нових бізнес-процесів. Основний недолік даної методології пов'язаний з відсутністю явних засобів для об'єктно-орієнтованого уявлення моделей складних систем. Хоча, деякі аналітики відзначають важливість знання та застосування нотації

IDEF-SADT, обмежені можливості цієї методології стосовно реалізації відповідних графічних моделей в об'єктно-орієнтованому програмному коді істотно звужують діапазон розв'язуваних з її допомогою задач.

6.3. Діаграми потоків даних

Основою даної методології графічного моделювання інформаційних систем є спеціальна технологія побудови діаграм потоків даних DFD.

Модель системи у контексті DFD має вид деякої інформаційної моделі, основними компонентами якої є різні потоки даних, які передають інформацію від однієї підсистеми до іншої. Кожна з підсистем виконує певні перетворення вхідного потоку даних та передає результати обробки інформації у вигляді потоків даних для інших підсистем.

Основними компонентами діаграм потоків даних є:

- Зовнішні сутності;
- Процеси;
- Системи/підсистеми;
- Накопичувачі даних або сховища;
- Потоки даних.

Зовнішня сутність є матеріальним об'єктом або фізичною особою, які можуть виступати як джерело або отримувача інформації. Визначення деякого об'єкта чи системи як зовнішньої сутності не є суворо фіксованим.

Хоча зовнішня сутність знаходиться за межами аналізованої системи, у процесі подальшого аналізу деякі зовнішні сутності можуть бути перенесені у діаграму

моделі системи. З іншого боку, окремі процеси можуть бути винесені за межі діаграми і представлені як зовнішні сутності. Прикладами зовнішніх сутностей можуть бути: клієнти організації, замовники, персонал, постачальники. Зовнішня сутність позначається прямокутником з тінню, усередині якого вказується її ім'я. При цьому, для позначення ім'я рекомендують використовувати іменник у називному відмінку. Іноді, зовнішню сутність називають також термінатором.

Рисунок 10.

Процес є сукупністю операцій з перетворення вхідних потоків даних у вихідні відповідно до певного алгоритму або правила. Хоча фізично, процес може бути реалізований різними способами. Найчастіше мається на увазі програмну реалізацію процесу. Процес на діаграмі потоків даних зображують прямокутником із закругленими вершинами, розділеними на три секції або поля горизонтальними лініями.

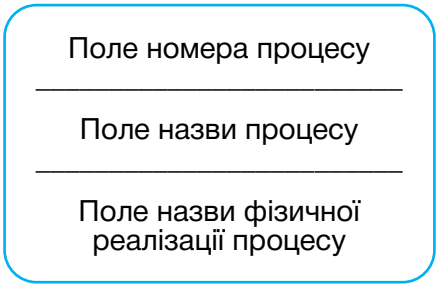

Рисунок 11.

Поле номера процесу слугує для ідентифікації процесу. У середньому полі вказується назва процесу. Для назви рекомендовано використовувати дієслово в невизначеній формі з необхідними доповненнями. Нижнє поле містить вказівку на спосіб фізичної реалізації процесу.

Інформаційна модель системи будується як деяка ієрархічна схема у вигляді так званої контекстної діаграми, на якій вихідна модель послідовно зображується як модель підсистем відповідних процесів перетворення даних. При цьому, підсистема або система на контекстній діаграмі DFD зображується так само, як і процес — прямокутник із закругленими вершинами.

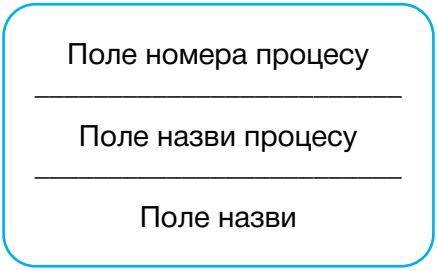

Рисунок 12.

Накопичувач даних, або сховище, являє собою абстрактний пристрій або спосіб зберігання інформації, що переміщається між процесами. Передбачається, що дані можна будь-якої миті помістити в накопичувач і через деякий час витягти, причому фізичні способи розміщення та вилучення даних можуть бути довільними. Накопичувач даних може бути фізично реалізований різними способами, але найчастіше передбачається його реалізація в електронному вигляді на магнітних носіях.

Накопичувач даних на діаграмі потоків даних є прямокутником з двома полями. В першому полі вказується номер або ідентифікатор накопичувача, який починається з літери «D». В другому полі зазначається назва. При цьому, для назви накопичувача рекомендовано використовувати іменник, який характеризує спосіб зберігання відповідної інформації

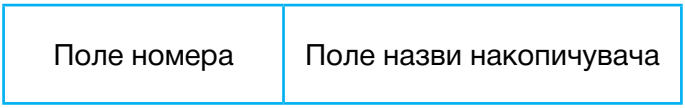

Рисунок 13.

Потік даних визначає якісний характер інформації, що передається через з'єднання джерела до отримувача. Потік даних може передаватися мережею між двома комп'ютерами або будь-яким іншим способом, який припускає вилучення даних та їх відновлення у потрібному форматі. Потік даних на діаграмі DFD зображується лінією зі стрілкою на одному з її кінців, при цьому стрілка показує напрямок потоку даних. Кожен потік даних має свою власну назву, що відбиває зміст потоку. Таким чином, інформаційна модель системи в нотації DFD будується у вигляді діаграм потоків даних, які графічно представляються з використанням відповідної систем позначень. Розглянемо на прикладі спрощену модель отримання деякої суми готівкою через кредитну картку клієнтом банку.

Зовнішніми сутностями цього прикладу є клієнт банку та, можливо, працівник банку, який контролює процес обслуговування клієнтів. Накопичувачем даних

може бути база даних про стан рахунків окремих клієнтів банку. Окремі потоки даних відображають характер необхідної інформації, яка передається для обслуговування клієнта банку. Відповідна модель для цього прикладу може бути представлена наступним чином:

Рисунок 14.

В даний час діаграми потоків даних використовуються в деяких CASE — засобах для побудови інформаційних моделей систем обробки даних. Основний недолік цієї методології також пов'язаний з відсутністю явних засобів для об'єктно-орієнтованого уявлення моделей складних систем, а також для уявлення складних алгоритмів обробки даних. Оскільки на діаграмах DFD не вказуються характеристики часу виконання окремих процесів і передачі даних між процесами, то моделі систем, які

реалізують синхронну обробку даних, не можуть бути правильно представлені у нотації DFD. Усі ці особливості методології структурного системного аналізу обмежили можливості її широкого застосування і посприяли включенню відповідних засобів до уніфікованої мови моделювання.

7. Історія розвитку мови UML

Окремі мови об'єктно-орієнтованого моделювання стали з'являтися в середині 1970-х — кінці 1980-х років, коли різні дослідники та програмісти пропонували свої підходи до ООАП (методологія об'єктно-орієнтованого аналізу та проектування). У період між 1989–1994 роками загальна кількість найбільш відомих мов моделювання зросла з 10 до 50. Багато користувачів відчували складнощі при виборі мови ООАП, оскільки жодна з них не задовольняла усіх вимог, що висуваються до побудови моделей складних систем. Прийняття окремих методик та графічних нотацій як стандартів (IDEF0, IDEFIX) не дало змоги змінити ситуацію непереможної конкуренції між ними на початку 90-х років.

До середини 1990-х деякі з методів були суттєво покращені і набули самостійного значення при розв'язанні різних завдань ООАП. Найбільш відомими в цей період стають:

- Метод Граді Буча (*Grady Booch*), який отримав умовну назву Booch.
- Метод Джеймса Румбаха (*James Rumbaugh*), який отримав назву Object Modeling Technique — OMT.
- Метод Айвара Джекобсона (*Ivar Jacobson*), який отримав назву Object-Oriented Software Engineering — OOSE.

Кожен із цих методів був орієнтований на підтримку окремих етапів ООАП. Наприклад, метод OOSE містив засоби представлення варіантів використання, які мають

суттєве значення на етапі аналізу вимог у процесі проєктування бізнес-додатків. Метод OMT найбільш підходив для аналізу процесів обробки даних в інформаційних системах. Метод Booch знайшов найбільше застосування на етапах проєктування і розробки різних програмних систем.

Історія розвитку мови UML бере початок з жовтня 1994 року, коли Граді Буч і Джеймс Румбах з Rational Software Corporation розпочали роботу з уніфікації методів Booch і OMT. Хоча ці методи і були досить популярними, спільна робота була спрямована на вивчення усіх відомих об'єктно-орієнтованих методів з метою об'єднання їх переваг. При цьому, Г. Буч і Дж. Румбах зосередили зусилля на повній уніфікації результатів своєї роботи. Проєкт так званого уніфікованого методу (*Unified Method*) був підготовлений та опублікований у жовтні 1995 року. Восени цього року до них приєднався А. Джекобсон, головний технолог із компанії Objectory AB (Швеція) з метою інтеграції свого методу OOSE з двома попередніми.

Спочатку автори методів Booch, OMT та OOSE передбачали розробити уніфіковану мову моделювання лише для цих методик. З одного боку, кожен з цих методів був перевірений на практиці і показав свою конструктивність під час вирішення окремих завдань ООАП.

Це давало підставу для подальшої їх модифікації на основі усунення наявної невідповідності окремих понять та позначень. З іншого боку, уніфікація семантики і нотації повинна була забезпечити деяку стабільність на ринку об'єктно-орієнтованих CASE-засобів, необхідних для успішного просування відповідних програмних

інструментаріїв. Зрештою, спільна робота давала надію на суттєве поліпшення всіх трьох методів.

Починаючи роботу з уніфікації своїх методів, Г. Буч, Дж. Румбах та А. Джекобсон сформулювали такі вимоги до мови моделювання. Вона має:

- Дозволяти моделювати не лише програмне забезпечення, але й ширші класи систем і бізнес-додатків з використанням об'єктно-орієнтованих понять.
- Явним чином забезпечувати взаємозв'язок між базовими поняттями для моделей концептуального і фізичного рівнів.
- Забезпечувати масштабування моделей, що є важливою особливістю складних багатоцільових систем.
- Бути зрозумілою аналітикам і програмістам, а також має підтримуватися спеціальними інструментами і засобами, реалізованими на різних комп'ютерних платформах.

Розробка системи позначок або нотації для ООАП виявилася несхожою на розробку нової мови програмування. По-перше, необхідно було вирішити дві проблеми:

- Чи має ця нотація містити специфікацію вимог?
- Чи слід розширювати цю нотацію до рівня мови візуального програмування?

По-друге, необхідно було знайти вдалий баланс між виразністю та простотою мови. З одного боку, надто проста нотація обмежує коло потенційних проблем, які можуть бути вирішені за допомогою відповідної системи позначок. З іншого боку, надто складна нотація створює додаткові труднощі для її вивчення і застосування

аналітиками і програмістами. У разі уніфікації існуючих методів, необхідно враховувати інтереси фахівців, які вже мають досвід роботи з ними, оскільки внесення серйозних змін до нової нотації може призвести до нерозуміння і неприйняття її користувачами старих методик. Щоб виключити неявний опір з боку окремих фахівців, необхідно враховувати інтереси найширшого кола користувачів. Наступна робота над мовою UML мала враховувати усі ці обставини.

У цей період підтримка розробки мови UML стає однією з цілей консорціуму OMG (*Object Management Group*). Хоча консорціум OMG утворений ще в 1989 році з метою розробки пропозицій щодо стандартизації об'єктних і компонентних технологій CORBA, мова UML набула статусу другого стратегічного напряму в роботі OMG. Саме в рамках OMG створюється команда розробників під керівництвом

Річарда Солі, яка забезпечуватиме подальшу роботу з уніфікації та стандартизації мови UML. У червні 1995 року, OMG організувала нараду усіх вагомих фахівців і представників, що входять до консорціуму компаній з методології ООАП, на якому вперше в міжнародному масштабі було визнано доцільність пошуку індустріальних стандартів у галузі мов моделювання під егідою OMG.

Зусилля Г. Буча, Дж. Румбаха і А. Джекобсона привели до появи перших документів, в яких міститься опис саме мови UML версії 0.9 (червень 1996 р.) та версії 0.91 (жовтень 1996). Ці документи, що мали статус запиту пропозицій RFP (*Request For Proposal*), слугували своєрідним каталізатором для широкого обговорення мови UML

різними категоріями спеціалістів. Перші відгуки та реакція на мову UML вказували на необхідність її доповнення окремими поняттями та конструкціями.

У той же час стало зрозуміло, що деякі компанії і організації бачать у мові UML лінію стратегічних інтересів для свого бізнесу. Компанія Rational Software разом із декількома організаціями, які виявили бажання виділити ресурси для розробки версії 1.0 мови UML, започаткувала консорціум партнерів UML.

Компанії, які увійшли до цього консорціуму, забезпечили підтримку подальшої роботи з більш точного і суворого визначення нотації, що призвело до появи версії 1.0 мови UML. У січні 1997 року, було опубліковано документ з описом мови UML 1.0 як початковий варіант відповіді на запит пропозицій RFP. Ця версія мови моделювання була досить добре визначена, забезпечувала необхідну виразність і потужність, і передбачала розв'язання широкого спектру задач.

Інструментальні CASE-засоби та діапазон їх практичного застосування більшою мірою залежать від вдалого визначення семантики і нотації відповідної мови моделювання. Специфіка мови UML полягає у тому, що він визначає семантичну метамодель, а не модель конкретного інтерфейсу і засоби уявлення або реалізації компонентів.

У січні 1997 року, низку інших компаній представили на розгляд до OMG свої власні відповіді на запит пропозицій від RFP. Надалі, ці компанії приєдналися до партнерів UML, пропонуючи включити до мови UML деякі свої ідеї. В результаті спільної роботи з партнерами UML була запропонована переглянута версія 1.1 мови

UML. Особлива увага при розробці мови UML 1.1 було приділено досягненню більшої ясності семантики мови у порівнянні з UML 1.0, а також обліку пропозицій нових партнерів. Ця версія мови була представлена на розгляд OMG і була схвалена і сприйнята як стандарт OMG у листопаді 1997 року.

Черговий етап розвитку цієї мови закінчився в березні 1999 року, коли консорціум OMG був опублікований опис мови UML 1.3.

Статус мови UML визначений відкритим для усіх пропозицій щодо її доопрацювання і вдосконалення. Сама мова UML не є чиєюсь власністю і не є запатентованою. У той же час, абревіатура UML є торговою маркою їх законних власників.

Мова UML орієнтована на застосування як мова моделювання різними користувачами і науковими спільнотами для вирішення широкого класу завдань ООАП.

Урок №1
Введення в об’єктно-орієнтований аналіз та проектування