ТЕОРЕТИЧНІ ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ 1

Від процедурного програмування до об'єктного 1
Поняття об'єктно-орієнтованого програмування 4
Основні принципи ООП 7
Основні поняття ООП 11
Основні механізми ООП 13
Сучасні об'єктно-орієнтовані мови програмування 22
Основи об'єктно-орієнтованого підходу до розроблення складних програм 17
Резюме ..... **Error! Bookmark not defined.**

ТЕОРЕТИЧНІ ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

Від процедурного програмування до об'єктного

Ще донедавна традиційною при розробці програмного забезпечення була імперативна парадигма програмування. Вона базується на алгоритмічній декомпозиції задачі (рис. 1 а, б) і передбачає відокремлення у програмах даних і функцій їхньої обробки (рис. 2).

а)

б)

Рис. 1. Алгоритмічна декомпозиція системи платіжної відомості
а) верхнього рівня
б) детальна

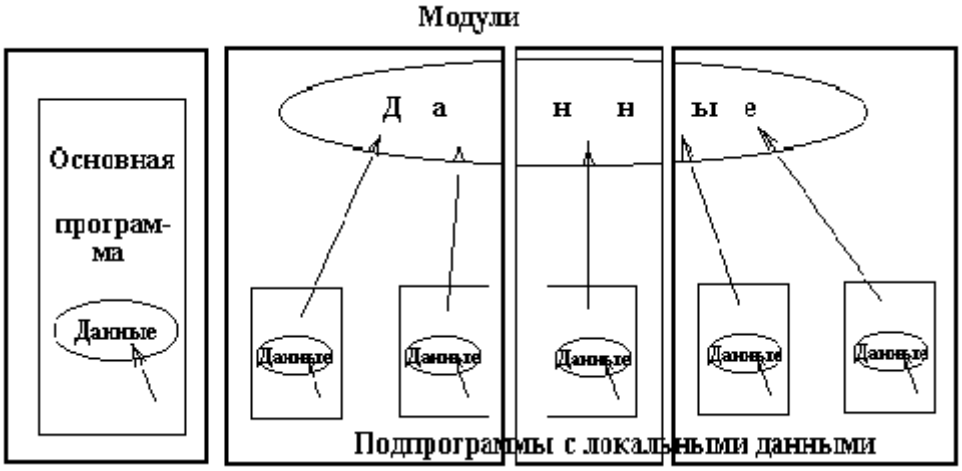

Рис. 2. Архітектура програми, що складається з модулів

Процедурно-орієнтована програма структурується таким чином, що обов'язково є головна функція і декілька інших функцій, які викликаються з головної. Головна функція, як правило, невелика, і основна робота покладена на інші функції. Програма починається з першого оператора головної функції і закінчується останньою командою цієї ж функції.

Такий підхід має орієнтацію на дію. Функції (або процедури) є основною програмною одиницею. Сукупність дій, які виконують деяку спільну задачу, оформлюються у вигляді функцій, а сукупність функцій створює програму. При такому підході дані існують для підтримки виконання дій функціями. Відповідно, технологія структурного проектування програм, перш за все, приділяє увагу розробці алгоритму.

Як показує практика, структурний підхід в поєднанні з модульним програмуванням дозволяє отримувати досить надійні програми. Однак при збільшенні розміру програми зазвичай зростає складність міжмодульних інтерфейсів (рис. 3), і передбачити взаємовплив окремих частин програми стає практично неможливо.

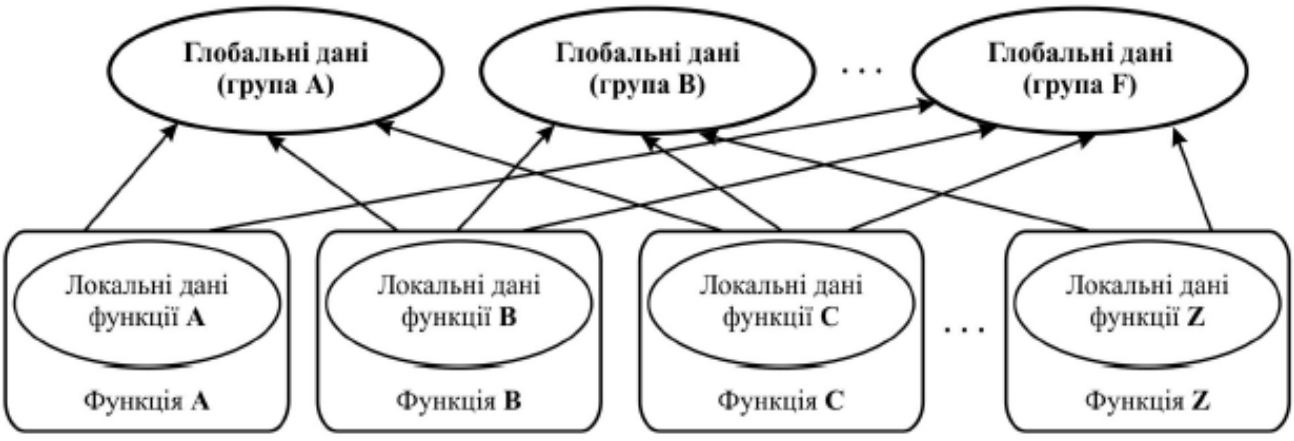

Рис. 3. Структурний підхід до встановлення зв'язків між глобальними даними і функціями програми

Це призводить до неконтрольованого доступу до даних, а також до ускладнення можливості модифікації програми.

Наприклад, при розробці програми ведення складського обліку матеріальних цінностей існує потреба сумісного використання деяких даних, зокрема,

закупівельної ціни та наявної кількості матеріалів на складі, декількома функціями. Такі дані мають бути оголошені як глобальні. Різні функції отримують доступ до глобальних даних для виконання різних операцій: створення нового запису в обліковій книзі, виведення запису на екран, зміни полів наявного запису і т.ін.

Будь-яка зміна структури глобальних даних може вимагати коректування всіх функцій, які використовують ці дані. Зокрема, якщо розробник вирішить зробити заміну деяких глобальних даних з 6-значного коду на 10-значний, то необхідно змінити відповідний тип даних з **`short`** на **`long`**. Це означає, що в усіх функціях, які оперують цими даними, потрібно внести зміни у локальні дані, тобто оголосити їх типом **`long`**.

Коли зміни вносяться в глобальні дані великих програм, то не завжди можна швидко визначити, які функції при цьому необхідно скоректувати. Навіть тоді, коли це вдається зробити оперативно, то згодом через значну кількість зв'язків між функціями і даними виправлені функції починають некоректно працювати з іншими глобальними даними. Таким чином, будь-яка зміна у коді програми призводить до виникнення негативних подальших дій і, як наслідок, появи відповідних проблем.

Методологія імперативного програмування має ще одне, досить суттєве обмеження: відокремлення даних від функцій виявляється малопридатним для достовірного відображення "картини реального світу", тобто, адекватного відтворення предметної області задачі. Так у реальному світі доводиться мати справу з фізичними об'єктами, такими, наприклад, як люди, автомобілі, тощо. Ці об'єкти не можна віднести ні до даних, ні до функцій, оскільки їх одночасно характеризує сукупність певних *властивостей* (даних) та певна *поведінка* (дії).

Властивостями (іноді їх називають характеристиками) для людей, може бути, наприклад, їх вік або місце роботи; для автомобіля – потужність двигуна, кількість дверей. *Поведінка* – це певна реакція фізичного об'єкта у відповідь на зовнішню дію. Так, якщо натиснути на педаль газу автомобіля, то це призведе до його руху – вперед або назад; якщо на педаль тормозу, то – до зупинки автомобіля. Рух або зупинка автомобіля і є прикладом його поведінки. Поведінка схожа з роботою функції: виклик функції призводить до виконання якої-небудь дії.

Таким чином, ні окремо взяті глобальні дані, ні відокремлені від них функції не здатні адекватно відображати фізичні об'єкти реального світу.

Поява більш потужних комп'ютерів дала життя більш об'ємним і складним програмам. Сучасне програмне забезпечення у своїй більшості фактично представляє собою великі програмні системи, суттєвою рисою яких є їх значна складність, обумовлена наступними основними чинниками:

1) *Складністю прикладної області*. Прикладні проблеми часто містять складні елементи, до яких ставиться багато різних, часто суперечливих вимог. Ситуація погіршується тим, що розробники ПЗ можуть мати недостатню кваліфікацію в прикладній області, а замовники часто не до кінця уявляють, що власне їм потрібно.
2) *Складністю керування процесом розробки*. Складність проблеми і велика кількість вимог до ПЗ вимагає колективної роботи розробників. У цій ситуації важливим завданням є координація робіт і підтримання цілісності основної ідеї.

3) *Складність опису поведінки великих дискретних систем*. Програма сучасного комп'ютера є системою зі скінченною кількістю дискретних станів, причому їх може бути дуже і дуже багато. Переходи системи з одного стану в інший не можна моделювати неперервними фунціями, тому іноді зміна стану однієї частини системи може привести до катастрофічних змін у інших частинах. Адже всеохоплююче тестування великої програмної системи провести просто неможливо.

Щоб розробити такий сучасний програмний продукт, не достатньо просто об'єднати в послідовність певні машинні інструкції, оператори мови високого рівня чи навіть набори процедур та модулів. Як показала практика, традиційні методи імперативного програмування не здатні впоратися ні з наростаючою складністю програм та процесу їх розробки, ні з необхідністю підвищення надійності ПЗ.

Тому виникла нагальна потреба в новій методології програмування, яка була б здатна вирішити весь цей комплекс проблем. Такою методологію і стало об'єктно-орієнтоване програмування (ООП).

Концептуальна модель ООП

Визначальнми ідеями об'єктно-орієнтованого програмування є:

1) логічне об'єднання *даних* та *дій* над ними в єдине ціле, яке називають *об'єктом*;
2) представлення програми як сукупності взаємозв'язаних *об'єктів*, які взаємодіють між собою таким чином, щоб забезпечити певну поведінку системи.

Як і реальний об'єкт навколишнього світу, *об'єкт* в ООП визначає певну сутність предметної області, яка характеризується своїм *станом* і *поведінкою*. Стан - це дані, поведінка - це те, як об'єкт діє і реагує. Причому стан (дані) може змінюватися тільки в процесі поведінки і не може бути зміненим безпосередньо із зовнішнього світу.

Поведінка об'єкта визначається допустимим набором операцій над його даними. Реалізацію операції над об'єктом часто називають ***методом.*** Методи подібні функціям, але вони являються частиною об'єкта. Об'єкти можуть "породжуватися" і "вмирати".

Об'єкт представляє собою інформаційну модель і природну імітацію діяльності деякого елемента реального світу. Як інформаційна модель, об'єкт зберігає в собі ту інформацію, яка необхідна для реалізації системи, що проектується. Наприклад, якщо система, що проектується, повинна підтримувати взаємодію банка і клієнта, основними об'єктами цієї системи будуть **`Банк`** і **`Клієнт`**. Ці об'єкти являють собою інформаційні моделі відповідно банку і особи (юридичної чи фізичної), що користується його послугами.

Щоб забезпечити певну поведінку системи, об'єкти мають взаємодіяти між собою. Взаємодія об'єктів полягає в обміні повідомленнями. ***Повідомлення*** являє собою *запит* об'єкта-відправника об'єкту-одержувачу на виконання деякої операції, доповнений набором аргументів, які можуть знадобитися при виконанні відповідної дії. Об'єкт-відправник в акті взаємодії називають *клієнтом*, а об'єкт-одержувач

повідомлення – *сервером*. Запит, що є сукупністю даних певного типу, може містити параметри – дані, необхідні серверу для виконання доручення.

Об'єкт, що приймає повідомлення, реагує на нього виконанням одного із своїх методів. При цьому він може послати повідомлення іншим об'єктам, отримати від них відповіді, змінити свій стан і, якщо це необхідно, повернути відповідь тому об'єкту, який послав йому повідомлення. Наприклад, об'єкт **`Покупець`**, може надіслати об'єкту **`Банк`** повідомлення з дорученням перевести на рахунок об'єкта **`Магазин`** суму **`N`**. Параметрами повідомлення будуть число **`N`** і юридична адреса об'єкта **`Магазин`**.

Виконання програми зводиться до взаємодії об'єктів, кожен з яких динамічно може створити нові об'єкти, керуючи ними в подальшому, або знищити ці об'єкти, включаючи самого себе.

Таким чином, об'єктно-орієнтована парадигма програмування використовує *об'єктну декомпозицію* задачі, при якій структура програми (системи) описується в термінах об'єктів і зв'язків між ними, а поведінка - в термінах обміну повідомленнями між цими об'єктами (рис. 4). Головною ідеєю ООП є логічне об’єднання в межах об'єкта даних та операцій над ними.

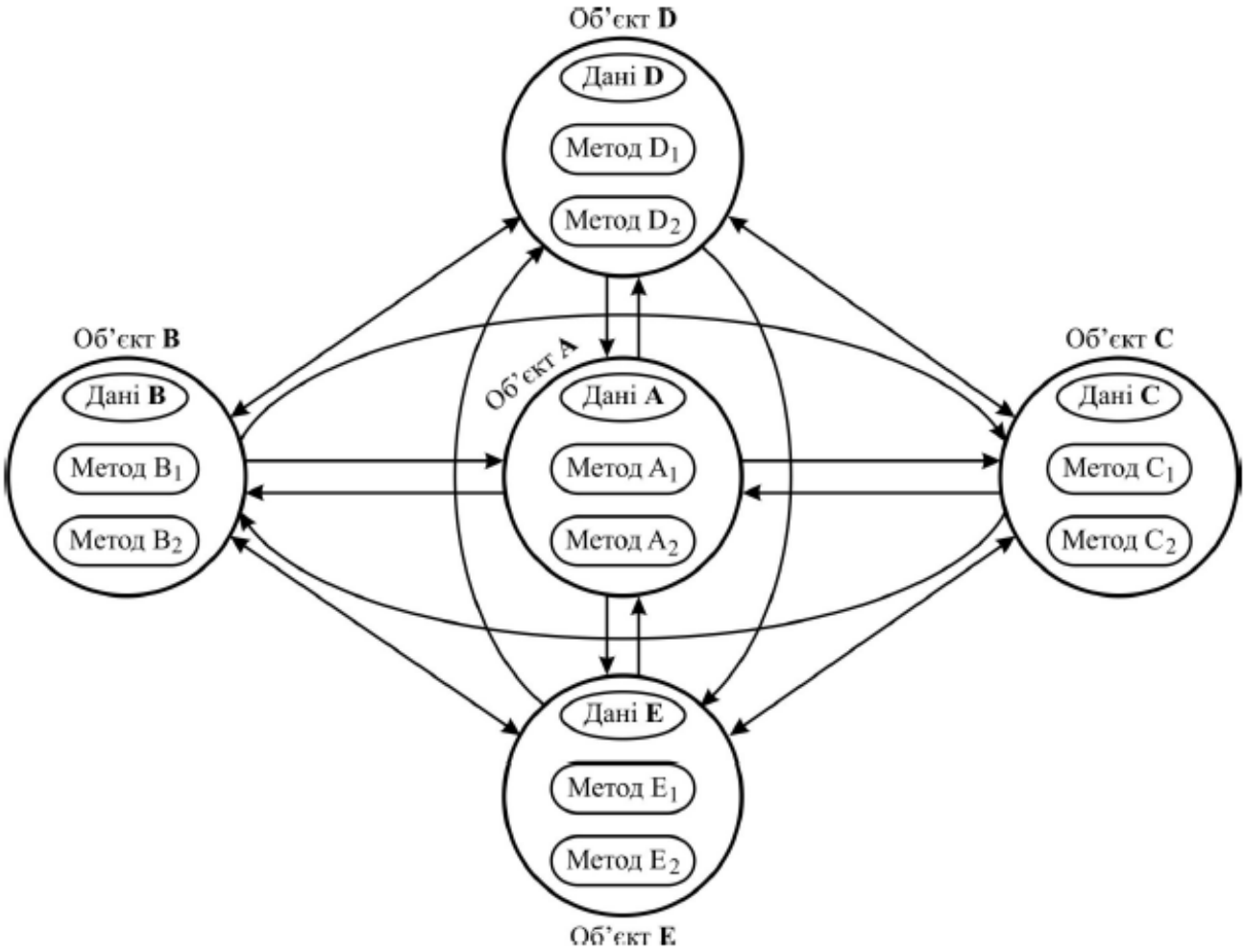

Рис. 4. Об'єктно-орієнтована декомпозиція

При переході від імперативного до об'єктно-орієнтованого програмування важливо зрозуміти ключову ідею: схема "дані -> процедура -> дані" замінюється на схему "запит -> об'єкт -> дані".

Для кращого розуміння призначення об'єктів, уявімо собі деяке підприємство, яке складається з бухгалтерії, відділу кадрів, відділу реалізації продукції, відділів головного технолога та головного механіка (рис. 5) і т.д. Поділ підприємства на відділи є важливою частиною структурної організації його виробничої діяльності. Для більшості підприємств в обов'язки окремого співробітника не входить вирішення одночасно кадрових, виробничих і обліково-бухгалтерських питань. Різні обов'язки чітко розподіляються між підрозділами, тобто у кожного підрозділу є дані, з якими він працює: у бухгалтерії – заробітна плата, у відділі реалізації продукції – інформація, яка стосуються торгівлі, у відділі кадрів – персональна інформація про співробітників, у відділі головного технолога – стан технологічного процесу виготовлення продукції, у відділі головного механіка – технічний стан обладнання та устаткування, транспортних засобів і т.д. Співробітники кожного відділу виконують операції тільки з тими даними, які їм належать.

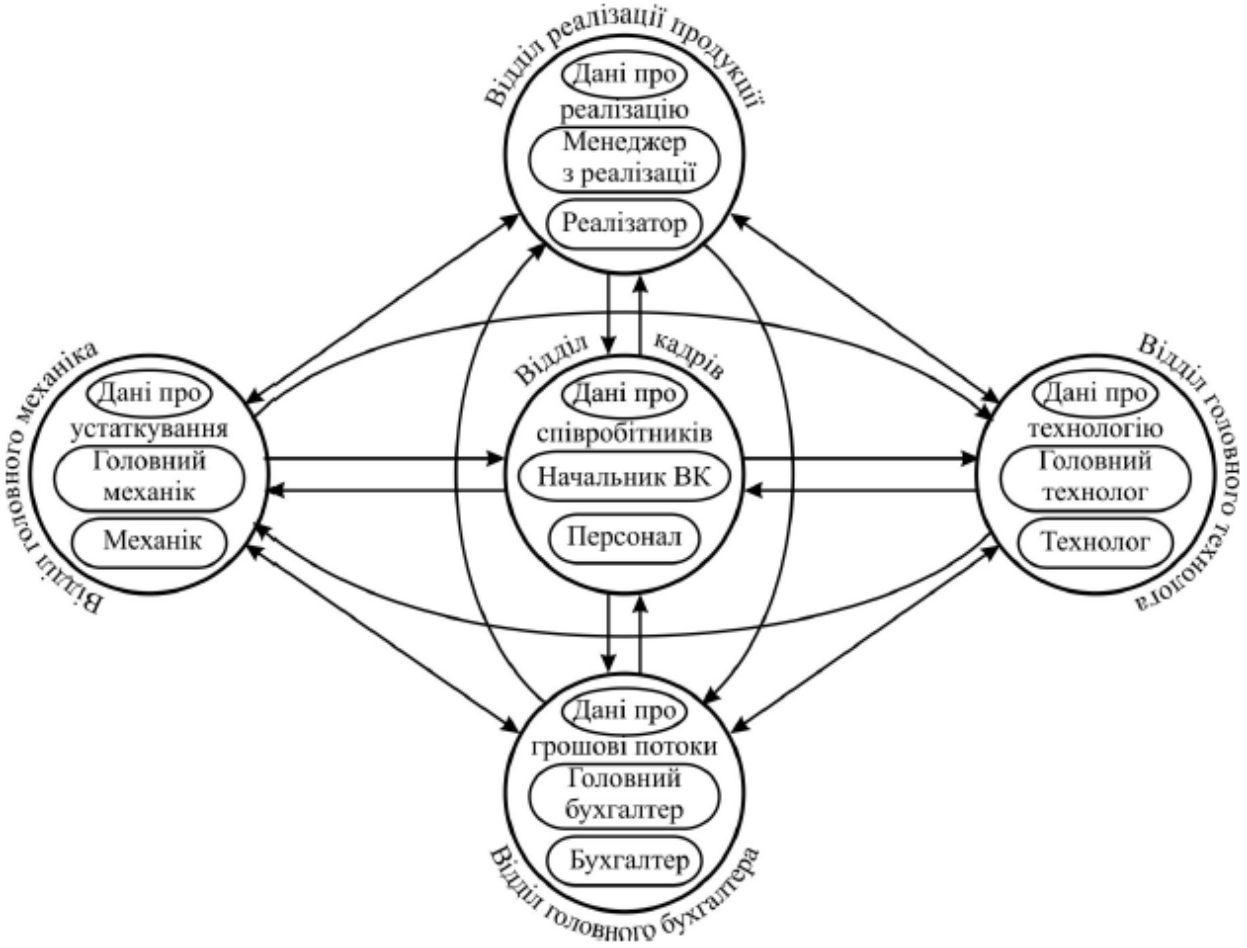

Рис. 5. Структура зв'язків між відділами і їх виробничими функціями

Структурний поділ обов'язків дає змогу легко стежити за виробничою діяльністю підприємства та контролювати її, а також підтримувати цілісність інформаційного простору підприємства. Наприклад, бухгалтерія несе відповідальність за інформацію, яка стосується нарахування заробітної плати, сплати податків, здійснення обліку матеріальних цінностей тощо. Якщо у менеджера з реалізації продукції виникне потреба дізнатися про загальний оклад співробітників фірми за деякий місяць, то йому не потрібно йти в бухгалтерію і ритися в картотеках; йому достатньо послати запит бухгалтеру з нарахування заробітної плати, який має знайти потрібну інформацію, обробити її та надати достовірну відповідь на запит.

Така схема організації роботи фірми забезпечує правильне оброблення даних, а також здійснює їх захист від можливої дії сторонніх осіб.

Аналогічні об'єкти коду програми створюють таку саму її організацію, яка має забезпечити цілісність її даних, доступ до них і виконання над ними відповідних дій.

Отже, об'єктно-орієнтована парадигма програмування заснована на об'єктному моделюванні задачі. Вона дозволяє описати завдання в контексті самого завдання, а не в контексті комп'ютера, на якому воно буде розв'язане. Роль програміста у цьому випадку полягає у формуванні і реалізації таких об'єктів, взаємодія яких при функціонуванні програми приведе до досягнення необхідного кінцевого результату.

При цьому методологія об'єктно-орієнтованого програмування успадкувала всі найкращі надбання структурного та модульного програмування, використавши їх для реалізації ряду принципово нових підходів до проектування програмного забезпечення.

Слід зауважити, що переваги ООП повною мірою виявляються лише при розробці досить складних програмних систем. Спроби використовувати ООП для програмування нескладних алгоритмів, пов'язаних, наприклад, з розрахунками за готовими формулами, найчастіше виглядають штучними нагромадженнями непотрібних мовних конструкцій. Такі програми зазвичай не потребують структуризації, поділу алгоритму на ряд відносно незалежних частин, їх простіше і природніше розробляти традиційним методом процедурного програмування.

Складові об'єктного підходу

Концептуальною основою об'єктно-орієнтованої парадигми програмування є ряд принципів:

1) абстрагування;
2) обмеження доступу;
3) модульність;
4) ієрархічність;
5) типізація;
6) паралелізм;
7) збережуваність.

Абстрагування. Це процес виділення *абстракцій* в предметній області задачі.

Абстракція - це сукупність характеристик деякого об'єкта, суттєвих з точки зору задачі (опис реального об'єкта без непотрібних для задачі подробиць).

Відповідно до визначення, абстракція реального предмета істотно залежить від розв'язуваної задачі: в одному випадку розробника буде цікавити форма предмета, в іншому вага, в третьому - матеріали, з яких він зроблений, в четвертому - закон руху предмета і т.д.

Отже, основна задача абстрагування - відділити основне від другорядного. Абстрагування концентрує увагу на зовнішніх особливостях об'єкта і дозволяє відокремити його найістотніші особливості поведінки від несуттєвих.

Обмеження доступу. Це приховування окремих елементів реалізації абстракції, які не зачіпають суттєвих характеристик її як цілого.

Принцип приховування інформації був започаткований ще у рамках структурного підходу. Він полягав у тому, щоб розділити програмний модуль на інтерфейсну і реалізаційну частини, надати в розпорядження користувача цього модуля інтерфейсний протокол і приховати від нього реалізацію.

У ООП цей принцип набув подальшого розвитку. Він передбачає відокремлення двох частин в описі абстракції:

a) *інтерфейсу* - сукупності доступних ззовні елементів реалізації абстракції (основні характеристики стану і поведінки);
b) *реалізації* - сукупності недоступних ззовні елементів реалізації абстракції (внутрішня організація абстракції і механізми реалізації її поведінки).

Обмеження доступу в ООП дозволяє розробнику:

1) виконувати конструювання системи поетапно, не відволікаючись на особливості реалізації використовуваних абстракцій;
2) легко модифікувати реалізацію окремих об'єктів, що в правильно організованій системі не зажадає зміни інших об'єктів.

Поєднання об'єднання всіх властивостей об'єкта (складових його стану і поведінки) в єдину абстракцію і приховування деталей реалізації цих властивостей отримало назву ***інкапсуляції***.

Модульність. Це принцип розробки програмної системи, що передбачає реалізацію її у вигляді окремих частин (модулів).

Даний принцип успадкований від модульного програмування. У більшості мов, що підтримують принцип модульності як самостійну концепцію, інтерфейс модуля відокремлений від його реалізації.

При виконанні декомпозиції системи на модулі бажано об'єднувати логічно пов'язані абстракції і мінімізувати взаємні зв’язки між модулями.

Відповідно до парадигми ООП об’єкти необхідно фізично розподілити так, щоб вони складали логічну структуру проекту. Таким чином програміст повинен знайти розумний компроміс між двома полярними тенденціями: прагненням приховати інформацію та необхідністю забезпечити видимість тих чи інакших абстракцій в декількох модулях. Тут необхідно враховувати, що зміна інтерфейсної частини модуля може привести до ланцюгової перекомпіляції всього проекту. Тому, як правило, частину реалізації, яка вже остаточно відпрацьована відносять до інтерфейсу, а ту яка ще може піддаватися інтенсивним змінам приховують.

Таким чином, принципи абстрагування, інкапсуляції та модульності є взаємодоповнюючими. Об’єкт логічно визначає межі визначення певної абстракції, а інкапсуляція та модульність роблять ці межі фізично непорушними.

При колективній розробці програмного забезпечення роботу розподіляють за модульним принципом. Правильний поділ системи на модулі мінімізує зв’язки між працівниками. Це особливо актуально коли програма розробляється різними групами чи компаніями.

Ієрархічність. Це принцип, який передбачає використання при розробці програмних систем так званих *ієрархій*. ***Ієрархія*** - це впорядкована за рівнями система абстракцій.

За рахунок утворення абстракцій, що мають ієрархічну структуру, досягається спрощення розуміння складних об'єктів.

У ООП використовуються два види ієрархії:

- "*загальне/конкретне*" ("**is-a**" ["є"]) - показує, що деяка абстракція є окремим випадком (конкретизацією) іншої абстракції. Наприклад, овал і багатокутник служать конкретизацією плоскої фігури, коло — конкретизацією овалу, чотирикутник — конкретизацією багатокутника, подальшими конкретизаціями чотирикутника можуть служити паралелограм, прямокутник, ромб, квадрат.
- "*ціле/частина*" ("**part of**" ["частина ..."])) - показує, що деяка абстракція є частиною іншої абстракції. Наприклад, грань є частиною багатогранника, ребро — частиною грані, вершина — частиною ребра.

При побудові ієрархії "is-a" реалізується механізм *успадкування* властивостей, який є одним з найважливіших механізмів ООП. ***Успадкування*** - це таке співвідношення між абстракціями, коли одна з них використовує структурну або функціональну частину іншої або кількох інших абстракцій (рис. 6а).

Іншими словами, успадкування породжує ієрархію "узагальнення-спеціалізація", в якій абстракція-нащадок є спеціалізованим окремим випадком узагальненої (батьківської) абстракції.

При побудові ієрархії "part of" реалізується механізм *агрегації*. ***Агрегація*** - це таке співвідношення між абстракціями, коли одна з них об'єднує в собі декілька інших (рис. 6б).

Рис.6 – Відношення спадкоємства (а), відношення агрегації (б)

Типізація. Це принцип, що передбачає класифікацію абстракцій за *типом* - характеристикою їх властивостей, що включає структуру та поведінку.

Типізація - це накладання обмежень на властивості об'єктів, які перешкоджають взаємозамінності абстракцій різних типів (або сильно звужують можливість такої заміни).

Використання принципу типізації забезпечує:

- раннє виявлення помилок, пов'язаних з неприпустимими операціями над програмними об'єктами (помилки виявляються на етапі компіляції програми при перевірці допустимості виконання даної операції над програмним об'єктом);

– спрощення документування;
– можливість генерації більш ефективного коду.

Тип може зв'язуватися з програмним об'єктом статично і динамічно. *Статичне* (або *раннє*) *зв'язування* означає, що тип об'єкту визначається на етапі компіляції. *Динамічне* (або *пізнє*) *зв'язування* означає, що тип об'єкту визначається тільки під час виконання програми. Реалізація пізнього зв'язування в мові програмування дозволяє створювати змінні - покажчики на об'єкти, що належать різним класам (*поліморфні об'єкти*), що істотно розширює можливості мови.

Концепції типізації та зв'язування є незалежними. В різних мовах програмування вони зустрічається в різних комбінаціях: типізація – сильна, зв'язування – статичне (Ada); типізація – сильна, зв'язування – динамічне (C++, Object Pascal); типи – відсутні, зв'язування – динамічне (Smalltalk).

Паралелізм. ***Паралелізм*** - властивість декількох абстракцій одночасно перебувати в активному стані, тобто виконувати деякі операції.

Існує цілий ряд задач, вирішення яких вимагає одночасного виконання деяких послідовностей дій - *процесів* (*потоків керування*). До таких задач, наприклад, відносяться задачі автоматичного управління декількома процесами.

Кожна програма має, як мінімум, один потік керування. Паралельна ж система має багато таких потоків. При використанні паралелізму ключовим питанням є синхронізація потоків та розподіл доступу до спільних ресурсів.

Реальний паралелізм досягається тільки при реалізації задач такого типу в багатопроцесорних системах, коли є можливість виконання кожного процесу окремим процесором. Системи з одним процесором імітують паралелізм за рахунок алгоритмів поділу часу процесора між задачами управління різними процесами.

Залежно від типу операційної системи (одно- або мультипрограмної) поділ часу може виконуватися або системою, що розробляється (як в MS DOS), або використовуваною ОС (як в системах Windows).

Збережуваність. ***Збережуваність*** (***стійкість***) – властивість абстракції існувати в часі незалежно від процесу, який породив даний програмний об'єкт, і / або в просторі, переміщаючись з адресного простору, в якому він був створений.

Розрізняють:

– *тимчасові* об'єкти, що зберігають проміжні результати деяких дій, наприклад, обчислень;
– *локальні* об'єкти, що існують усередині підпрограм, час життя яких обчислюється від виклику підпрограми до її завершення;
– *глобальні* об'єкти, існуючі поки програма завантажена в пам'ять;
– *збережувані об'єкти* - дані яких зберігаються в файлах зовнішньої пам'яті між сеансами роботи програми.

Всі зазначені вище принципи в тій чи іншій мірі реалізовані в різних версіях об'єктно-орієнтованих мов. Однак, перші чотири із них (абстрагування, інкапсуляція, модульність, ієрархічність) є основними, оскільки без наявності хоча б одного з них модель не можна назвати об'єктно-орієнтованою. Останні три

принципа є додатковими, оскільки вони корисні в об’єктно-орієнтованому стилі, але не є обов’язковими.

Основні поняття ООП

Об'єкти

Як уже зазначалося, ***об'єкт*** в ООП визначає певну сутність предметної області, що характеризується певним станом і поведінкою.

З точки зору сприйняття людини, об’єктом може бути:

- відчутний і (або) видимий предмет;
- дещо, що сприймається мисленням;
- дещо, на що спрямована думка або дія.

Таким чином об’єкт моделює частину навколишньої дійсності, що має чітко визначене функціональне призначення в даній предметній області.

Приклади об’єктів: Банк «Аваль», Іван Сидоров, група ІП-61, телефон Nokia тощо. Тобто об’єктом є будь-яка річ, яка має зміст для розглядуваної прикладної області.

Стан об'єкта визначається сукупністю усіх його *атрибутів*, а також їх поточними значеннями. ***Атрибутами*** називають дані (характеристики, риси, якості), що притаманні об’єкту. Наприклад, модель геометричної фігури «коло» – об'єкт Circle, може містити такі атрибути як координати центра кола, його радіус, колір тощо.

Серед атрибутів можуть бути такі, значення яких не впливають на поведінку об'єкта, і такі, значення яких керують поведінкою об'єкта. Так, наприклад, реакція банку на одержання платіжного доручення від клієнта А не залежить від фінансового стану іншого клієнта. Однак, банк повинен контролювати рахунок клієнта, що відправив платіжне доручення (цей рахунок може бути просто заблокований). Тому об'єкт Рахунок Клієнта, що є частиною об’єкта Банк, крім змінної-атрибута *Сума*, повинен містити логічну змінну-атрибут *Відкритий*, значення якої керує поведінкою Банку.

Методи об'єкта дозволяють визначати так звані *властивості* цього об'єкта. ***Властивостями*** називають ті дані про об'єкт, які можна отримати в результаті застосування методів. Поняття властивості об'єкта, таким чином, не співпадає з поняттям атрибута. Наприклад, площа кола може не входити до списку атрибутів об'єкта Circle, але її можна обчислити, застосувавши відповідний метод.

Атрибути об’єкту є, як правило, статичними, так як вони складають основу об’єкту, що не змінюється.

Поведінка об'єкта, як зазначалося раніше, визначається допустимим набором операцій (*методів*) над його даними. Часто метод об'єкта виконується за декілька кроків, на кожному з яких об'єкт приймає рішення про зміну свого стану, і, отже, зміну поведінки. З погляду теорії, об'єкт є автоматом з виходом. Таким чином, задачі опису поведінки взаємодіючих об'єктів розв’язуються методами теорії автоматів.

Об’єкти можуть бути активними та пасивними, володіти або ні власним потоком керування (паралелізм). Активний об'єкт може бути автономним, міняти свій стан без зовнішнього впливу.

Буч запропонував більш розширений опис об’єкта: *об’єкт має стани, поведінку та індивідуальність*.

Суть сказаного полягає в тому, що об’єкт може мати у своєму розпорядженні внутрішні дані (які і є станом об’єкта), методи (які визначають поведінку об’єкта), і кожен об’єкт можна унікальним чином відрізнити від будь-якого іншого об’єкта – говорячи більш конкретно, кожен об’єкт має унікальну адресу у пам’яті.

Таким чином, структурно об'єкт - це сукупність *атрибутів* і *методів* їхньої обробки. Тобто, разом з даними, що описують якийсь об'єкт, зберігається і програмний код, який описує поведінку цього об'єкта. При цьому об'єкт - це дещо неподільне, сутність, яка зберігає свої властивості, тільки залишаючись єдиним цілим. Наприклад, модель геометричної фігури – об'єкт Фігура, повинна містити як атрибути-дані, що характеризують певну фігуру, так і методи - операції обробки цих даних відповідно функціоналу системи, що розробляється. Якщо мова йде про систему, що призначена для розв’язання обчислювальних задач, то до цих операцій можна віднести обчислення площі, периметра і інших чисельних характеристик геометричної фігури; якщо ж система призначена для виконання геометричних перетворень, то операціями будуть паралельне перенесення, поворот і т.ін.

Класи

Об'єкти можуть мати ідентичну структуру і відрізнятися тільки значеннями атрибутів. Сукупність об'єктів, які мають загальний набір атрибутів (загальну структуру) і однакову поведінку, називають ***класом***. Тобто, клас визначає абстрактні характеристики деякої сутності та дії, які вона здатна виконувати. Наприклад, клас Автомобіль може мати атрибути (властивості), притаманні всім автомобілям, зокрема, максимальну швидкість, потужність двигуна, колір кузова і т.ін., а також специфічну поведінку – здатність їздити(), гальмувати(), стояти() тощо, яка реалізується за допомогою методів.

Будь-який об'єкт - це екземпляр певного класу. Наприклад, об'єкт Audi_6 є екземпляром класу Автомобіль зі своїм варіантом значень зазначених вище атрибутів (властивостей) і конкретною поведінкою (стоїть, їде тощо). Терміни «екземпляр класу» і «об'єкт» взаємозамінні.

Об'єкти, які не мають повністю однакових атрибутів або не характеризуються однаковою поведінкою, за визначенням, не можуть бути віднесені до одного класу.

Визначення класу, по суті, аналогічне абстракції типу даних. Використання типів даних, визначених програмістом - одна з характерних рис структурного стилю програмування. Тому визначення об'єкта як змінної деякого об'єктного типу є розвитком методології структурного програмування.

Клас може розглядатися як шаблон для створення будь-якої кількості екземплярів об'єктів одного і того ж класу. Структурно він включає локальні дані (атрибути) і локальні функції (методи), які ще називають даними-членами і функціями-членами (рис. 7):

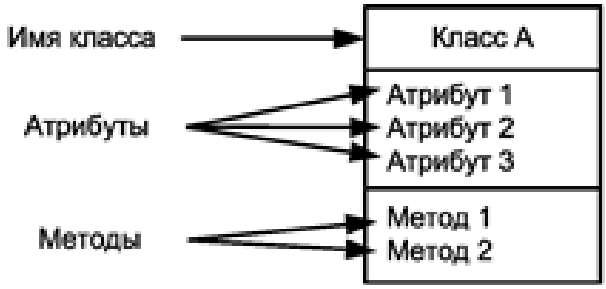

Рис. 7. Структура класу

Об'єкт можна розглядати як змінну певного типу (класу). Наприклад, тип int може розглядатися як клас цілих чисел, поведінка яких визначається допустимим набором операцій. Тоді об'єкт можна розглядати як екземпляр (змінну) цілого типу. Проте синтаксична конструкція визначення класу та її семантика істотно розширені по відношенню до визначення типу.

Важливо розуміти, що клас – це логічна абстракція, яка реально не існує доти, доки не буде створено об'єкт цього класу. Це аналогічно тому, що існування типу int ще не означає наявність змінних цього типу.

Класи є основним способом організації інформації в ООП. Будь-яку програмну систему, побудовану в об'єктному стилі, можна розглядати як сукупність класів.

Поняття класу та об’єкту дуже тісно зв’язані між собою, тому часто важко вести мову про об’єкт не опираючись на його клас. Але об’єкт визначає певну сутність, що визначена в часі та просторі, а клас визначає тільки абстракцію потенційних об’єктів.

Основні механізми ООП

Інкапсуляція

Як уже зазначалось, принципово новою і важливою ідеєю методології ООП є логічне об’єднання в єдине ціле даних (атрибутів) і методів їхньої обробки. Таке поєднання атрибутів і методів часто називають їх *інкапсуляцією* (включенням) в клас.

Дані прийнято інкапсулювати в класі таким чином, щоб доступ до них (для читання або для запису) здійснювався не безпосередньо, а за допомогою методів.

Відповідно до принципу обмеження доступу, повинен забезпечуватися різний ступінь доступності до даних і методів об'єкта: від загальнодоступних до таких, які доступні тільки з методів самого об'єкта. Деякі атрибути об'єкта можна приховати, якщо не визначити метод їх виведення. Також можна приховати деякі атрибути і методи, оголосивши їх приватними.

Користувач може взаємодіяти з об'єктом тільки за допомогою загальнодоступних методів, одержуючи від об'єкта атрибути і змінюючи їх, якщо це дозволено. Повний список доступних методів і атрибутів об'єкта називають його *інтерфейсом* або *протоколом* (рис. 8).

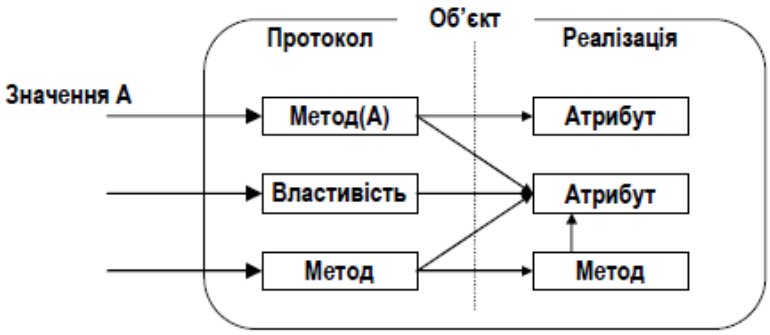

Рис. 8. Принцип інкапсуляції даних і операцій об'єкта

Наприклад, щоб переключити телевізійну програму достатньо на пульті дистанційного керування набрати її номер, що запустить складний механізм, який у результаті і призведе до бажаного результату. Користувачу не обов'язково знати, що відбувається в пульті дистанційного керування і телевізорі, - достатньо лише знати, що телевізор має таку можливістю (метод) і як її можна активувати.

Механізм об'єднання атрибутів і методів в межах класу та обмеження доступу до них, називають ***інкапсуляцією.*** Користувач може бачити і використовувати тільки інтерфейсну частину класу (список задекларованих атрибутів і методів класу), що захищає дані від зовнішнього втручання і некоректного використання.

Принцип інкапсуляції дозволяє мінімізувати число зв'язків між класами і, відповідно, спростити незалежну реалізацію і модифікацію класів.

Важливість інкапсуляції різко зростає із зростанням розмірів програми і збільшенням областей її застосування.

Успадкування

Ще однією новою ідеєю об’єктно-орієнтованої методології програмування є використання механізму *успадкування* (*наслідування*) як основного методу опису класів.

У повсякденному житті типовим є поділ класів на підкласи. Наприклад, клас Транспорт поділяється на класи Автомобілі, Вантажівки, Автобуси, Мотоцикли і т.д.

Принцип, закладений в основу такого поділу, полягає в тому, що кожен підклас володіє характеристиками, притаманними тому класу, з якого виділений даний підклас. Автомобілі, вантажівки, автобуси і мотоцикли мають колеса і двигун, які є характеристиками цих транспортних засобів. Окрім тих характеристик, які є загальними для класу і підкласу, підклас може мати і власні. Наприклад, автобуси мають велику кількість посадкових місць для пасажирів, тоді як вантажівки мають значний простір і потужність двигуна для перевезення вантажів.

Подібно до наведеного вище класу Транспорт, у програмуванні клас також може породити множину підкласів. Механізм, згідно з яким будь-який клас можна породити від іншого зі збереженням усіх атрибутів і методів класу-предка та додаючи, при необхідності, нові атрибути і методи, називають *успадкуванням* (або *наслідуванням*).

Клас, який породжує всю решту класів, називають *базовим*. Решта класів, які успадковують його властивості і водночас мають власні, називають *похідними*. Тобто, механізм успадкування дозволяє створювати нові (похідні) класи з розширеним представленням і модифікованими методами, не зачіпаючи при цьому код базових класів.

Так, якщо A і B – класи і B успадковує A, це означає, що всі атрибути класу A, за означенням є атрибутами класу B, і всі методи класу A, за означенням, є методами класу B (рис. 9). Клас A можна назвати класом-предком, B – класом-нащадком.

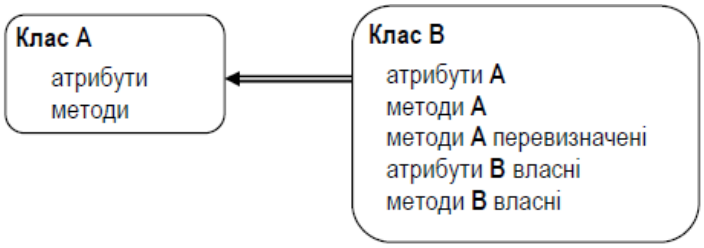

Рис. 9. Принцип успадкування класів

Метод породження нових об'єктів за допомогою механізму успадкування використовується для створення ієрархій класів (рис. 10).

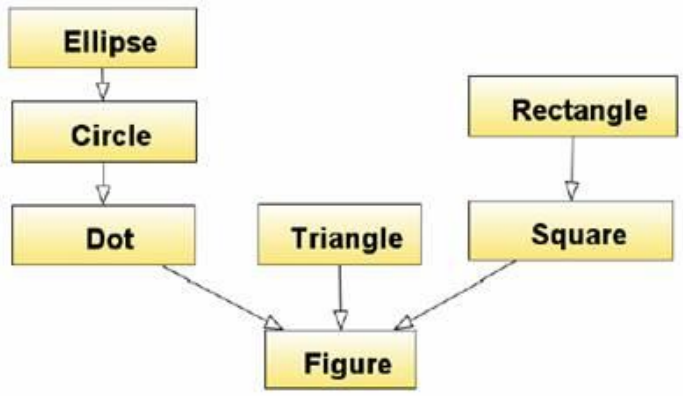

Рис 10. Приклад ієрархії класів

Як уже зазначалося, успадкування породжує ієрархію "узагальнення-спеціалізація", в якій клас-нащадок є спеціалізованим окремим випадком узагальненого (батьківського) класу. Наприклад, овал і багатокутник служать конкретизацією плоскої фігури, коло — конкретизацією овалу, чотирикутник — конкретизацією багатокутника, подальшими конкретизаціями чотирикутника можуть служити паралелограм, прямокутник, ромб, квадрат.

Верхні рівні ієрархічного дерева часто представляються так званими *абстрактними класами* - такими об'єктними типами, які не можна (і не має сенсу) використовувати для створення і використання екземплярів класів. Так для АСУ ВНЗ не потрібні об'єкти типу Людина, оскільки всі її об'єкти – або студенти, або викладачі, або інші категорії осіб, що працюють чи вчаться у даному навчальному закладі. Всі вони – об'єкти спеціалізованих класів, дочірніх для класу Людина. Таким чином, абстрактні класи використовуються тільки для визначення спеціалізованих класів.

Успадкування уможливлює повторне використання коду програми, тобто внесення одного разу розробленого класу в будь-які інші коди програм.

При роботі з об'єктами програміст зазвичай підбирає об'єкт, найбільш близький по своїх властивостях для вирішення конкретного завдання, і створює одного або декількох нащадків від нього, які «вміють» робити те, що не реалізовано в предку.

Послідовне проведення в життя принципу «успадковуй і змінюй» добре погоджується з поетапним підходом до розробки великих програмных проектів і багато в чому стимулює такий підхід.

Значення успадкування в ООП таке ж саме, як і функцій у структурному програмуванні, – скоротити розмір коду програми і спростити зв'язки між її елементами.

Слід зазначити, що підтримка механізму успадкування відрізняє об'єктно-орієнтовані мови від об'єктних. Зокрема, програмування, не засноване на ієрархічних відносинах, не відноситься до ООП, а називається *об'єктним* або *програмуванням на основі абстрактних типів даних*.

Поліморфізм

Механізм успадкування забезпечує спадкоємність атрибутів та методів і, як наслідок, подібність структури та поведінки предка і нащадка. Однак, разом із спадкоємністю, можна перевизначати частину поведінки батьківського класу: метод класу-нащадка, що має те ж ім'я, що і метод батьківського класу, може виконувати над об'єктом зовсім іншу за змістом операцію.

Той факт, що для різних об'єктів можуть бути визначені однойменні методи, означає, що інтерпретація цих методів (операції, що реалізуються методами), залежить від об'єкта, якому адресоване повідомлення. Можливість класу-нащадка змінювати реалізацію методів батьківського класу називають *поліморфністю* класу, а механізм, згідно з яким родинні об'єкти, тобто об'єкти, що мають одного загального предка, можуть виконувати однотипні дії по-різному, залежно від того, об'єкт якого класу використовується при виклику відповідного методу, - ***поліморфізмом*** об'єктів (рис. 11).

Рис. 11. Поліморфна поведінка об'єктів

Поліморфізм забезпечується тим, що в класі-нащадку змінюють реалізацію методу класу-предка з обов'язковим збереженням сигнатури методу. Це забезпечує збереження незмінним інтерфейсу класу-предка і дозволяє здійснити зв'язування імені методу в коді з різними класами - з об'єкта якого класу здійснюється виклик, з того класу і береться метод з даним ім'ям. У результаті в об'єкті-предку і об'єкті-

нащадку будуть діяти однойменні методи, що мають різну алгоритмічну основу і, відповідно, надають об'єктам різні властивості. Поліморфізм дозволяє приховувати різні реалізації методів за однаковим їх інтерфейсом.

Такий механізм зв'язування ще називають *динамічним* (або *пізнім*) *зв'язуванням* - на відміну від *статичного* (*раннього*) *зв'язування*, що здійснюється на етапі компіляції програми.

Розглянемо, наприклад, кілька класів: Квадрат, Коло, Трикутник, похідних від базового класу Геометрична фігура. Нехай базовий клас має метод Draw(), який змушує об’єкт малювати сам себе. Тоді всі похідні класи-фігури, повинні перевизначати цей метод, адже кожну фігуру потрібно малювати по-своєму.

Власне, поліморфізм, як і успадкування, теж є методом забезпечення повторного використання коду. Механізм успадкування дозволяє використовувати спільний код, реалізований у методах класу-предка, а механізм перевизначення методів, дає змогу модифікувати цей код, задовольняючи потреби програміста в реалізації індивідуальних особливостей поведінки об'єктів.

Таким чином, методологія ООП ґрунтується на трьох основних механізмах:

- *інкапсуляція* даних та функцій (методів) їх обробки в єдиній інформаційній структурі;
- *успадкування* класів – імпорт властивостей базових класів у похідні класи;
- *поліморфізм*, який полягає у використанні однакових інтерфейсів для роботи з різними за функціональністю родинними об’єктами.

Основи об’єктно-орієнтованого підходу до розроблення складних програм

Як уже зазначалося, переваги ООП повною мірою виявляються лише при розробці досить складних програмних систем, що розробляються групою розробників або навіть і декількома групами. Тому необхідно мати надійні способи побудови таких систем, які б з високою ймовірністю гарантували успіх в їх проектуванні.

Зазвичай складні програмні системи мають такі основні властивості:

- вони є ієрархічними і складаються із взаємопов’язаних частин, що також можуть поділятися на підсистеми;
- поділ систем на елементарні компоненти є відносно довільним і здебільшого залежить від проектувальника;
- зв’язок в середині компонентів набагато сильніший ніж між компонентами, це обумовлює “високочастотну” взаємодію всередині компонентів та “низькочастотну” між компонентами;
- ієрархічні системи складаються з невеликої кількості типів підсистем різним чином скомбінованих та організованих;
- діюча складна система є результатом розвитку більш простої системи, що працювала раніше (складна система, що спроектована з “нуля” навряд чи запрацює)

Під проектуванням зазвичай розуміється деякий уніфікований підхід, за допомогою якого здійснюється пошук шляхів вирішення певної проблеми. У контексті інженерного проектування мета проектування визначається як створення системи, яка

- задовольняє заданим функціональним специфікаціям;
- узгоджена з обмеженнями, що накладаються обладнанням;
- задовольняє явні і неявні вимоги з експлуатації та ресурсоспоживання;
- задовольняє явним і неявним критеріям дизайну продукту;
- відповідає вимогам до самого процесу розробки, таким, наприклад, як тривалість та вартість, а також використання додаткових інструментальних засобів.

Проектування передбачає врахування суперечливих вимог. Його продуктами є моделі, що дозволяють зрозуміти структуру майбутньої системи, збалансувати вимоги і намітити схему реалізації.

Досвід проектування складних програмних систем показує [Буч], що найбільш успішними є ті системи в яких закладено добре продумані структури класів та об'єктів і які володіють згаданими вище властивостями. Структури класів та об'єктів системи разом називають *архітектурою системи*.

Методи проектування мають деякі спільні властивості:

- *умовні позначення* – мову для описання кожної моделі;
- *процес* - правила проектування моделей;
- *інструменти* – засоби, які пришвидшують процес створення моделей, і в яких вже закладені закони функціонування моделей. Інструменти дозволяють виявити помилки в процесі розробки.

При проектуванні таких систем зосереджуються на трьох основних аспектах:

- декомпозиція;
- абстрагування;
- ієрархія.

Декомпозиція. Побудова та управління складною програмною системою базується на древньому принципі управління: "розділяй та володарюй". При проектування така система розкладається на все менші та менші підсистеми, що можуть розроблятися та вдосконалюватися незалежно.

На відміну від структурного проектування "зверху вниз", коли декомпозиція сприймається як розділення алгоритмів, де кожен модуль виконує якусь частину загального процесу, при об'єктно-орієнтованій декомпозиції система розбивається на об'єкти, кожен з яких живе своїм життям (знаходиться у якомусь стані). Обмінюючись повідомленнями об'єкти об'єднуються у систему для виконання спільних дій.

При проектуванні програмних систем важливі як алгоритмічна, так і об'єктна декомпозиції. Вони є взаємодоповнюючими, а не взаємозамінними. Але об'єктно-орієнтована декомпозиція значно зменшує об'єми програмних систем за рахунок

повторного використання загальних механізмів, що приводить до економії засобів вираження сутності системи.

Абстракція. Абстракція полягає у виділенні (на певному рівні) найбільш важливих ознак об’єктів (інформації про об’єкти) реального світу, які моделюються, та в абстрагуванні від незначних деталей. Таким чином формується узагальнююча, ідеалізована модель об’єкту. Незначні на певному рівні абстракції елементи моделі можуть уточнюватися на нижчих рівнях і на їх основі об’єкти розрізняються між собою.

Ієрархія. Ієрархія класів та об’єктів є способом розширення інформативності елементів системи. Об’єктна структура ілюструє схему взаємодії об’єктів між собою; структура класів визначає узагальнення структур та їх поведінку в середині системи.

Визначити ієрархічні зв’язки в системі не завжди просто, але після їх визначення структура складної системи та сама система стає зрозумілішою.

Уніфікованим підходом до створення великих програмних систем є так званий *об'єктно-орієнтований підхід*. Він включає в себе 3 основні компоненти:

- об'єктно-орієнтований аналіз (*object-oriented analysis*, OOA),
- об'єктно-орієнтоване проектування (*object-oriented design*, OOD),
- об'єктно-орієнтоване програмування (ООП).

Об'єктно-орієнтований аналіз. *Об'єктно-орієнтований аналіз* - це методологія, що виявляє концептуальні сутності (абстракції) предметної області для розуміння і пояснення того, як вони взаємодіють між собою.

Мета об'єктно-орієнтованого аналізу - максимально повний опис задачі. На цьому етапі виконується аналіз предметної області задачі, об'єктна декомпозиція системи, визначаються найважливіші особливості поведінки об'єктів (опис абстракцій).

За результатами аналізу розробляється структурна схема програмної системи, на якій показуються основні об'єкти і повідомлення, що передаються між ними, а також виконується опис абстракцій.

Таким чином, об'єктно-орієнтований аналіз спрямований на створення моделей реальної дійсності на основі об'єктно-орієнтованого світогляду.

Об’єктно-орієнтоване проектування. *Об’єктно-орієнтоване проектування* – це методологія проектування, що поєднує в собі процес об’єктної декомпозиції та прийоми представлення логічної і фізичної моделі (структури) системи, а також її статичні та динамічні аспекти.

Саме об'єктна декомпозиція відрізняє об'єктно-орієнтоване проектування від структурного: в першому випадку логічна структура системи відображається абстракціями у вигляді класів і об'єктів, у другому - алгоритмами.

Розрізняють:

1) *логічне проектування*;
2) *фізичне проектування*.

Логічне проектування полягає в розробці структури класів: визначаються атрибути для зберігання складових стану об'єктів і алгоритми методів, що реалізують аспекти поведінки об'єктів. При цьому використовуються розглянуті вище прийоми розробки класів: успадкування, композиція, поліморфізм і т.д. Результатом є ієрархія (діаграма) класів, що відображає взаємозв'язок класів, а також опис класів.

Фізичне проектування включає об'єднання описів класів в модулі, вибір схеми їх підключення (статична або динамічна компоновка), визначення способів взаємодії з обладнанням, з операційною системою і/або іншим програмним забезпеченням (наприклад, базами даних, мережевими програмами), забезпечення синхронізації процесів для систем паралельної обробки і т.ін.

Результатом об'єктно-орієнтованого проектування є побудова *діаграми класів* (*design class diagram*) і *діаграм взаємодій* (*collaboration diagram*).

Припустимо, потрібно розробити систему автоматизації банку. Один із прецендентів цієї системи може бути представлений наступним чином (рис. 12):

Рис. 12. Схема взаємодії об’єктів

Тобто, в операції зняття грошей через банкомат беруть участь 3 об'єкти: «клієнт Іванов», «банкомат на Тверській» і «рахунок № 66579801», який відкритий в даному банку для Іванова. Підійшовши до банкомату і засунувши свою картку, об'єкт «клієнт Іванов» посилає банкомату повідомлення «Почати роботу». Отримавши таке повідомлення, банкомат виводить на екран якусь інформацію і запитує код доступу, тобто об'єкт «банкомат на Тверській» посилає повідомлення об'єкту «клієнт Іванов» - «Повідомити ідентифікаційний код». Якщо ідентифікація пройшла успішно, «клієнт Іванов» просить видати йому 1000 гривень. Він посилає повідомлення про це банкомату, а той в свою чергу об'єкту «рахунок № 66579801». Прийнявши це повідомлення, об'єкт «рахунок № 66579801» перевіряє чи є у нього 1000 гривень і, якщо є, пересилає дозвіл на зняття грошей, одночасно зменшуючи свій баланс на відповідну суму. Банкомат передає гроші і на цьому процедура закінчується. Об'єкти виконують необхідні дії, передаючи один одному повідомлення.

Опис у вигляді об'єктів дозволяє визначити різні компоненти системи. Ті ж самі об'єкти - «рахунок № 66579801» і «клієнт Іванов» - будуть брати участь в інший операції, при якій клієнт приходить у відділення банку для зняття або зарахування грошей на свій рахунок.

Прийоми представлення логічної (класи і об'єкти) і фізичної (модулі і процеси) моделі системи передбачають використання певної системи позначень, наприклад, UML-діаграм.

UML (уніфікована мова моделювання) – це графічна мова, що містить різні діаграми, які допомагають фахівцям з системного аналізу створювати алгоритми, а програмістам – з'ясовувати принципи роботи коду програми (рис. 13). UML є потужним інструментом, який полегшує процес програмування.

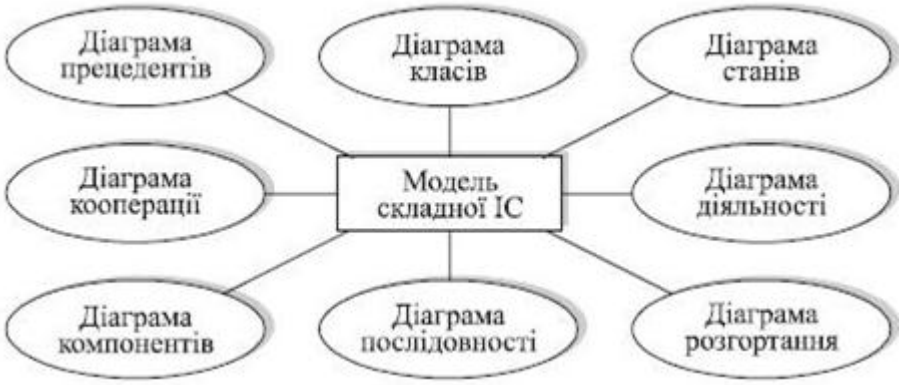

Рис. 13. Структура моделі складної інформаційної системи у нотації UML

Об'єктно-орієнтоване програмування. *Об'єктно-орієнтоване програмування* (*конструювання*) - це процес безпосереднього кодування (реалізації) проекту системи з використанням об'єктно-орієнтованої мови програмування.

Процес конструювання програмних систем поділяють на 2 етапи: *еволюція* системи та її *модифікація*.

Еволюція системи - це процес поетапної реалізації та підключення класів до проекту.

Процес починається зі створення проекту майбутнього програмного продукту. Потім реалізуються і підключаються класи, так щоб створити грубий, але, по можливості, працюючий прототип майбутньої системи. Він тестується та налагоджується. Наприклад, таким прототипом може служити система, що включає реалізацію основного інтерфейсу програмного продукту, коли передача повідомлень у відсутню частину системи не виконується. В результаті отримується працездатний прототип продукту, який може бути, наприклад, показаний замовнику для уточнення вимог.

Потім до системи підключається наступна група класів, наприклад, пов'язана з реалізацією деякого пункту меню. Отриманий варіант також тестується та налагоджується, і так далі, до реалізації всіх можливостей системи.

Використання поетапної реалізації істотно спрощує тестування і налагодження програмного продукту.

Модифікація - це процес додавання нових функціональних можливостей або зміна існуючих властивостей системи.

Як правило, зміни зачіпають реалізацію класу, залишаючи без зміни його інтерфейс, що при використанні ООП зазвичай обходиться без особливих неприємностей, тому що процес змін зачіпає локальну область. Зміна інтерфейсу - так само не дуже складна задача, але її вирішення може привести до необхідності узгодження процесів взаємодії об'єктів, що спричинить необхідність внесення змін в інших класах програми. Однак скорочення кількості параметрів в інтерфейсній частині в порівнянні з модульним програмуванням істотно полегшує і цей процес.

Простота модифікації дозволяє порівняно легко адаптувати програмні системи до постійно змінюваних умов експлуатації, що збільшує час життя систем, на розробку яких витрачаються величезні часові і матеріальні ресурси.

Програмування передбачає насамперед правильне та ефективне використання механізмів конкретних мов програмування; проектування ж, навпаки, основну увагу приділяє правильному і ефективному структуруванню складних систем.

Сучасні об'єктно-орієнтовані мови програмування

Історично першою мовою ООП є Simula, розроблена у 1967 р. Закладена у Simula об'єктна концепція була розвинута у мовах Smalltalk, Objective C, Eiffel, C++, CLOS, Ada, Oberon, Object Pascal, Visual Basic, Java, PHP, Perl, Python, C#, Ruby, Action Script 3 та ін.

Об'єктно-орієнтовані мови програмування поділяють на дві групи: чисті та гібридні. Чисті об'єктно-орієнтовані мови, до яких належать Smalltalk, Objective C, Eiffel, Java, C#, Ruby, Action Script 3, у повній мірі підтримують концепцію ООП. У цих мовах усе розглядається як елементи певних класів і у програмі не можна оперувати з глобальними даними або функціями, що не належать класам. Гібридні об'єктно-орієнтовані мови, такі як Object Pascal, C++, Perl, є більш універсальними, оскільки реалізують засоби процедурного та об'єктно-орієнтованого програмування.

Однією із найбільш відомих і популярних серед об'єктно-орієнтованих є мова C++, яка вже достатньо довгий час залишається визнаним лідером при розробці великих і складних програмних систем. Мова C++ є розвитком мови C. Вона має наскільки можливо повну сумісність з ANSI C. Відмінними особливостями C++ є:

- механізм класів, що дає можливість визначати і використовувати нові типи даних;
- повноцінний механізм структурної обробки виключень, відсутність якого в C значно ускладнювало написання надійних програм;
- механізм шаблонів - витончений механізм макрогенерації, глибоко вбудований в мову, що відкриває ще один шлях до повторного використання коду і багато іншого.

Довгий час мова C++ розвивалася поза рамками відповідних стандартів, тобто кожен розробник компіляторів по-своєму реалізовував окремі нюанси мови. Натепер існують наступні стандарти мови C++:

- ANSI C++ / ISO-C++ (1996),
- ISO/IEC 14882:1998 (1998),
- ISO/IEC 14882:2003 (2003),
- C++/CLI (2005),
- TR1 (2005),
- C++11 (2011).

Слід зазначити, що мова C++ - це універсальна мова програмування високого рівня з підтримкою декількох сучасних парадигм програмування: імперативної

(представлена мовою С), об'єктно-орієнтованої (представлена таким поняттям, як клас) та узагальненої (представлена шаблонами мови С++).

Java - це структурна об'єктно-орієнтована мова програмування, синтаксис якої нагадує синтаксис мови С++, однак ці мови мають мало спільного. Зокрема, мова Java позбавлена таких складових як покажчики, шаблони і множинне успадкування, що зробило її менш могутньою і гнучкою порівняно з мовою С++.

Своєю популярністю Java зобов'язана не стільки прогресу в мистецтві програмування, скільки змінам в комп'ютерному середовищі. З появою мережі Internet, в якій виявилися зв'язаними різні типи процесорів і операційних систем, актуальною стала проблема переносимості програм з одного середовища в інше. Її вирішенням і стала мова Java, єдиним найбільш важливим аспектом якої є можливість створювати на ній *кросплатформний* (сумісний з декількома операційними середовищами) переносимий програмний код.

Переносимість у Java досягається за допомогою перетворення вихідного коду програми в проміжний машинно-незалежний код, що іменується *байт-кодом*. Байт-код виконується спеціальною операційною системою – так званою віртуальною машиною Java (*Java Virtual Machine* - JVM). Java-програма може працювати в будь-якому середовищі, де доступна JVM. А оскільки JVM відносно проста для реалізації, вона швидко стала доступною для великої кількості середовищ.

Переносимість на нові платформи і адаптованість до нових оточень дозволяє виключити вплив програм, написаних мовою Java, на інші програми і файли і тим самим забезпечити безпеку при виконанні цих програм. Ці властивості мови Java дозволяють використовувати її як основну мову програмування для програм, орієнтованих на мережі комп'ютерів і, насамперед, на Internet.

Розробники Java успішно вирішили багато проблем, пов'язаних з переносимістю в середовищі Internet, але далеко не всі. Одна з них - міжмовна можливість взаємодії (*cross-language interoperability*) програмних і апаратних продуктів різних постачальників, або багатомовне програмування (*mixed-language programming*). В разі вирішення цієї проблеми програми, написані на різних мовах, могли б успішно працювати одна з іншою. Така взаємодія необхідна для створення великих систем з розподіленим програмним забезпеченням (ПЗ), а також для програмування компонентів ПЗ, оскільки найціннішим є компонент, який можна використовувати у широкому діапазоні комп'ютерних мов і операційних середовищ.

Окрім того, в Java не досягнута повна інтеграція з платформою Windows. Хоча Java-програми можуть виконуватися в середовищі Windows (за умови встановлення віртуальної машини Java), Java і Windows не є міцно зв'язаними середовищами. А оскільки Windows - це найбільш широко використовувана операційна система в світі, відсутність прямої підтримки Windows - серйозний недолік Java.

Аби задовольнити ці потреби, Microsoft розробила мову C#, яка стала частиною спільної .NET-стратегії Microsoft.

C# безпосередньо пов'язана із С, С++ і Java (рис. 14). Прамовою C# є мова С. Від С мова C# успадкувала синтаксис, багато ключових слів і оператори. Крім того, C# побудований на покращеній об'єктній моделі, визначеній в С++. C# і Java зв'язані між собою дещо складніше. Як згадувалося вище, Java також є нащадком С і С++. У

неї теж загальний з нею синтаксис і схожа об'єктна модель. Подібно Java, C# призначений для створення переносимого коду. Проте C# - не нащадок Java. Швидше C# і Java можна вважати двоюрідними братами, що мають загальних предків, але отримали від батьків різні набори "генів".

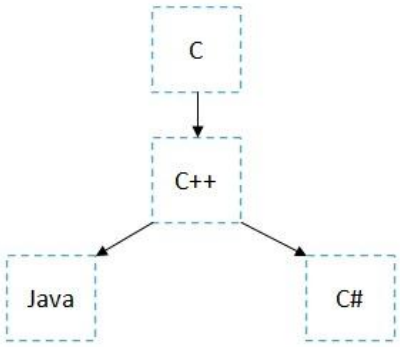

Рис. 14. Взаємозв'язок С-образних мов програмування

В теорії програмування прийнято розрізняти об'єктно-орієнтовані і об'єктні мови програмування. Останні відрізняються тим, що вони не підтримують успадкування властивостей в ієрархії абстракцій, наприклад, Ада - об'єктна мова, а C++ - об'єктно-орієнтована.

Узагальнимо фундаментальні характеристики об'єктно-орієнтованої мови (положення А. Кея):

- *Все є об'єктами.* Об'єкт можна сприймати як удосконалену змінну: він зберігає дані, але до нього можна «звертатися із запитами», вимагаючи виконати операції над собою. Теоретично абсолютно будь-який компонент розв'язуваної задачі (людина, будівля, послуга і т. ін.) може бути представлений у вигляді об'єкта.
- *Програма – це група об'єктів, які вказують один одному, що робити, за допомогою повідомлень.* Щоб звернутися із запитом до об'єкта, слід «послати йому повідомлення». Більш наочно повідомлення можна розглядати як виклик методу, що належить певному об'єкту.
- *Кожен об'єкт має власну «пам'ять», що складається з інших об'єктів.* Іншими словами, створення нового об'єкту здійснюється за допомогою вбудовування в нього вже існуючих об'єктів. Таким чином, можна сконструювати як завгодно складну програму, приховавши загальну складність за простотою окремих об'єктів.
- *У кожного об'єкта є тип.* Іншими словами, кожен об'єкт є екземпляром класу, де «клас» є аналогом слова «тип». Найважливіша відміна класів один від одного як раз і полягає у відповіді на питання: «Які повідомлення можна посилати об'єкту?»
- *Усі об'єкти певного типу можуть отримувати однакові повідомлення.* Це дуже важлива обставина, оскільки об'єкт типу «коло» також є об'єктом типу «фігура», і тому справедливим є твердження, що «коло» явно здатне приймати повідомлення для «фігури». А це означає, що можна писати код для фігур і бути впевненим у тому, що він підійде для всього, що потрапляє під поняття фігури. Взаємозамінність представляє одне з найбільш потужних понять ООП

Ці характеристики являють «чистий», академічний підхід до об’єктно-орієнтованого програмування.