9.1.2. Базові концепції об’єктно-орієнтованого програмування: абстракція, інкапсуляція, спадкування, поліморфізм

Абстракція в ООП

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

В програмуванні, **абстрагування** — це виділення лише важливих характеристик які потрібні для виконання завдання і відкидання інших.

Для різних цілей в одного і того ж об'єкта будуть важливі різні параметри.

Наприклад, яблуко. Для користувача, який має його з'їсти важливо сорт, соковитість, достиглість. Для користувача який має його кинути — вага і форма.

Якщо абстрагують дії то це абстрагування керування.

Якщо абстрагують структур даних — це абстрагування даних.

Наприклад, абстрагування керування в структурному програмуванні полягає у використанні підпрограм та визначених керівних конструкцій. Абстрагування даних дозволяє обробляти одиниці даних у змістовний спосіб.

Наприклад, абстрагування є основною мотивацією створення типів даних. Абстрагування є однією з парадигм Об'єктно-орієнтованого програмування.

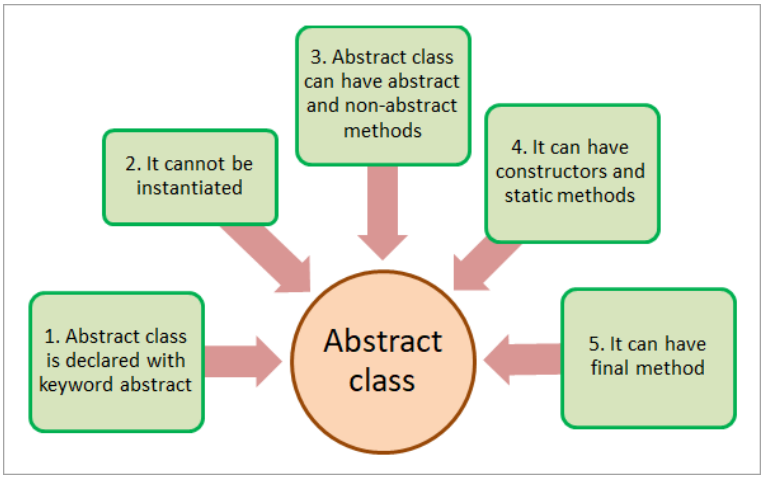
Абстракція в ООП – це виділення загальних характеристик об’єктів, їхніх властивостей і методів, при ігноруванні деталей реалізації. Цей процес дає змогу створювати простіші моделі складних систем, які містять тільки необхідні елементи для розв’язання задачі. Наприклад, ми можемо створювати моделі комп’ютерів, турбін або людського тіла, без згадки окремих деталей їхньої структури та функціонування.

Абстракція в ООП – приклад того, як можна спростити розуміння складної системи шляхом виокремлення її основних характеристик і визначення інтерфейсів для роботи з цими характеристиками. Вона дає змогу приховати деталі реалізації, що спрощує використання складної системи. Важливо зазначити, що вона не є атрибутом тільки ООП або програмування загалом, її застосовують у багатьох галузях знання, де необхідно створювати спрощені моделі складних систем для розв’язання конкретних завдань.

Абстракція є ключовою концепцією об’єктно-орієнтованого програмування (ООП), що дає змогу спростити складні системи, розділивши їх на більш дрібні частини та

приховуючи деталі реалізації. В ООП, абстракція визначається як процес створення нового класу на основі існуючого, який виступає як загальна концепція або шаблон для інших класів.

Наприклад, припустімо, у нас є клас “Користувач” з деталями реалізації, такими як ім’я, електронна пошта та пароль. Якщо ми створимо абстрактний клас “Акаунт”, який успадковує клас “Користувач”, ми зможемо використовувати його як загальний шаблон для створення інших класів, таких як “Адміністратор”, “Гість” тощо.



Рівні абстракції в ООП можуть бути різними, від найконкретніших до найабстрактніших. Наприклад, на найнижчому рівні, ми можемо мати конкретні класи з деталями реалізації, в той час як на найвищому рівні ми можемо мати абстрактні класи та інтерфейси, які представляють загальні концепції без деталей реалізації.

Прикладом абстракції в ООП може слугувати клас “Фігура”. Цей клас не має деталей реалізації, але визначає загальні властивості, такі як площа і периметр, які можуть бути успадковані іншими класами, такими як “Коло” або “Прямокутник”. Таким чином, ми можемо використовувати абстрактний клас “Фігура” як загальний шаблон для створення інших класів.

Таким чином, абстракція є важливою концепцією в ООП, яка дає змогу розділити складні системи на простіші компоненти, узагальнюючи їхні властивості та методи в абстрактні класи та інтерфейси. Це допомагає спростити розробку програмного забезпечення та підвищити його якість і надійність.

Різновиди абстракції в ООП. В ООП класи та об’єкти є основними формами абстракції. Клас визначає загальні властивості та методи, які можуть бути використані об’єктами цього класу, а об’єкт є конкретною реалізацією класу і містить властивості та методи, визначені в класі. Вона дає змогу створювати об’єкти, які можуть бути використані в різних контекстах, не знаючи всіх деталей їхньої реалізації.

Для створення абстракції в ООП використовуються інтерфейси та абстрактні класи. Інтерфейс визначає тільки сигнатури методів, але не містить реалізації. Абстрактний клас може містити як абстрактні методи, так і реалізації методів, але не може бути інстантований безпосередньо.

Рівні абстракції в програмуванні включають фізичний рівень, рівень реалізації, рівень інтерфейсів і рівень поведінки.

Прикладом використання абстракції в ООП може слугувати створення класу “Фігури”, який визначає загальні властивості та методи для всіх видів фігур (коло, квадрат, трикутник тощо). Цей клас може мати абстрактні методи для обчислення площі та

периметра, які мають бути визначені в класах-спадкоємцях для кожного конкретного виду фігур.

Класи та об’єкти як основні форми абстракції в ООП визначають структуру та поведінку об’єктів, а об’єкти представляють конкретні екземпляри класів. Класи визначають властивості (атрибути) і дії (методи), які можуть бути виконані об’єктами цього класу. Інтерфейси та абстрактні класи є ще одним способом створення абстракції в ООП.

Таким чином, використання абстракції в ООП дає змогу створювати гнучкі та розширювані програми, які можуть бути адаптовані до різних ситуацій та умов.

Використання абстракції в ООП має низку **переваг**, які дають змогу створювати більш ефективні та легко підтримувані програми.

Розглянемо деякі з них/

1. **Спрощення складних систем.** Абстракція дає змогу розбивати складні системи на простіші компоненти, що спрощує їхнє розуміння та управління. Кожен компонент може бути реалізований як окремий клас або об’єкт, що полегшує їх тестування, налагодження та підтримку.
2. **Поліпшення перевикористання коду.** Абстракція дає змогу створювати класи та об’єкти, які можуть бути перевикористані в різних частинах програми або навіть у різних програмах. Крім того, абстракція дає змогу створювати більш гнучкі та розширювані компоненти, які можна легко модифікувати й адаптувати під нові вимоги.
3. **Поліпшення безпеки та надійності.** Абстракція дає змогу приховати складність низькорівневих деталей від користувача, що покращує безпеку та надійність програми. Наприклад, використання абстракції вводу-виводу дає змогу створювати безпечні та надійні програми, які можуть обробляти дані без ризику виникнення помилок або витоків інформації.
4. **Збільшення продуктивності.** Абстракція дає змогу створювати більш ефективні програми, які можуть працювати з великими обсягами даних і виконувати складні операції швидше. Наприклад, створення оптимізованих структур даних, які можуть оброблятися швидше і займати менше пам’яті.

Таким чином, використання абстракції дає змогу створювати більш ефективні та легко підтримувані програми, що є ключовою перевагою ООП.

Далі розглянемо кілька **прикладів** використання абстракції в ООП.

1. Приклад використання абстракції на рівні класів. Припустимо, у нас є клас “Тварина”, який має кілька спадкоємців: “Собака”, “Кішка”, “Корова”, “Кролик” тощо. Кожен спадкоємець має свої унікальні властивості та методи, але всі вони успадковують загальні властивості та методи від класу “Тварина”. Клас “Тварина” – це абстрактний клас, оскільки ми не створюємо його екземпляри безпосередньо, а використовуємо тільки для створення спадкоємців. Він надає абстракцію для всіх його спадкоємців, об’єднуючи їх під загальною назвою “Тварини”.

2. Приклад використання абстракції на рівні інтерфейсів. Інтерфейс – це абстрактний тип даних, що визначає деякий набір методів, які мають бути реалізовані в класах, що реалізують цей інтерфейс. Інтерфейси використовуються для опису функціональності без визначення конкретної реалізації. Припустимо, у нас є інтерфейс “Фігура”, який визначає метод “площа”. Класи “Коло”, “Квадрат” і “Прямокутник”

можуть реалізувати цей інтерфейс і надавати свою власну реалізацію методу “площа”. Таким чином, ми можемо працювати з об’єктами різних класів, які реалізують інтерфейс “Фігура”, не знаючи конкретних деталей їхньої реалізації.

3. Приклад використання абстракції на рівні методів. Методи також можуть бути абстрактними. Абстрактний метод – це метод, який оголошений, але не має реалізації в абстрактному класі. Реалізація цього методу залишається для конкретних спадкоємців. Припустимо, у нас є абстрактний клас “Фігура”, який має абстрактний метод “периметр”. Класи “Коло”, “Квадрат” і “Прямокутник” успадковують клас “Фігура” і повинні реалізувати метод “периметр” відповідно до їхніх унікальних властивостей і форми.

Усі ці приклади показують, що абстракція допомагає створити більш зрозумілу модель, яка дає змогу уникнути зайвої складності та зменшити рівень деталізації.

Важливо зазначити, що рівні абстракції можуть варіюватися залежно від контексту і завдання. Наприклад, для розроблення гри може бути необхідно створити високорівневі абстракції для ігрових об’єктів, а для розроблення більш низькорівневих систем, таких як драйвери пристроїв, можуть знадобитися більш низькорівневі абстракції.

Хоча абстракція в ООП може бути дуже корисною, вона також може спричинити деякі **проблеми**.

- **Перевантаження абстракцій**: у деяких випадках може бути занадто багато рівнів абстракції, що може призвести до складності розуміння коду і ускладнити його підтримку.
- **Неправильна абстракція**: іноді розробник може вибрати неправильний рівень абстракції або неправильно визначити інтерфейс, що призведе до створення неефективного коду або коду, який неправильно працює.
- **Надлишкова абстракція**: іноді розробники можуть намагатися створити занадто абстрактний код, який не відображає дійсності. Це може призвести до створення занадто загального коду, який не є корисним для конкретних завдань.
- **Ускладнення коду**: іноді абстракція може ускладнити код і зробити його більш важким для розуміння та супроводу. Це може статися, якщо класи та інтерфейси створюють без ясної причини або без ясного розуміння того, як їх використовуватимуть.
- **Недостатня абстракція**: у деяких випадках розробники можуть не використовувати достатньо абстракції, що може призвести до створення дубльованого коду або до відсутності перевикористання коду.

Загалом, абстракція в ООП є дуже потужним інструментом, однак, важливо розуміти, що вона не є універсальним рішенням для всіх завдань і може призвести до небажаних наслідків, якщо її використання не буде обмеженим і раціонально керованим. Тому, під час використання абстракції, необхідно ретельно планувати та опрацьовувати інтерфейси та рівні абстракції, щоб уникнути можливих проблем, пов’язаних із перевантаженням, неправильною або надлишковою абстракцією, ускладненням коду або недостатньою абстракцією.

Приховування інформації (інкапсуляція).

Приховування деталей про роботу класів від об'єктів, що їх використовують або надсилають їм повідомлення. Так, наприклад, клас Собака має метод гавкати(). Реалізація цього методу описує як саме повинно відбуватись гавкання (приміром, спочатку

вдихнути(), а потім видихнути() на обраній частоті та гучності). Петро, хазяїн пса Сірка, не повинен знати як він гавкає. Інкапсуляція досягається шляхом вказування, які класи можуть звертатися до членів об'єкта. Як наслідок, кожен об'єкт надає кожному іншому класу певний інтерфейс — члени, доступні іншим класам. Інкапсуляція потрібна для того, аби запобігти використанню користувачами інтерфейсу тих частин реалізації, які, швидше за все, будуть змінюватись. Це дасть змогу полегшити внесення змін без потреби змінювати й користувачів інтерфейсу. Наприклад, інтерфейс може гарантувати, що щенята можуть додаватись лише до об'єктів класу Собака кодом самого класу. Часто, члени класу позначаються як публічні (англ. public), захищені (англ. protected) та приватні (англ. private), визначаючи, чи доступні вони всім класам, підкласам, або лише до класу в якому їх визначено. Деякі мови програмування йдуть ще далі: Java використовує ключове слово private для обмеження доступу, що буде дозволений лише з методів того самого класу, protected — лише з методів того самого класу і його нащадків та з класів із того ж самого пакету, C# та VB.NET відкривають деякі члени лише для класів із тієї ж збірки шляхом використання ключового слова internal (C#) або Friend (VB.NET), а Eiffel дозволяє вказувати які класи мають доступ до будь-яких членів.

Інкапсуляція — один з трьох основних механізмів об'єктно-орієнтованого програмування. Йдеться про те, що об’єкт вміщує не тільки дані, але і правила їх обробки, оформлені в вигляді виконуваних фрагментів (методів). А також про те, що доступ до стану об'єкта напряму заборонено, і ззовні з ним можна взаємодіяти виключно через заданий інтерфейс (відкриті поля та методи), що дозволяє знизити зв'язність. Таким чином контролюються звернення до полів класів та їхня правильна ініціалізація, усуваються можливі помилки пов'язані з неправильним викликом методу. Оскільки користувачі працюють лише через відкриті елементи класів, то розробники класу можуть як завгодно змінювати всі закриті елементи і, навіть, перейменовувати та видаляти їх, не турбуючись, що десь хтось їх використовує у своїх програмах.

Приклад на C++:

```
class Point {

  //можливий доступ лише з методів даного класу
   int x, y;
   bool visibility;
```

```
public:
  void createPoint(int a, int b) {
    x = a; y = b;
    visibility = true;
  }

  void setVisibility(bool visibility) {
    this->visibility = visibility;
  }

  int getX() {
    return x;
  }

  int getY() {
    return y;
  }
};
```

У цьому прикладі клас Point інкапсулює (приховує) координати точки. Доступ до них можливий лише за певними правилами, які реалізовуються через відповідні методи. Такими правилами можуть бути, наприклад, операція створення точки (установка значень координат), а також операції «увімкнення» і «вимкнення» видимості точки, отримання координат. Як видно з прикладу, для створення точки необхідно вказати координати точки і «увімкнути» дану точку (зробити її видимою). Якщо б клас був повністю відкритий, то можна б було вручну встановити відповідні поля класу x, y та встановити visible в true. Проте програміст може легко забути встановити якусь з координат або забути встановити видимість. Метод createPoint забезпечує виконання усіх необхідних дій, а закриття доступу до координат змушує діяти лише через використання даного методу. Інкапсульованими також можуть бути і методи класу.

Інші особливості. В ООП рекомендується з самого початку створювати закриті поля і лише в разі крайньої необхідності надавати ширший доступ до них. Для роботи із закритими полями класів краще використовувати відповідний метод доступу (getter) та метод-мутатор (setter).

При збільшенні розмірів програм і комп'ютерних систем із використанням принципів ООП стає більш ясним їх огляд (у тому числі, тестування ПЗ і налагодження), написання документації щодо їх використання.

Термін **«інкапсуляція»**, він же перший принцип ООП, має два трактування. Найчастіше фахівці використовують цей термін тільки в одному значенні, забуваючи про інше, а це невірно.

Перша трактування – в один об’єкт або клас об’єднуються і дані, і методи, які працюють з цими даними. Друге трактування – інкапсуляція це приховування внутрішньої структури об’єкта від зовнішніх впливів. Всі зміни стану об’єкта відбуваються тільки за допомогою звернень до методів самого об’єкту.

Особисто я розумію під терміном «інкапсуляція» обидві ці трактування – і з’єднання методів з даними, і приховування даних від зовнішніх впливів, зміна їх тільки через звернення до методів самого об’єкту. Справа в тому, що друга трактування без першої не працює зовсім. Що ви будете приховувати, якщо у вас в об’єкті не буде методів і даних?

До чого відноситься принцип інкапсуляції? З точки зору ООП, об'єктом є не тільки клас або об'єкт, або як це називається в вашій мові програмування, а й інші структури, більш високого рівня, а саме: packages, namespaces, модулі, сабмодулі, підсистеми і вся система цілком. Якщо розглядати инкапсуляцию як приховування, ми розуміємо, що наш об'єкт не повинен змінюватися нічим зовні, крім його ж методів. Це допоможе уникнути випадкових взаємодій типу «гайку відкрутили – жопа відвалилася».

Якщо до стану об'єкта має доступ тільки сам об'єкт – я нагадую, що стан об'єкта це значення всіх його полів – то це нам дозволяє уникнути зайвих взаємозв'язків, несподіваних сайд ефектів, коли об'єкт працював-працював, і тут у нього раптово змінилося поле, і в ньому вже зовсім інше значення. Програма вилітає, і виходять інші проблеми типу raise condition.

Для програміста важливо пам'ятати, що завжди має сенс приховувати всі дані, закривати всі стану об'єкта від зовнішнього об'єкта, і пам'ятати про те, що приховування внутрішньої структури вашого модуля від зовнішнього втручання, теж має сенс. Звернення через інтерфейс або через його зовнішній паблік клас – це робота з модулем через його фасад, через сервіс, а не грубі роботи з чорного ходу, щоб подивитися, що там всередині.

Робота між модулями повинна йти тільки через прийняті інтерфейси, а не безпосередньо викликом якихось методів, які наступна команда може поміняти, і вони будуть працювати не так. Навіть якщо інтерфейс залишиться тим же, метод може діяти інакше. Через порушення інкапсуляції ми отримуємо велику кількість порушень, дисфункцію системи, складну підтримку – коли не можна, наприклад, швидко виправити баг або внести новий функціонал.

З принципу інкапсуляції безпосередньо виникає безліч паттернів GRASP. Наприклад, патерн GRASP "Information expert" – це пряма імплементація паттерна інкапсуляції. Де повинні оброблятися дані? В об'єкті, який їх містить. Це приватна, більш специфічна формулювання тієї ж самої інкапсуляції.

Те ж саме стосується low coping. Менше зв'язків між об'єктами означає, що до об'єктів ми звертається тільки через потрібні методи, а не смикаємо все підряд, не використовуємо reflection, щоб длубатися в кишках об'єкта. Тільки через паблік інтерфейс, тільки через методи, спочатку призначені для того, щоб до них звертатися.

Знання декількох принципів звільняє від знання багатьох фактів. Програмісти, які пам'ятають, що таке інкапсуляція в широкому сенсі – не лазити, куди не слід, чи не проскакувати між Лейер, не лізти в базу поверх UI – менше стикається з сайд ефектами. Будь-яке порушення структури програми загрожує ефектами, і всі їх можна звести до порушення інкапсуляції.

Наслідування, успадкування

Клас може мати «підкласи», спеціалізовані, розширені версії надкласу. Можуть навіть утворюватись цілі дерева успадкування. Наприклад, клас Собака може мати підкласи Коллі, Пекінес, Вівчарка тощо. Так, Сірко може бути екземпляром класу Вівчарка. Підкласи **успадковують** атрибути та поведінку своїх батьківських класів, і можуть вводити свої власні. Успадкування може бути одиничне (один безпосередній батьківський клас) та множинне (кілька батьківських класів). Це залежить від вибору програміста, який реалізовує клас та мови програмування. Так, наприклад, в Java дозволене лише одинарне успадкування, а в C++ і те й інше.

Наслідування, успадкування (англ. inheritance) — один з принципів об'єктно-орієнтовного програмування, який дає класу можливість використовувати програмний код іншого (базового) класу, доповнюючи його своїми власними деталями реалізації. Іншими словами, під час наслідування відбувається отримання нового (похідного) класу, який містить програмний код базового класу з зазначенням власних особливостей використання. Наслідування належить до типу is-a відношень між класами. При успадкуванні створюється спеціалізована версія вже існуючого класу.

Правильне використання механізму наслідування дає наступні взаємозв'язані **переваги**:

1. ефективна побудова важких ієрархій класів з можливістю їх модифікації. Роботу класів в ієрархії можна змінювати шляхом додавання нових успадкованих класів в потрібному місці ієрархії;
2. повторне використання раніше написаного коду з подальшою його модифікацією під поставлену задачу. Своєю чергою, новостворений код також може використовуватися на ієрархіях нижчих класів;
3. зручність в супроводі (доповнені) програмного коду шляхом введення нових класів з новими можливостями;
4. зменшення кількості логічних помилок при розробці складних програмних систем. Повторно використовуваний код частіше тестується, а, отже, менша ймовірність наявності в ньому помилок;
5. легкість в узгодженні різних частин програмного коду шляхом використання інтерфейсів. Якщо два класи успадковані від загального нащадка, поведінка цих класів буде однакова у всіх випадках. Це твердження виходить з вимоги, що схожі об'єкти повинні мати схожу поведінку. Саме використання інтерфейсів зумовлює схожість поведінки об'єктів;
6. створення бібліотек коду, які можна використовувати й доповнювати власними розробками;
7. можливість реалізовувати відомі шаблони проєктування для побудови гнучкого коду, який не змінює попередніх розробок;
8. використання переваг поліморфізму неможливо без успадкування. Завдяки поліморфізму забезпечується принцип: один інтерфейс — декілька реалізацій;
9. забезпечення дослідницького програмування (швидкого макетування). Таке програмування використовується у випадках, коли цілі та потреби до програмної системи на початку нечіткі. Спочатку створюється макет структури, потім цей макет поетапно вдосконалюється шляхом наслідування попереднього. Процес триває до отримання потрібного результату;
10. ліпше розуміння структури програмної системи програмістом завдяки природному представленню механізму успадкування. Якщо при побудові складних ієрархій намагатись використовувати інші принципи, то це може значно ускладнити розуміння усієї задачі та призведе до збільшення кількості помилок.

При використанні наслідування в програмах були помічені наступні **недоліки**:

1. неможливо змінити успадковану реалізацію під час виконання;

2. низька швидкість виконання. Швидкість виконання програмного коду загального призначення нижча ніж у випадку використання спеціалізованого коду, який написаний конкретно для цієї задачі. Однак, цей недолік можна виправити завдяки оптимізації коду;
3. велика розмірність програм завдяки використанню бібліотек загального призначення. Якщо для деякої задачі розробляти вузькоспеціалізований програмний код, то цей код буде займати менше пам'яти ніж код загального призначення;
4. збільшення складності програми у випадку неправильного або невмілого використання успадкування. Програміст зобов'язаний вміти коректно використовувати наслідування при побудові ієрархій класів. В іншому випадку це призведе до великого ускладненню програмного коду, і, як результат, збільшенню кількості помилок;
5. складність засвоєння початковими програмістами основ побудови програм, які використовують успадкування. Однак, цей недолік умовний, так як залежить від досвіду програміста.

Наслідування є механізмом повторного використання коду (англ. code reuse) і сприяє незалежному розширенню програмного забезпечення через відкриті класи (англ. public classes) та інтерфейси (англ. interfaces). Встановлення відношення наслідування між класами породжує ієрархію класів[en].

Типи наслідування.

«Просте» наслідування. «Просте» наслідування, або одинарне успадкування, описує спорідненість між двома класами: один з яких успадковується від іншого. З одного класу може виводитися багато класів, але навіть в цьому випадку подібний вид взаємозвязку залишається «простим» успадкуванням.

Абстрактні класи і створення об'єктів. Для деяких мов програмування справедлива наступна концепція. Існують «абстрактні» класи (оголошуються такими довільно або через приписаних їм абстрактних методів); їх можна описувати наявними поля та методи. Створення ж об'єктів (екземплярів) означає конкретизацію, застосовну тільки до неабстрактних класів (в тому числі, до неабстрактних наслідникам абстрактних), — представниками яких, в результаті, будуть створені об'єкти.

Множинне наслідування.

При множинному успадкуванні, у класа може бути більше одного предка. В цьому випадку клас успадковує методи всіх предків. Переваги такого підходу в більшій гнучкості.

Множинне наслідування реалізовано в C++. З інших мов, що надають цю можливість, можна відмітити Python і Eiffel. Множинне наслідування підтримується в мові UML.

Множинне успадкування — потенційне джерело помилок, які можуть виникати через наявність однакових імен методів у предків. В мовах, які позиціонуются як спадкоємці C++ (Java, C# та інші), було прийнято рішення відмовитись від множинного наслідування на користь інтерфейсів. Практично завжди можна обійтися без використання даного механізму. Однак, якщо така необхідність все-таки виникла, то для вирішення конфліктів використання успадкованих методів з однаковими іменами можливо,

наприклад, застосувати операцію розширення видимості — «::» — для виклику конкретного метода конкретного предка.

Спроба вирішення проблеми наявності однакових імен методів в предках була здійснена у мові Eiffel, в якій при описі нового класу необхідно явно вказати імпортовані члени кожного з успадкованих класів і їх імена у дочірньому класі.

Більшість сучасних об'єктно-орієнтованих мов програмування (C#, Java, Delphi та інші) підтримують можливість одночасно успадковуватись від класа-предка і реалізовувати методи декількох інтерфейсів одним і тим же класом. Цей механізм дозволяє в багато чому замінити множинне успадкування — методи інтерфейсів необхідно перевизначати явно, що виключає помилки при успадкуванні функціональності однакових методів різних класів-предків.

Множинна спадковість — властивість деяких об’єктно-орієнтованих мов програмування, в яких класи можуть успадкувати поведінку і властивості більш ніж від одного суперкласу (безпосереднього батьківського класу). Це відрізняється від простого спадкування, у випадку якого клас може мати тільки один суперклас.

Мови програмування з підтримкою множинного спадкування: Eiffel, C++, Dylan, Python, Perl, Curl, Common Lisp (завдяки CLOS), OCaml, Tcl (завдяки Incremental Tcl) та Object REXX (завдяки використанню класів домішок).

Множинне спадкування дозволяє класу успадковувати функціональність від декількох інших класів, оскільки дозволяє класу StreetMusician успадковуватись від класів Human, Musician, Worker. Це можна скоротити як StreetMusician : Human, Musician, Worker. При множинному спадкуванні в попередньому прикладі може виникнути невизначеність, якщо, наприклад, клас Musician походить від Human і Worker, а клас Worker також походить від Human. В такому випадку кажуть про присутність ромбоподібного спадкування. Таким чином отримуємо:

```
Worker        :  Human
Musician      :  Human, Worker
StreetMusician  :  Human, Musician, Worker
```

Якщо компілятор переглядає клас StreetMusician, йому необхідно знати, коли об'єднувати однакові властивості, а коли тримати їх окремо. Наприклад, має сенс об'єднати властивості Age класу Human в StreetMusician. Вік людини не змінюється незалежно від того, чи ми розглядаємо її як музиканта, працівника або як людину загалом. З іншого боку, ім'я може бути як сценічним псевдонімом, так і справжнім ім'ям. Вибір об'єднати або відокремити покладається на програміста, який має знати, що саме є правильним при розробці певного класу.

Різні мови обробляють повторюване успадкування різними шляхами.

- Eiffel дозволяє програмісту явно об'єднати або розділити властивості успадковані від суперкласів. Eiffel автоматично об'єднує властивості, якщо вони мають однакові імена та реалізації. Програміст має змогу перейменувати успадковані властивості, щоб розділити їх. Eiffel також дозволяє явне повторюване спадкування, таке як A: B, B.
- C++ вимагає явної вказівки, з якого батьківського класу треба використати дану властивість, тобто "Worker::Human.Age". C++ на відміну від Eiffel не дозволяє явного повторюваного спадкування через відсутність можливості вказати, який з суперкласів треба використовувати. C++ підтримує можливість уникнення

неоднозначності через створення єдиного екземпляра батьківського класу через використання механізму віртуальної спадковості (тобто "Worker::Human" і "Musician::Human" будуть вказувати на один і той самий об'єкт).

– Perl використовує список класів для спадкування як впорядкований список. Компілятор використовує метод знайденим першим за допомогою пошуку в глибину серед списку суперкласів або C3 лінеаризації ієрархії класів. Різні розширення забезпечують альтернативні побудови. Python має таку саму структуру, але, на відміну від Perl, містить це як частину синтаксису самої мови. В Perl і Python на семантику класу впливає порядок спадкування.

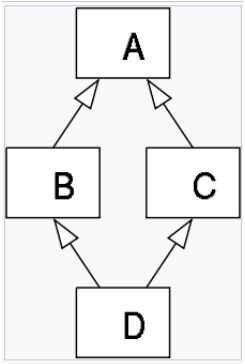
Smalltalk, C#, Objective-C, Object Pascal, Java, Nemerle, та PHP не підтримують множинної спадковості реалізації, і це дозволяє уникнути будь-якої неоднозначності. Однак всі вони, крім Smalltalk, надають класам можливість реалізувати декілька інтерфейсів.

Перевіркою на необхідність множинного спадкування може стати ситуація, коли після побудови структури класів кінцеві класи (листя) можна згрупувати в набори за різними ортогональними ознаками і ці набори між собою перетинаються, що може слугувати ознакою неможливості обійтися однією структурою спадкування, в якій існували б якісь проміжні класи з потрібною поведінкою.

«Алмазна проблема» (англ. diamond problem) (інколи згадувана як «Смертоносний діамант смерті») це неоднозначність, яка виникає, коли два класи B і C успадковуються від A, а клас D успадковується від обох B і C. Якщо в A є метод, який B і C перевизначили, а D не замістив його, тоді яку версію методу успадковує D: метод B чи метод C?

Наприклад, у контексті розробки програмного забезпечення для GUI, клас Button може успадкуватися від обох класів Rectangle (для зовнішнього вигляду) і Clickable (для функціональності/обробляння введення), а класи Rectangle і Clickable обидва успадковуються від класу Object . Тепер, якщо метод equals викликається для об'єкта Button, і такого методу немає в класі Button, але є заміщення методу equals у Rectangle або Clickable (або в обох), який метод слід викликати?

Це явище називають «алмазною проблемою» через форму діаграми успадкування класів у цій ситуації. У цьому випадку клас A знаходиться вгорі, B і C окремо під ним, а D об’єднує обидва внизу, утворюючи форму ромба.



Діаграма ромбоподібного успадкування.

Наслідування — такий цікавий принцип, якого більшість сучасних програмістів взагалі намагається уникати. Що таке наслідування? За Вікіпедії це так: «абстрактний тип даних може наслідувати дані і функціональність деякого існуючого типу, сприяючи повторному використанню компонентів програмного забезпечення».

Перекладаю на людський: один клас може наслідувати інший клас, його поля і методи. Що значить наслідувати? Перевикористати. Після того, як клас оголошує себе наслідувачем якогось класу, відповідні поля і методи з'являються в ньому автоматично. Цей принцип використовується в різних мовах. В Java це extence, в C ++ це двокрапка, в Ruby – трикутна дужка, і так далі.

Наслідування — це форма відносин між класами. Клас-наслідувач використовує методи класу-предка, але не навпаки. Наприклад, клас «собака» є наслідувачем класу «тварина», але «тварина» не буде наслідувати властивості класу «собака». Отже, наслідувач — це більш вузький клас порівняно з предком.

При цьому наслідування називається словом extence, що означає "розширення". Наприклад, ми вказуємо для класу «собака» поле «лапи» – а для класу «тварина» ми не можемо його використовувати, тому що у тварин часто зовсім немає лап, якщо це риба чи змія. Так що клас-наслідувач може розширювати властивості базового класу, використовуючи його код.

Наслідування буває **одиночне** і **множинне**. Одиночне — це коли один або кілька класів наслідуються тільки від одного базового класу. Множинне наслідування — коли клас наслідується від декількох базових класів. Множинне наслідування є в багатьох мовах: ви можете перенести дані і \ або поведінку з інших класів. Але, наприклад, в Java множинне наслідування обмежено і можливо тільки від певного типу класів, наприклад, від інтерфейсу — так, інтерфейс це теж клас.

Через що у багатьох мовах обмежують множинне наслідування? Через ромбовидного наслідування. Коли два класи наслідують властивості одного базового, а потім якийсь четвертий клас наслідує другий і третій одночасно. Виходить плутанина: незрозуміло, яка реалізація повинна використовуватися. У деяких мовах, наприклад, Scala, це вирішили за допомогою порядку записи. Але це не така вже й важлива проблема: зрештою, множинне наслідування не так вже необхідно, так що не таке велике це і обмеження.

Найважливіше. Раніше було прийнято наслідувати все від усього і перевикористати код виключно через наслідування. В результаті програмісти загрузли в позамежному рівні дерев наслідування. Кожен програміст придумував собі базовий клас (або декілька), від яких наслідувалось все. Типовою була ситуація, коли у класу був п'ятнадцятий або двадцятий рівень наслідування. У цих класах могло взагалі не бути коду, а назви у них були просто наркоманські. Ця мода привела до того, що безліч провідних програмістів переключилася на делегування замість наслідування. Це коли клас не буде наслідувати, а викликає інший клас. І вони, звичайно, мали рацію, але в результаті маятник хитнувся в інший бік.

Зараз багато початківці і не дуже програмісти вважають, що наслідування не треба використовувати ніколи, а треба використовувати делегування. На жаль, але таким чином вони стріляють собі в ногу. Припустимо, ви пишете кадрову систему. У вас є об'єкт типу «інженер», об'єкт типу «бухгалтер», об'єкт типу «менеджер». Якщо вони не є наслідниками від класу «person", а просто три окремих класи, то щоб підрахувати кількість співробітників компанії, вам потрібно перебрати всі три списки. А коли додасться новий вид співробітників, вам потрібно не забути змінити весь код, який підраховує співробітників, і додати в нього четвертий список. Якщо ж знадобиться підрахувати, наприклад, тільки тих співробітників, які знаходяться в офісі, ви з розуму зійдете. У вас

п’ять видів співробітників, які між собою не пов’язані між собою. Їх обробка займе купу часу, код виростає в рази. Це нерозумно.

Часто програмісти відмовлялися робити наслідування там, де воно буквально напрошувалося. Раджу: використовуйте наслідування, коли ці об’єкти дійсно виникають одна з одної. Нерозумно наслідувати непов’язаний об’єкт просто для того, щоб наслідувати властивості: тут краще застосувати делегування. Але в очевидних випадках, відмовившись від наслідування, ви вистрілите собі в ногу і створите масу проблем на рівному місці.

З багатьма фреймворками виникають питання, як правильно Меппен, як правильно працювати з деревами наслідування, що робити, і так далі. Але якщо ви знайомі з GoF-овськими патернами, згадайте: майже всі вони використовують наслідування і поліформізм. А справжній поліморфізм, як ви здогадуєтеся, без наслідування практично не працює. Тому, якщо ви повністю відмовляєтеся від наслідування, ви відмовляєтеся від всієї потужності ООП і відкочується в кам’яний вік, в процедурне програмування. Наслідування — це одна з головних сил ООП, і відмовлятися від неї нерозумно.

Поліморфізм

Існує декілька видів **поліморфізм** в програмуванні. Один реалізовується шляхом реалізації ряду підпрограм (функцій, процедур, методів тощо) з однаковими іменами, але з різними параметрами. Залежно від того, що передається, і вибирається відповідна підпрограма. Щодо власне ООП, то під поліморфізмом розуміється можливість використання об'єктів(екземплярів) підкласів взамін батьківського класу. При цьому в кожному класі реалізуються одні і ті ж підпрограми з одними і тими ж параметрами. Так можна реалізувати клас Тварина, з підкласами Свиня та Собака. Оскільки підкласи матимуть один інтерфейс взаємодії, тобто в кожному реалізований метод голос(), то можна підставляти в один і той же код, замість батьківського класу, об'єкт одного з підкласів і отримати різний звук або Гав, або Рох-рох. На практиці, підставновка реалізується шляхом присвоєнню змінній з типом батьківського класу (Тварина) посилання на об'єкт підкласу (на Свиня або Собака).

Поліморфізм (з грец. πολύς «багато» + μορφή «форма») — концепція в програмуванні та теорії типів, в основі якої лежить використання єдиного інтерфейсу для різнотипних сутностей або у використанні однакового символу для маніпуляцій над даними різного типу.

На противагу поліморфізму, концепція **мономорфізму** вимагає однозначного зіставлення.

Типи поліморфізму:

- Спеціалізований поліморфізм — коли функції з однаковою назвою реалізовують різну логіку для різних типів вхідних параметрів. Підтримується в багатьох мовах програмування через перевантаження функцій.
- Параметричний поліморфізм — коли код написаний без указування конкретного типу параметрів. В ООП це називається узагальнене програмування. Це основний тип поліморфізму в функційному програмуванні.
- Поліморфізм підтипів — коли під одним ім'ям може використовуватись декілька типів похідних від одного базового. Основний тип поліморфізму в ООП.

Взаємодія параметричного поліморфізму і підтипів призводить до понять варіативності та обмеженої квантифікації.

Приклади.

У контексті об'єктно-орієнтованого програмування найпоширенішим різновидом поліморфізму є здатність екземплярів підкласу грати роль об'єктів батьківського класу, завдяки чому екземпляри підкласу можна використовувати там, де використовуються екземпляри батьківського класу.

Прикладом спеціалізованого поліморфізму є бінарний оператор +, який може мати своїми аргументами дані різного типу: цілі числа, числа з рухомою комою, комплексні числа та навіть рядки:

- 1 + 2 — операнди цілого типу, результат цілого типу.
- 1.2 + 1.0e3 — операнди дійсних типів, результат дійсного типу
- «Бульдог» + «Носоріг» — операнди рядки, результат — конкатенований рядок

У наведеному далі прикладі на C++ в залежності від типу переданих даних будуть застосовуватись різні методи:

```
class Point {

  private:
   int x, y;
   char x2,y2;
  public:
    void setXY(int _x, int _y)
    {
      x=_x;
      y=_y;
    }
    void setXY(char _x, char _y)
    {
      x2=_x;
      y2=_y;
    }
};
```

Поліморфізм підтипів. Поліморфізм — один з трьох найважливіших механізмів об'єктно-орієнтованого програмування. Поліморфізм дозволяє створювати абстрактніші програми та підвищити коефіцієнт повторного використання коду.

Спільні властивості об'єктів об'єднуються в систему, яку можуть називати по різному: інтерфейс, клас. Спільність має зовнішнє і внутрішнє вираження. Зовнішня спільність проявляється як однаковий набір методів з однаковими іменами і сигнатурами (типами аргументів і результатів).

Внутрішня спільність є однакова функціональність методів. Її можна описати інтуїтивно виразити у вигляді строгих законів, правил, яким повинні підкорятись методи.

Наприклад:

```cpp
#include <iostream>

class Felid {
public:
 virtual void meow() = 0;
};

class Cat : public Felid {
public:
 void meow() { std::cout << "Meowing like a regular cat! meow!\n"; }
};

class Tiger : public Felid {
public:
 void meow() { std::cout << "Meowing like a tiger! MREOWWW!\n"; }
};

class Ocelot : public Felid {
public:
 void meow() { std::cout << "Meowing like an ocelot! mews!\n"; }
};

void do_meowing(Felid *cat) {
 cat->meow();
}

int main() {
 Cat cat;
 Tiger tiger;
 Ocelot ocelot;

 do_meowing(&cat);
 do_meowing(&tiger);
 do_meowing(&ocelot);
}
```

За часом вибору поліморфізм поділяють на **статичний** та **динамічний**:

– статичний поліморфізм (чи статична диспетчеризація) — якщо код, що буде виконуватись, вибирається під час компіляції;

– динамічний поліморфізм (чи динамічна диспетчеризація) — якщо код, що буде виконуватись, вибирається під час виконання програми (пізнє зв'язування).

Перевагами статичного поліморфізму є:

– кращий статичний аналіз коду компілятором,

– менший розмір коду (через кращу оптимізацію коду та відсутність таблиці віртуальних методів)

– швидший час виконання.

Зате динамічний поліморфізм:

– більш гнучкий,

– дозволяє качину типізацію,

– дозволяє використовувати динамічно приєднані бібліотеки для похідних типів.

Динамічний поліморфізм, зазвичай, це поліморфізм підтипів.

Статичний поліморфізм це ad hoc чи параметричний поліморфізм, хоча його також можна реалізувати через наслідування та шаблонне метапрограмування — це називається дивно рекурсивний шаблон.

Поліморфізм, напевно, найважливіший принцип ООП: на його основі будується величезна кількість патернів і рішень. Саме через поліморфізму ООП така потужна і класна штука, яка дозволяє не писати один і той же код по 50 разів, а дублювати його і використовувати для роботи з іншими даними.

Визначення поліморфізму – «здатність функції використовувати дані різних типів». Просто, щоб не просто. Якщо задуматися, то функція може використовувати дані різних типів по-різному. Є два типи поліморфізму: поліморфізм ad-hoc, тобто за запитом, і параметричний, або істинний поліморфізм.

Перший тип поліморфізму (ad-hoc) – це поліморфізм за запитом. Це банальні речі: приведення даних – коли метод отримує дані, наведені до того типу, з яким він зазвичай працює – і перевантаження методів, коли метод існує в декількох варіантах, з однаковою назвою, але різними параметрами. Це несправжній поліморфізм. Якщо це перевантаження, то це різні методи: вони мають різний тіло, який же це поліморфізм? Якщо ми говоримо про приведення даних, то тіло методу одне, але воно працює з одним типом даних, тому що до входу в метод параметр був перетворений в потрібний тип даних. Такий недороблений поліморфізм.

Параметричний, або справжній поліморфізм, це коли функція, одна і та ж, з одним і тим же тілом, може приймати в якості параметра дані різних класів. Як це можливо? Ну, наприклад, коли параметром функції є базовий клас для деякої ієрархії об’єктів. Тому функція може приймати будь-який з підкласів цього класу. Як ми говорили в минулій лекції, спадкоємець пов’язаний з базовим класом. Якщо параметр функції – базовий клас, будь спадкоємець може прийти туди і бути оброблений. Опрацьовано він може бути по-різному, може бути однаково, все залежить від внутрішньої структури цих об’єктів і того, як написаний метод.

Про корисність поліморфізму. Якось в книзі когось із великих програмістів я зустрів думка, яка спочатку мене здивувала. «Все IF в програмі можна замінити поліморфізмом». Думка про те, що всі умовні розгалуження в програмі можна замінити поліморфізмом, спочатку підірвала мені мозок. Але потім я зрозумів, що дійсно, будь-if в програмі можна замінити на поліморфізм, тобто Одна гілка йде в одного спадкоємця базового класу, інша гілка (else) – в іншого. Якщо немає ніякої гілки, то залишається порожнє місце – метод нічого не робить.

Якщо ви задумаєтеся, то зрозумієте, що поліморфізм допомагає зменшувати розмір програми на порядки. Саме за допомогою поліморфізму ви зможете забезпечити гнучкість і уникнути перевантаженості коду. Найстрашніший код, який попадався мені в житті – це п’ять тисяч рядків if-ів. І вся ця простирадло в п’ять тисяч рядків коду могла бути схлопнуться в невелике дерево успадкування. Це один з найпоширеніших способів рефакторінга. Поліморфізм якраз є способом уникнути заплутаного, складного і важко підтримуваного коду.