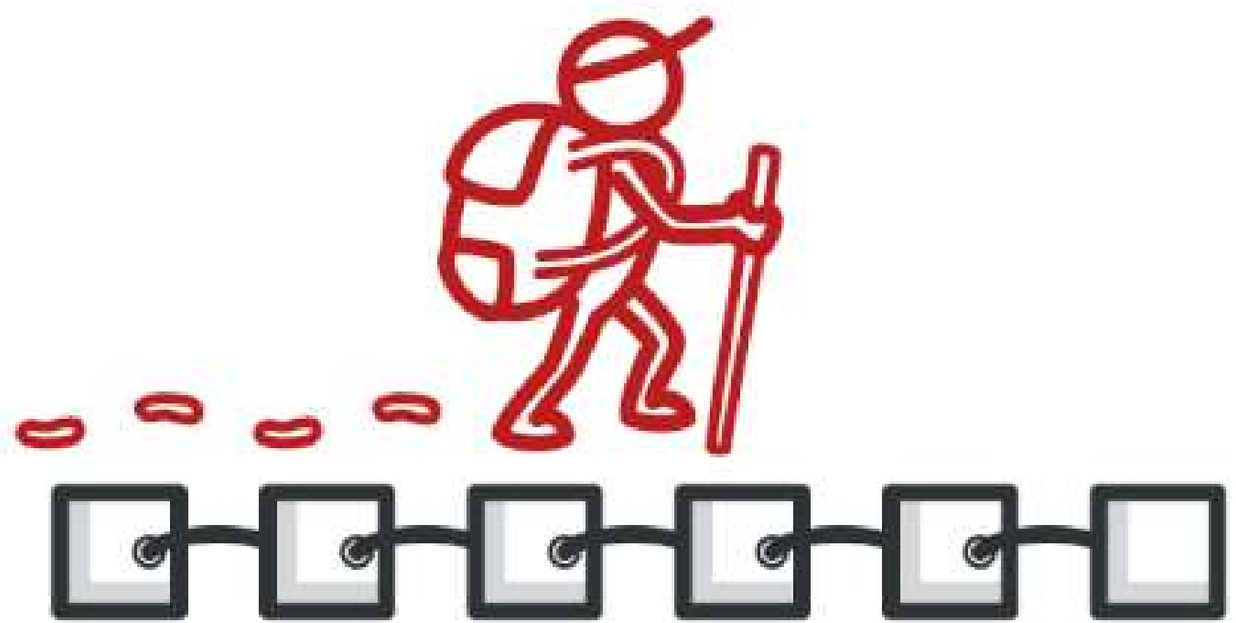
Ітератор

Ітератор — це поведінковий патерн проектування, що дає змогу послідовно обходити елементи складних об’єктів, не розкриваючи їхньої внутрішньої організації.



Проблема

У випадку коли ми маємо простий масив, можна обійтись звичайним циклом:

```
const unsigned SIZE = 5;
int* arr = new int[SIZE];
for (int i = 0; i < SIZE; ++i)
        cout << arr[i] << endl;
```

А що робити, коли структура даних дещо складніша, наприклад, зв'язний список?

```
LinkedList<int> list(arr, SIZE);
auto node = list.head; // порушення інкапсуляції!
while (node != nullptr) {
    cout << node->data << endl;
    node = node->next;
}
```

Якщо ми будемо напряму звертатися до складових компонентів списку (вузлів), то ми, по-перше, порушуємо принцип інкапсуляції, тобто доступаємося до складових внутрішньої реалізації об'єкту, а, по-друге, ускладнюємо клієнтський код.

Проблема

Тепер ми сховали внутрішню реалізацію від клієнта, натомість надаючи методи `size()` і `at()`.

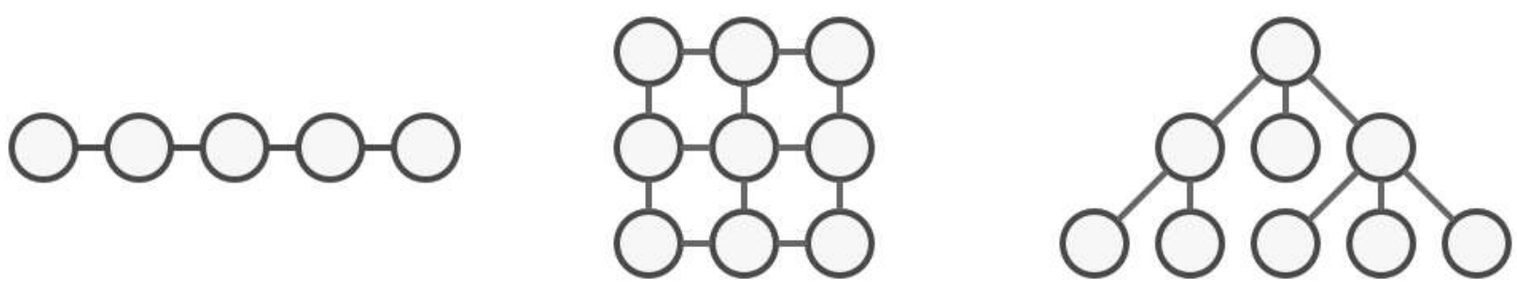
```
LinkedList<int> list(arr, SIZE);
for (int i = 0; i < list.size(); ++i)
{
    // LinkedList::at() is O(n) which is called n times
    // resulting in O(n^2) complexity for simple iteration!
    cout << list.at(i) << endl;
}
```

Але тут з'являються нові проблеми – реалізація методу `at()` полягає в тому, що ми кожен раз проходимось по списку, доки не доходимо до потрібного елементу. Це призводить до того, що простий прохід списку набуває часової складності $O(n^2)$, коли він не має бути довше ніж $O(n)$.

Проблема

А якщо у нас нелінійна структура даних (дерево, граф тощо), де ми маємо мати можливість обходу різними шляхами (BF/DF)?

Звісно, ми можемо реалізувати алгоритми обходу всередині самої колекції, але це розмиває основну задачу колекції (зберігання даних) і порушує **принцип єдиної відповідальності**.



Рішення

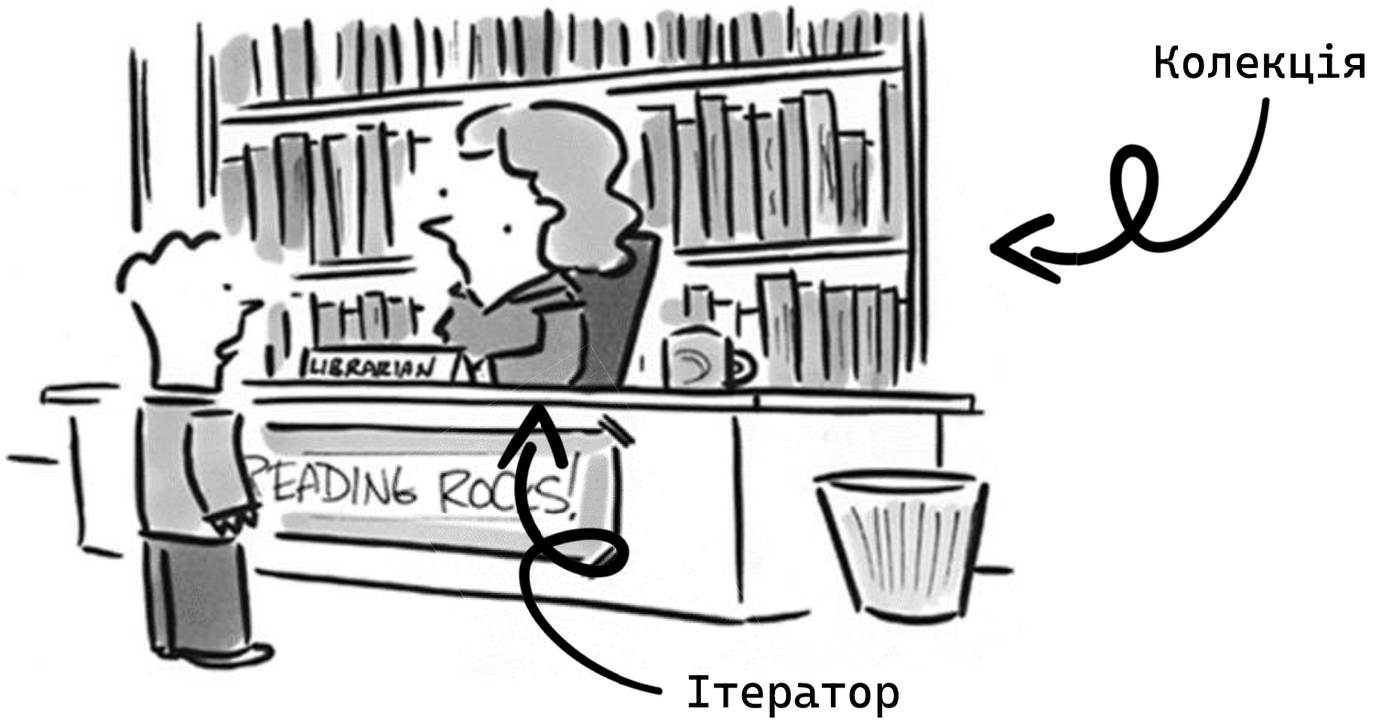
Рішення проблеми полягає у реалізації патерну Ітератор – винести поведінку обходу колекції в окремий об’єкт.

Таке рішення нам дає:

- Доступ до змісту колекції без розкриття реалізації (дотримання принципу єдиної відповідальності).
- Можливість визначати багато способів обходу.
- Єдиний інтерфейс для обходу різних колекцій (**забезпечення поліморфної ітерації**).

Що не так з цією діаграмою?

Приклад з життя

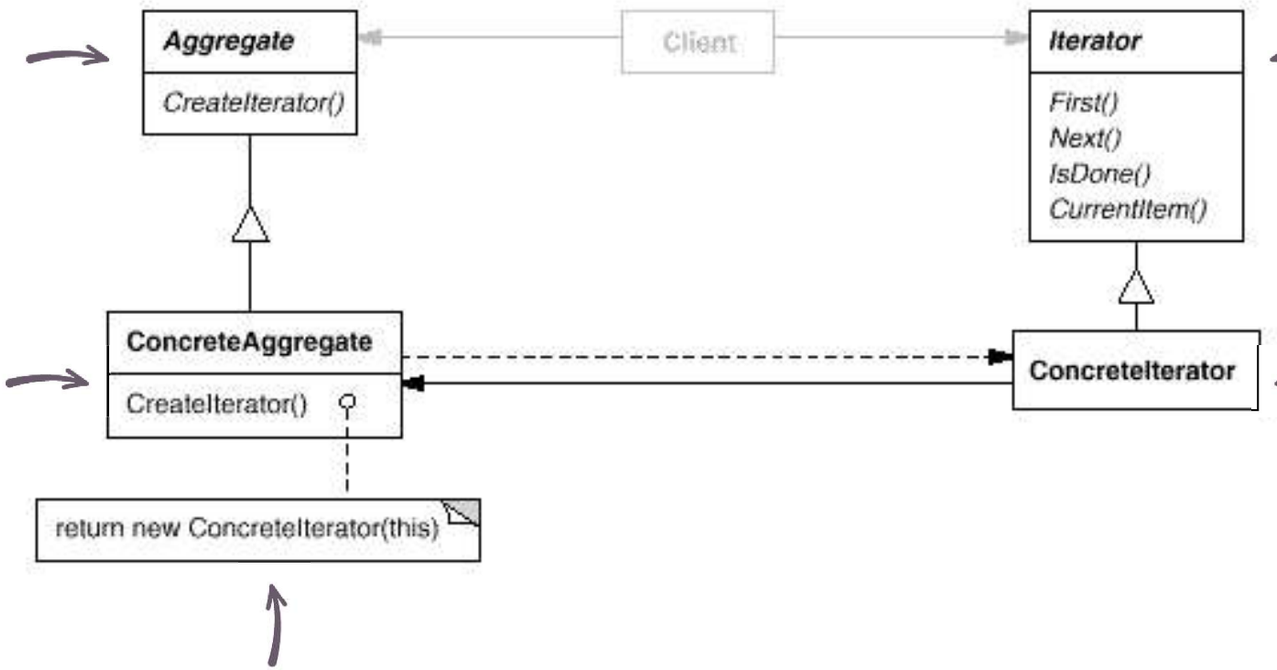


Книжкову полицю в бібліотеці можна уявити як колекцію з книгами, а бібліотекаря - як ітератор, який дозволяє клієнту отримати доступ до кожної книги в колекції.

Структура

Описує інтерфейс отримання ітератора з колекції.

Повертає новий екземпляр певного конкретного ітератора, зв'язавши його з поточним об'єктом колекції.



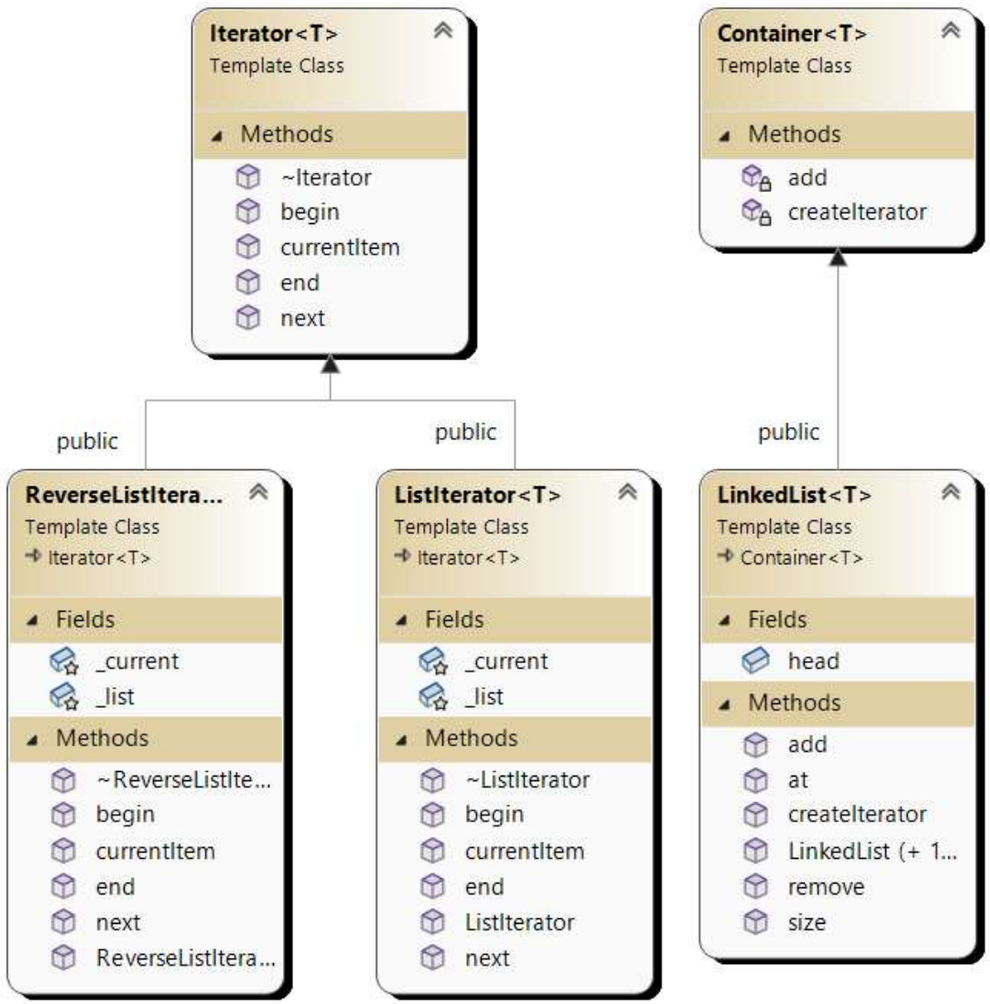
Описує інтерфейс для доступу та обходу елементів колекцій.

Реалізує алгоритм обходу якоїсь конкретної колекції.

CreateIterator is an example of a *factory method*. We use it here to let a client ask a list object for the appropriate iterator. (GoF)

Приклад використання

Приклад реалізації патерну Ітератор для вищезгаданої колекції LinkedList (діаграма згенерована у Visual Studio)



Приклад використання

З використанням ітератору код обходу колекції тепер виглядає так:

```
LinkedList<int> list(arr, SIZE);
auto iterator = list.createIterator();
for (iterator->begin(); !iterator->end(); iterator->next())
{
        cout << iterator->currentItem() << endl;
}
```

Тепер інкапсуляція не порушується і часова складність обходу O(n). Також з’явилася можливість визначати різні алгоритми обходу:

```
auto reverseIterator = make_unique<ReverseListIterator<int>>(&list);
for (reverseIterator->begin(); !reverseIterator->end(); reverseIterator->next())
{
        cout << reverseIterator->currentItem() << endl;
}
```

External/internal iterator

External iterator – ітерацію контролює клієнт, також клієнт має явно отримувати наступний елемент з ітератора. Вважаються набагато гнучкіше.

Internal iterator – ітерацію контролює ітератор, клієнт передає ітератору операцію для застосування, яку ітератор виконує на усіх елементах колекції. Вважаються більш обмеженими, але простіше у застосуванні.

Приклад internal ітератора на C++:

```
ListInternalIterator<int>(&list, [](int i) { cout << i << endl; return true; });
```

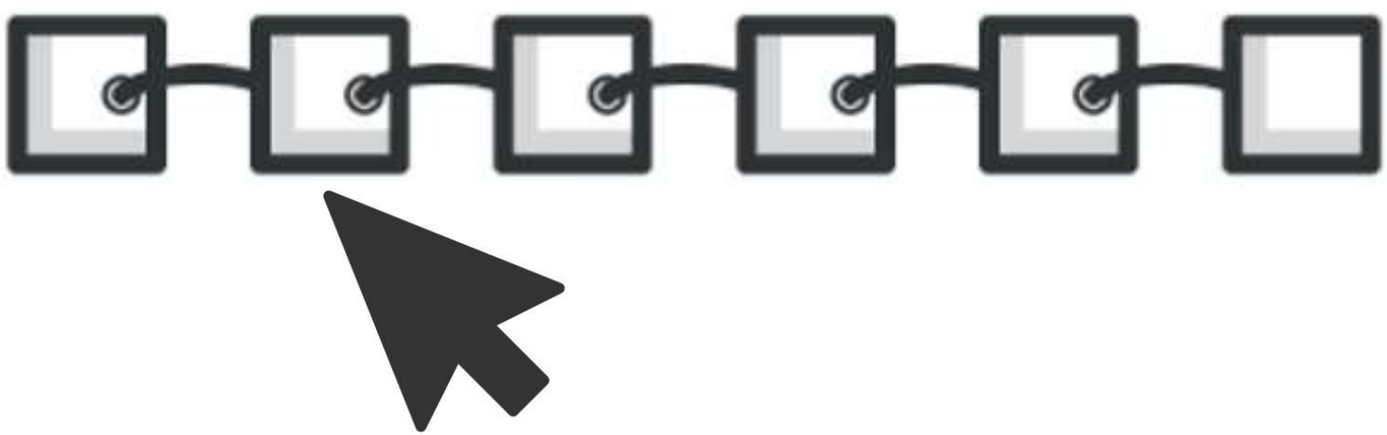
Приклад internal оператора, вбудованого в Java:

```
List<String> words;
words.stream().forEach(e -> System.out.println(e));
```

Cursor

Cursor – ітератор, який тільки зберігає поточну позицію, у той час як сам алгоритм обходу визначений у структурі. Клієнт має викликати метод `next(&cursor)` на колекції, аби змінити стан курсора і отримати наступний елемент.

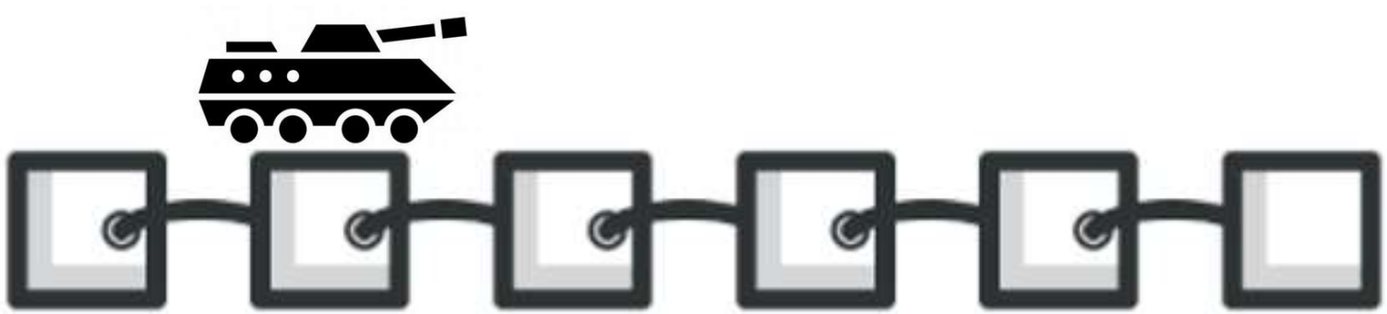
Cursors are a simple example of the Memento pattern and share many of its implementation issues. (GoF)



Robust iterator

При використанні ітераторів виникає нова проблема – якщо ми змінимо колекцію під час ітерації, то є ймовірність пропустити елемент, видати його двічі або інша непередбачувана поведінка. Найпримітивніше рішення – скопіювати колекцію, але це ресурсовитратно.

Рішення – реалізувати **robust ітератор**, який гарантує, що модифікації не впливатимуть на обхід. Один з способів реалізації – event listener, який буде повідомляти ітератор про зміни.



Null iterator

Null ітератор – це спеціальний тип ітератора, який використовується при обході деревоподібних структур для позначення вузлів, які не мають дочірніх елементів.

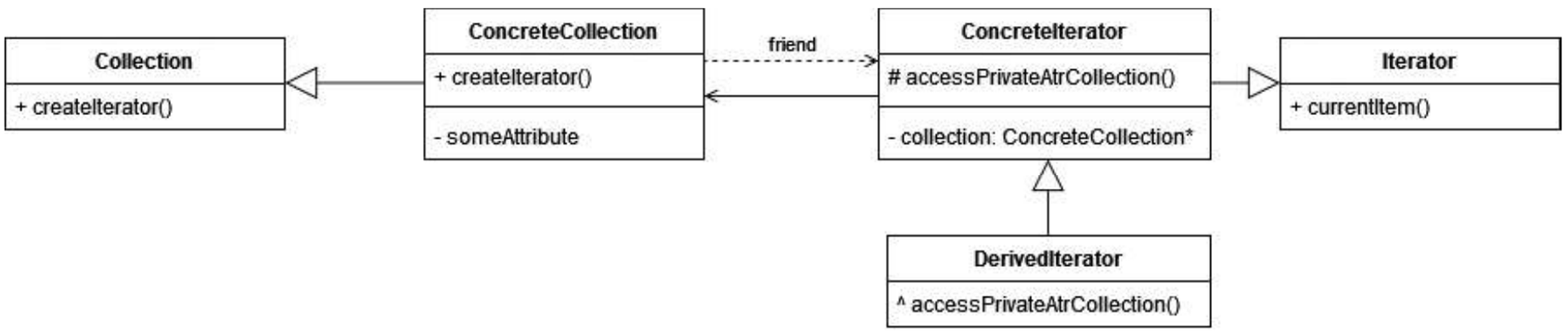
```
class TreeNode {
    std::unique_ptr<Iterator<T>> createIterator() override {
        if (_children.size() == 0)
                return std::make_unique<NullIterator<T>>(this);
        else return std::make_unique<TreeIterator<T>>(this);
    }
};
class NullIterator : public Iterator {
        bool end() override { return true; }
};
```

Iterator with privileged access

Ітератори можуть бути розглянуті як продовження колекції, з якою вони тісно пов'язані. Цей зв'язок можна визначити, зробивши ітератор дружнім класом колекції, тим самим надаючи ітератору доступ до внутрішньої реалізації.

Але це створює нову проблему – для визначення нових ітераторів, нам потрібно буде знову змінювати інтерфейс колекції для додання нового дружнього класу.

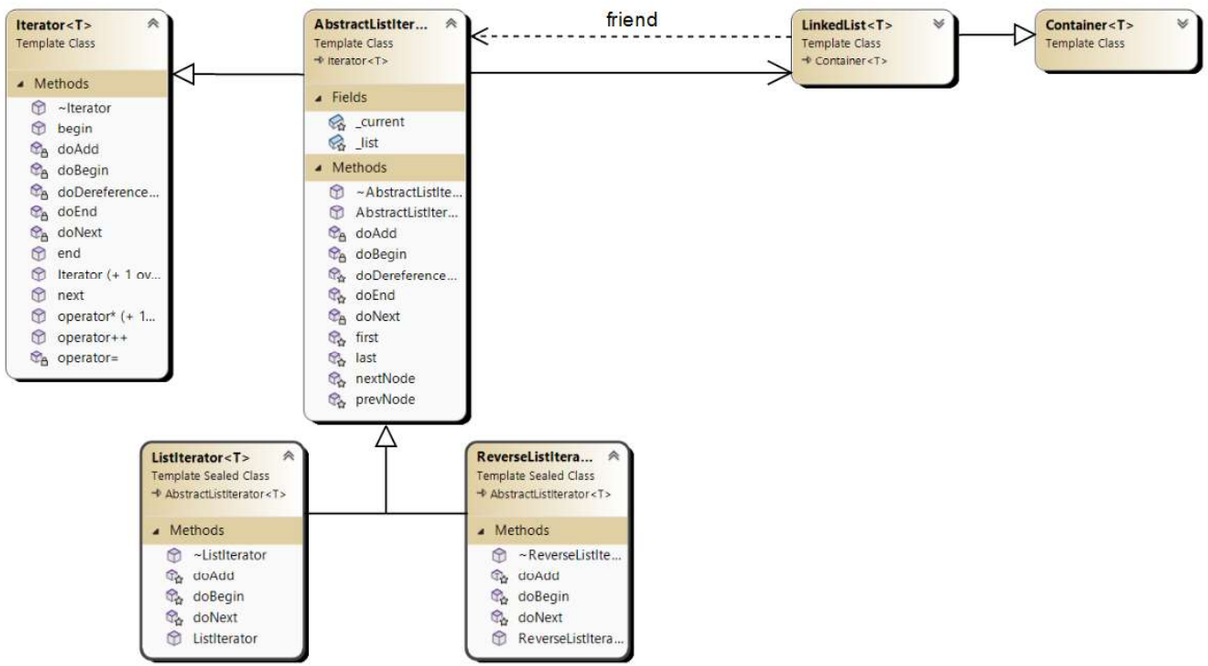
Один з шляхів вирішення – визначення одного головного типу ітератора, який буде дружнім класом колекції, і реалізація в ньому захищених методів, які міститимуть публічні поля колекції., а інші ітератори визначати похідними від головного.



Iterator with privileged access

Це рішення призводить до нових проблем, а саме до того, що ми тепер маємо конкретний базовий клас, що порушує **принцип нетермінальної абстракції**.

Можливе рішення – реалізувати базовий абстрактний клас, який наслідує інтерфейс ітератора і буде дружнім до колекції. Від цього класу будуть унаслідуватися вже конкретні термінальні класи, які реалізовуватимуть невизначені методи інтерфейсу потрібною поведінкою.



Cloning iterator

Припустимо, у нас з’явилася потреба створити новий екземпляр абстрактного ітератору, і саме такого самого типу, якого цей абстрактний ітератор набув. Для цього нам треба реалізувати операцію **клонування**.

```
virtual Iterator<T>& clone() = 0; // в інтерфейсі ітератора

ListIterator<T>& ListIterator<T>::clone() {
    return *(new ListIterator(_list)); // в конкретному класі
}

// клонований ітератор поводиться так само як оригінальний
Iterator<int>* iterator = new ListIterator<int>(&list);
auto clone = &iterator->clone();
```

Implicit iterator

Багато сучасних об'єктно-орієнтованих мов надають вбудований спосіб ітерації по елементах колекції без оголошення явного ітератора, тобто ітератор ініціалізований, але не показується в вихідному коді. Такий ітератор називається **неявним**.

Неявний ітератор у мові C++ виглядає так:

```
for (auto& el : list) {
        std::cout << el << std::endl;
}
```

Хоча насправді викликається це:

```
for (auto it = list.begin(); it != list.end(); ++it) {
       std::cout << *it << std::endl;
}
```

Generator

Генератор – це спеціальний тип ітератора, який визначається за допомогою функції або об'єкта, що генерує значення на льоту, замість того, щоб отримувати їх з колекції. При виклику генератора, він повертає ітератор, який можна використовувати для отримання згенерованих значень одне за одним.

Приклад мовою Python:

```
def fibonacci(limit):
    a, b = 0, 1
    for _ in range(limit):
        yield a
        a, b = b, a + b

for number in fibonacci(100): # Генератор створює ітератор
    print(number)
```

Типізація ітераторів у C++ STL

1) **Input iterator** – зчитування елементів колекції один за одним без редагування.

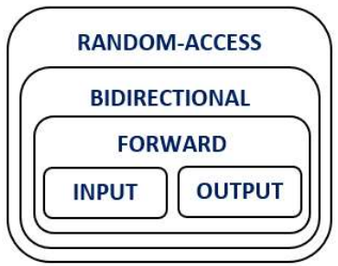
2) **Output iterator** – редагування елементів колекції один за одним без зчитування.

3) **Forward iterator** – обхід колекції в одному напрямку зі зчитуванням і редагуванням.

4) **Bidirectional iterator** – обхід колекції в обох напрямках зі зчитуванням і редагуванням.

5) **Random Access iterator** – довільний доступ до будь-якого елемента колекції.

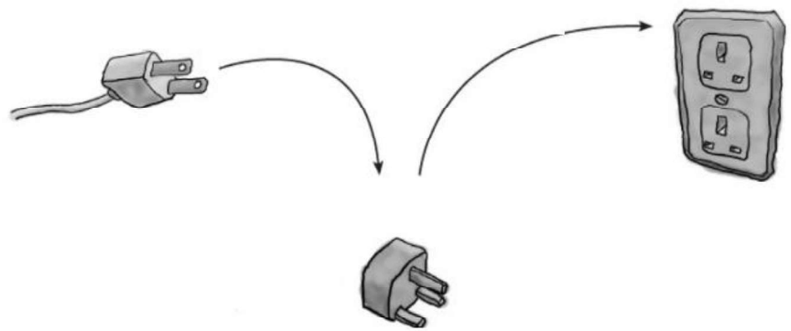
CONTAINER	TYPES OF ITERATOR SUPPORTED
Vector	Random-Access
List	Bidirectional
Deque	Random-Access
Map	Bidirectional
Multimap	Bidirectional
Set	Bidirectional
Multiset	Bidirectional
Stack	No iterator supported
Queue	No iterator supported
Priority Queue	No iterator supported



ITERATORS	PROPERTIES				
	ACCESS	READ	WRITE	ITERATE	COMPARE
Input	->	= *i		++	==, !=
Output			*i =	++	
Forward	->	= *i	*i =	++	==, !=
Bidirectional		= *i	*i =	++, --	==, !=
Random-Access	->, []	= *i	*i =	++, --, +=, -=, +, -	==, !=, <, >, <=, >=

Iterator Adaptors in STL

1) **Reverse iterator**.
   Adaptee: будь-який двосторонній ітератор.
   Utility: обертає напрям обходу колекції.



2) **Move iterator**.
   Adaptee: будь-який оператор введення та виведення.
   Utility: розіменування ітератора повертає rvalue.

3) **Stream iterator**.
   Adaptee: клас `istream`/`ostream`.
   Utility: надає зручний спосіб читання з потоку або запису до нього за допомогою ітератора.

4) **Insert iterator**.
   Adaptee: будь-який контейнер, що містить метод `insert()`.
   Utility: дозволяє вставляти елементи в контейнер за допомогою ітератора.

Переваги і недоліки

Pros	Cons
▪ Розділяє логіку зберігання даних в колекції від логіки обходу даних (single responsibility).	▪ Може бути «оверкілом» для простих структур даних.
▪ Дозволяє додавати нові способи обходу без втручання у реалізацію колекції (open/closed).	▪ При некоректному використанні може бути порушена інкапсуляція колекції.
▪ Менше повторюваного коду та сам код стає більш зрозумілішим.	
▪ Надає єдиний інтерфейс для реалізації обходу різних колекцій.	

Висновок: ітератор доречно використовувати, коли ми маємо складну структуру даних з неінтуїтивним алгоритмом обходу елементів, і для якої ми хочемо мати декілька різних способів обходу.

Де використовується?

Ітератори використовуються майже всюди, де є структури даних для зберігання інформації. Також вони доступні майже для кожної колекції у найуживаніших мовах програмування. Приклади використання патерну:

- **Результати запиту до БД**: проходитись по колекції виданих рядків один по одному.
- **Парсинг структурованих даних**: ітератор може бути використаний для проходження по даних у форматах JSON, XML, YAML, а також для парсингу веб-сторінок на HTML.
- **Обхід графів**: ітератор може бути використаний для проходження по графу чи іншій мережевій структурі (використовується для проходження по акаунтах в соц. мережах).
- **Обхід файлової системи**: ітератор використовується для обходження змісту директорії і ітерації по файлах і інших директоріях, які в ній містяться.
- **Компоненти користувацького інтерфейсу**: list view, table view і tree view використовують ітератори для проходження по зображених даних.

Взаємодія з іншими патернами

- **Composite**

  Ітератор може бути використаний для обходу дерева компонувальника.

- **Visitor**

  Ітератор відповідатиме за обхід структури, а відвідувач – за виконання дій над кожним елементом.

- **Factory method**

  Конкретні колекції створюють відповідні ітератори (поліморфна ітерація).

- **Memento**

  Знімок може бути використаний для збереження поточного стану ітератора для відновлення його у майбутньому.