ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN

ĐỀ TÀI: MẪU THIẾT KẾ ITERATOR TRONG LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG

Nhóm tác giả: Nhóm 09 – 18CLC1
Kiều Công Hậu – 18127259
Mai Đăng Khánh – 18127118
Huỳnh Nhật Nam – 18127014

Môn học: Phương pháp lập trình hướng đối tượng

Thành phố Hồ Chí Minh – 2019

LỜI CẢM ƠN

MỤC LỤC

DANH MỤC HÌNH

DANH MỤC BẢNG

DANH MỤC TỪ VIẾT TẮT

[1] **GoF**: Gang of Four - Bộ tứ huyền thoại: Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides – nhóm tác giả của cuốn sách "Design Patterns: Elements of Reusable Object-Oriented Software".

DANH MỤC THUẬT NGỮ

[1] **Class Diagram**: một loại sơ đồ cấu trúc tĩnh mô tả cấu trúc của hệ thống bằng cách hiển thị các lớp của hệ thống, thuộc tính, hoạt động của chúng và mối quan hệ giữa các đối tượng. Sơ đồ lớp là khối xây dựng chính của mô hình hướng đối tượng.

[2] **Stack**: một dạng cấu trúc dùng để lưu trữ dữ liệu trong lập trình.

[3] **Dynamic Array**: một dạng cấu trúc dùng để lưu trữ dữ liệu trong lập trình.

[4] **Linked List**: một dạng cấu trúc dùng để lưu trữ dữ liệu trong lập trình.

[5] **Singly Linked List**: một dạng cấu trúc dùng để lưu trữ dữ liệu trong lập trình.

[6] **Doubly Linked List**: một dạng cấu trúc dùng để lưu trữ dữ liệu trong lập trình.

[7] **Skip List**: một dạng cấu trúc dùng để lưu trữ dữ liệu trong lập trình.

[8] **Tree**: một dạng cấu trúc dùng để lưu trữ dữ liệu trong lập trình.

[9] **Hash Table**: một dạng cấu trúc dùng để lưu trữ dữ liệu trong lập trình.

CHƯƠNG 1. VẤN ĐỀ THỰC TẾ

1. Dạo đầu

Ở chương này, chúng ta sẽ tạm gác qua "mẫu thiết kế" (design pattern) là gì mà chỉ đi tìm ra lý do tại sao khái niệm "mẫu thiết kế" lại được ra đời.

2. Tình huống thực tế

Kevin, Harry và Ned đang là những sinh viên năm nhất của trường đại học Harvard. Họ đang cùng nhau viết một phần mềm quản lý trường học. Cả ba bắt đầu bàn bạc và chia ra các class cần thiết của dự án này cho từng người đảm nhiệm.

Class diagram mà nhóm sinh viên đã thiết kế cho phần mềm này

Kevin được phân công viết class ***cStudent*** để quản lý thông tin của 1 sinh viên và class ***cStudentCollection*** để quản lý tất các các sinh viên trong trường. Kevin quyết định sử dụng cấu trúc **Linked List** để lưu trữ các ***cStudent*** trong ***cStudentCollection***. Sau đây là đoạn code mà Kevin đã cài đặt:

```
1.  /* Khai báo class cStudent */
2.  class cStudent
3.  {
4.  public:
5.      cStudent(string name = "N/A", string id = "N/A");
6.      void input_info();        //Nhập thông tin của sinh viên từ bàn phím
7.      void show_info();         //In thông tin của sinh viên ra màn hình
8.      cStudent*& next();        //Trả về con trỏ next của sinh viên này (linked list)
9.
10. private:
11.     string Name;
12.     string ID;
13.     cStudent* Next;
14. };
15.
16. /* Định nghĩa class cStudent */
17. cStudent::cStudent(string name, string id)
18. {
19.     this->Name = name;
20.     this->ID = id;
21.     this->Next = nullptr;
22. }
23.
24. void cStudent::input_info()
25. {
26.     cout << "Input info for this Student" << endl;
27.     cout << "  - Name: ";
28.     getline(cin, this->Name);
29.     cout << "  - ID: ";
30.     getline(cin, this->ID);
31. }
32.
33. void cStudent::show_info()
34. {
35.     cout << "  - Type: Student" << endl;
36.     cout << "  - Name: " << this->Name << endl;
37.     cout << "  - ID: " << this->ID << endl;
38. }
39.
40. cStudent*& cStudent::next()
41. {
42.     return this->Next;
43. }
44.
45.
46. /* Khai báo class cStudentCollection */
47. class cStudentCollection
48. {
49.     friend class cMinistry;
50. public:
51.     cStudentCollection();
52.     ~cStudentCollection();
53.
54. private:
55.     cStudent* Head;           //Con trỏ trỏ tới phần tử đầu tiên của linked list
56. };
57.
58. /* Định nghĩa class cLecturerCollection */
59. cStudentCollection::cStudentCollection()
60. {
61.     this->Head = nullptr;
62. }
63.
64. cStudentCollection::~cStudentCollection()
65. {
```

```
66.     cStudent* temp;
67.     while (this->Head)
68.     {
69.         temp = this->Head;
70.         this->Head = temp->next();
71.         delete temp;
72.     }
73. }
```

Harry thì viết class ***cLecturer*** để quản lý thông tin của 1 giảng viên và class ***cLecturerCollection*** để quản lý tất cả các giảng viên trong trường. Harry quyết định sử dụng cấu trúc **Stack** để lưu trữ các ***cLecturer*** trong ***cLecturerCollection***. Sau đây là phần cài đặt của Harry:

```
1.  /* Khai báo class cLecturer */
2.  class cLecturer
3.  {
4.  public:
5.      cLecturer(string name = "N/A", string id = "N/A");
6.      void input_info();
7.      void show_info();
8.
9.  private:
10.     string Name;
11.     string ID;
12. };
13.
14. /* Định nghĩa class cLecturer */
15. cLecturer::cLecturer(string name, string id)
16. {
17.     this->Name = name;
18.     this->ID = id;
19. }
20.
21. void cLecturer::input_info()
22. {
23.     cout << "Input info for this Lecturer" << endl;
24.     cout << "  - Name: ";
25.     getline(cin, this->Name);
26.     cout << "  - ID: ";
27.     getline(cin, this->ID);
28. }
29.
30. void cLecturer::show_info()
31. {
32.     cout << "  - Type: Lecturer" << endl;
33.     cout << "  - Name: " << this->Name << endl;
34.     cout << "  - ID: " << this->ID << endl;
35. }
36.
37.
38. /* Khai báo class cLecturerCollection */
39. class cLecturerCollection
40. {
41.     friend class cMinistry;
42. public:
43.     cLecturerCollection();
44.     ~cLecturerCollection();
45.
46. private:
```

```
47.     static const int MAX = 100;      //Số lượng giảng viên tối đa
48.     cLecturer* Lecturers[MAX];
49.     int Count;                       //Số lượng giảng viện hiện tại
50. };
51.
52. /* Định nghĩa class cLecturerCollection */
53. cLecturerCollection::cLecturerCollection()
54. {
55.     this->Count = 0;
56.     for (int i = 0; i < this->MAX; ++i)
57.         this->Lecturers[i] = nullptr;
58. }
59.
60. cLecturerCollection::~cLecturerCollection()
61. {
62.     for (int i = 0; i < this->Count; ++i)
63.         delete this->Lecturers[i];
64. }
65.
```

Còn Ned viết class ***cMinistry*** để quản lý tất cả các phương thức của ban giáo vụ đối với tập hợp những sinh viên và giảng viên trong trường. Sau đây là phần cài đặt của Ned:

```
1.  /* Khai báo class cMinistry */
2.  class cMinistry
3.  {
4.  public:
5.      cMinistry(cLecturerCollection* lecturerCollection, cStudentCollection* student
    Collection);
6.      void show_list_lecturers();      // In ra danh sách các giảng viên trong trường
7.      void show_list_students();       // In ra danh sách các sinh viên trong trường
8.      void add_one_lecturer(const cLecturer& lecturer);  // Thêm 1 giáo viên
9.      void add_one_student(const cStudent& student);  // Thêm 1 sinh viên
10.
11. private:
12.     cLecturerCollection* LecturerCollection;
13.     cStudentCollection* StudentCollection;
14. };
15.
16. /* Định nghĩa class cMinistry */
17. cMinistry::cMinistry(cLecturerCollection* lecturerCollection, cStudentCollection*
    studentCollection)
18. {
19.     this->LecturerCollection = lecturerCollection;
20.     this->StudentCollection = studentCollection;
21. }
22.
23. void cMinistry::show_list_lecturers()
24. {
25.     cout << "~ List of lecturers ~" << endl;
26.     for (int i = 0; i < this->LecturerCollection->Count; ++i)
27.     {
28.         cout << " #" << i + 1 << endl;
29.         this->LecturerCollection->Lecturers[i]->show_info();
30.     }
31. }
32.
33. void cMinistry::show_list_students()
34. {
35.     cout << "~ List of students ~" << endl;
```

```
36.     int i = 0;
37.     for (cStudent* st = this->StudentCollection->Head; st != nullptr; st = st-
    >next())
38.     {
39.         cout << " #" << i + 1 << endl;
40.         st->show_info();
41.         i++;
42.     }
43. }
44.
45. void cMinistry::add_one_lecturer(const cLecturer& lecturer)
46. {
47.     if (this->LecturerCollection->Count == this->LecturerCollection->MAX)
48.         throw;  // Full
49.
50.     *this->LecturerCollection->Lecturers[this->LecturerCollection-
    >Count++] = lecturer;
51. }
52.
53. void cMinistry::add_one_student(const cStudent& student)
54. {
55.     cStudent* newStudent = new cStudent(student);
56.     newStudent->next() = this->StudentCollection->Head;
57.     this->StudentCollection->Head = newStudent;
58. }
59.
```

3. Vấn đề gặp phải

Rắc rối bắt đầu nảy sinh khi Ned bắt tay vào cài đặt class ***cMinistry*** sau khi Kevin và Harry đã hoàn thành phần cài đặt cho các class của mình.

Về căn bản, class ***cMinistry*** mà Ned đảm nhiệm sẽ chứa 2 đối tượng tập hợp (aggregate object) là tập hợp các sinh viên (***cStudentCollection***) và tập hợp các giảng viên (***cLecturerCollection***) trong trường. Và nhiệm vụ của class ***cMinistry*** là phải cung cấp các phương thức liên quan đến 2 tập hợp trên (Ví dụ: Liệt kê danh sách các sinh viên, giảng viên có trong trường; Thêm / Bớt sinh viên và giảng viên; Sắp xếp danh sách theo thứ tự tăng dần theo tên; Liệt kê các sinh viên được nhận học bổng;…).

Nhưng trở ngại lớn nhất là mỗi tập hợp lại có 1 cấu trúc khác nhau (**Linked List** của ***cStudentCollection*** và **Stack** của ***cLecturerCollection***), dẫn tới cách duyệt các phần tử trong các tập hợp đó là khác nhau. Lúc này, Ned phải đi hỏi Kevin và Harry về cấu trúc bên trong của 2 class tập hợp mà 2 người này đã viết kèm với cách để duyệt từng phần tử đối với mỗi tập hợp ấy (***cLecturerCollection*** thì xài vòng lặp for và index, ***cStudentCollection*** thì xài vòng lặp for và con trỏ next). Có vẻ rắc rối nhỉ!!!

Tới đây, Ned đã tốn một ít thời gian để hiểu được cấu trúc của 2 class tập hợp trên và cách để duyệt từng phần tử trong 2 tập hợp ấy và hoàn thành được 2 phương thức là `void show_list_lecturers();` và `void show_list_students();`. May mắn là vì class

cMinistry chỉ quản lý có 2 tập hợp là các sinh viên và các giảng viên trong trường, bên cạnh đó 2 class này có cấu trúc cũng khá đơn giản, nên vẫn khá nhẹ nhàng cho Ned. Nhưng sẽ là vấn đề nan giải nếu số tập hợp cần quản lý có tận 5 – 10 loại, mỗi loại có một loại cấu trúc đơn giản và phức tạp khác nhau (Stack, Dynamic Array, Singly Linked List, Doubly Linked List, Skip List, Tree, Hash Table,…). Điều này dẫn tới Ned sẽ phải hiểu cấu trúc và ghi nhớ từng tập hợp duyệt theo từng cách tương ứng nào.

Thậm chí sẽ còn tệ hơn nếu Kevin hoặc Harry thay đổi cấu trúc của 2 class tập hợp của mình sau khi Ned hoàn thành phần cài đặt class ***cMinistry***. Điều này dẫn tới cách duyệt từng phần tử trong những tập hợp bị thay đổi không còn hợp lệ nữa và Ned sẽ phải hỏi 2 người bạn của mình về cấu trúc mới và cách để duyệt từng phần tử trong loại tập hợp đó, sau đó Ned sẽ phải ngồi sửa lại một đống code đã viết.

“Không đời nào! =.=” – Ned nói.

4. Giải quyết

Để giải quyết được những vấn đề mà Kevin, Harry và Ned đang gặp phải, ta phải sử dụng **mẫu thiết kế Iterator** (Iterator design pattern) – một trong những mẫu thiết kế rất phổ biến trong lập trình hướng đối tượng. Mẫu thiết kế này như thế nào, cách sử dụng và cài đặt ra sao, lợi ích và tác hại của nó sẽ được trình bày cụ thể ở các chương sau.

CHƯƠNG 2. MẪU THIẾT KẾ

1. Dạo đầu

Chương này sẽ cho ta thấy cái nhìn tổng quan nhất về các mẫu thiết kế và tại sao phải sử dụng các mẫu thiết kế trong lập trình trước khi đi sâu vào tìm hiểu cụ thể mẫu thiết kế Iterator và cách ứng dụng của nó để giải quyết vấn đề của nhóm 3 sinh viên trên.

2. Mẫu thiết kế là gì?

Trong công nghệ phần mềm, mẫu thiết kế là một giải pháp rất điển hình để giải quyết các vấn đề chung trong thiết kế hệ thống. Mẫu thiết kế không phải là đoạn code hoàn chỉnh mà bạn có thể copy nó vào chương trình và chạy, mà nó chỉ đơn thuần là cách tư duy hay khung sườn mô tả cách giải quyết các vấn đề chung trong lập trình thông qua việc thể hiện các mối quan hệ và sự tương tác giữa các class với nhau một cách hợp lý.

Nhiều người sẽ nhầm lẫn khái niệm “mẫu thiết kế” (design pattern) với “thuật toán” (algorithm) vì cả 2 đều thể hiện cách để giải quyết một vấn đề. Tuy nhiên, mẫu thiết kế mô tả cách giải quyết vấn đề ở mức độ tổng quát hơn và có thể áp dụng vào các chương trình khác nhau nhưng các thuật toán để xử lý từng chương trình cụ thể thì khác nhau. Một liên tưởng thực tế khá đơn giản là Kevin, Harry và Ned đều tham gia cuộc thi nấu ăn MasterChef, yêu cầu của giám khảo là cả 3 người phải nấu món “Phở bò Việt Nam” với công thức chuẩn mà họ đưa ra. Lúc này có thể hiểu rằng công thức chuẩn ấy chính là ‘design pattern”, còn cách mỗi người áp dụng công thức và chế biến là “algorithm”, vậy nên các món ăn sau khi hoàn thành dù cho áp dụng 1 công thức vẫn sẽ có mùi vị khác nhau.

3. Tại sao ta nên tìm hiểu về các mẫu thiết kế?

Các mẫu thiết kế là bộ công cụ rất mạnh mẽ bao gồm các giải pháp đã được thử nghiệm và kiểm tra cho các vấn đề phổ biến trong thiết kế phần mềm. Ngay cả khi bạn không va chạm với các vấn đề ấy, thì việc hiểu rõ các mẫu thiết kế sẽ giúp bạn cải thiện hoàn toàn tư duy lập trình hướng đối tượng của mình và giải các bài toán khác bằng cách áp dụng những nguyên tắc trong thiết kế hướng đối tượng đã được giới thiệu trong các mẫu thiết kế ấy (The SOLID Design Principle,…).

Các mẫu thiết kế định nghĩa một loại ngôn ngữ chung giúp cho bạn và các đồng đội của mình có thể giao tiếp một cách hiệu quả. Ví dụ, Ned có thể nói rằng: “Chúng ta có thể dùng Iterator để giải quyết vấn đề này.”, thì sẽ rất tuyệt nếu cả Kevin và Harry đều hiểu “Iterator”

là cái gì. Tức Ned sẽ không cần sẽ phải giải thích ý tưởng thiết kế của anh ta cho hai cậu bạn của mình.

Áp dụng các mẫu thiết kế sẽ làm cho phần mềm của chúng ta trở nên gọn gàng hơn, dễ hiểu hơn, linh hoạt hơn, dễ nâng cấp và bảo trì hơn.

4. Phân loại các mẫu thiết kế

Hiện nay có khoảng 23 mẫu thiết kế điển hình và rất thông dụng, được chia làm 3 nhóm:

- Nhóm khởi tạo (**Creational Patterns**): Các mẫu thiết kế này cung cấp cơ chế để khởi tạo nhiều loại đối tượng khác nhau, giúp tăng sự linh hoạt và khả năng tái sử dụng cho code.
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Nhóm cấu trúc (**Structural Patterns**): Các mẫu thiết kế này giải thích cách để lắp ráp các đối tượng và các lớp thành một cấu trúc lớn hơn mà vẫn giữ được sự linh hoạt và hiệu quả của code.
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- Nhóm hành vi (**Behavioral Patterns**): Các mẫu này liên quan đến thuật toán và sự phân công trách nhiệm giữa các đối tượng.
  - Chain of responsibility
  - Command
  - Interpreter
  - **Iterator**
  - Mediator
  - Memento

- Observer
- State
- Strategy
- Template Method
- Visitor

CHƯƠNG 3. ITERATOR

1. Dạo đầu

Sau khi có cái nhìn tổng quan về các mẫu thiết kế, ta sẽ đi sâu hơn về mẫu thiết kế Iterator và cách ứng dụng nó vào giải quyết vấn đề mà Ned, Kevin và Harry đang gặp phải (Chương 1).

2. Ý tưởng

Tóm tắt các vấn đề mà nhóm sinh viên này gặp phải:

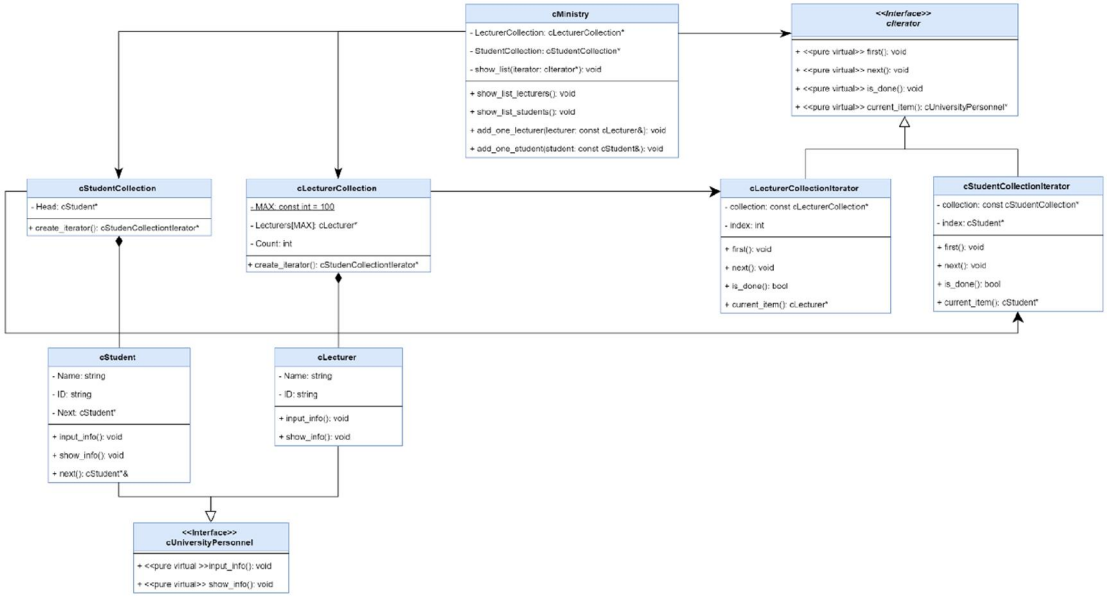
- 2 class ***cStudentCollection*** và ***cLecturerCollection*** được cài đặt tách rời nhau mà không có một "giao diện" (interface) chung.
- 2 class tập hợp trên lưu trữ các phần tử với 2 cấu trúc khác nhau nên cách duyệt từng phần tử trong mỗi tập hợp sẽ khác nhau, dẫn tới class ***cMinistry*** phải hiểu được cấu trúc sâu bên trong của 2 class tập hợp ấy để có thể duyệt được.
- Nếu ***cMinistry*** quản lý thêm 1 tập hợp nữa thì sẽ có thêm 1 cách duyệt khác xuất hiện.
- Nếu một trong các class tập hợp mà ***cMinistry*** quản lý thay đổi cấu trúc lưu trữ thì ***cMinistry*** cũng phải sửa code theo vì cách duyệt mảng cũ có thể không còn phù hợp nữa.

Sẽ rất tuyệt nếu ta tìm được cách cài đặt 2 class ***cStudentCollection*** và ***cLecturerCollection*** với cùng một interface và đồng bộ 1 cách duyệt mảng duy nhất là:

```
1.  for (object.first(); !object.is_done(); object.next())
2.  {
3.       object.current_item().show_info();
4.  }
```

3. Giải quyết

Để giải quyết được những vấn đề trên, ta sẽ áp dụng mẫu thiết kế Iterator. Mục đích là để mỗi một đối tượng tập hợp như **cLecturerCollection** và **cStudentCollection** sẽ tự chu cấp cho **cMinistry** một cách thức để duyệt từng phần tử mà không làm lộ cấu trúc bên trong của các đối tượng tập hợp đó. Đặc biệt hơn nữa là các cách thức duyệt phần tử này đều sử dụng cùng một giao diện, điều này đã thỏa mãn với mong muốn của chúng ta ở phần lên ý tưởng. Dưới đây là phần cài đặt và giải thích code sau khi áp dụng mẫu thiết kế Iterator:



Sơ đồ lớp của chương trình sử dụng mẫu thiết kế Iterator

Trước hết, ta sẽ cài đặt thêm 2 class mới toanh là **cLecturerCollectionIterator** và **cStudentCollectionIterator**, cả 2 class này đều sử dụng chung một giao diện là **cIterator**. 2 class trên sẽ cung cấp cho 2 class **cLecturerCollection** và **cStudentCollection** mỗi cách thức duyệt phần tử tương ứng nhưng thống nhất. Mỗi đối tượng được tạo ra từ 2 class này được gọi là iterator, trong đó, mỗi đối tượng iterator sẽ lưu giữ địa chỉ của đối tượng tập hợp tương ứng (***collection***) và trạng thái của phần tử hiện tại mà đối tượng này đang trỏ tới (***index***). Bên cạnh đó, iterator còn có các chức năng cơ bản như trỏ tới phần tử đầu tiên của đối tượng tập hợp (***first()***), trỏ tới phần tử kế tiếp (***next()***), kiểm tra xem iterator đã duyệt đến phần tử cuối cùng trong tập hợp ấy chưa (***is_done()***) và trả về địa chỉ của phần tử hiện tại mà iterator này đang trỏ đến (***current_item()***). Dưới đây là phần cài đặt code cho 3 class mới vừa nêu trên:

```
1.  /* Khai báo class cLecturerCollectionIterator */
2.  class cLecturerCollectionIterator : public cIterator
3.  {
4.  public:
5.      cLecturerCollectionIterator(const cLecturerCollection* lecturerCollection);
6.      void first() override;
7.      void next() override;
8.      bool is_done() override;
9.      cLecturer* current_item();
10.
11. private:
12.     const cLecturerCollection* collection;
13.     int index;
14. };
15.
16. /* Định nghĩa class cLecturerCollectionIterator */
```

```
17. cLecturerCollectionIterator::cLecturerCollectionIterator(const cLecturerCollection
    * lecturerCollection)
18. {
19.     this->collection = lecturerCollection;
20.     this->index = 0;
21. }
22.
23. void cLecturerCollectionIterator::first()
24. {
25.     index = 0;
26. }
27. void cLecturerCollectionIterator::next()
28. {
29.     ++index;
30. }
31. bool cLecturerCollectionIterator::is_done()
32. {
33.     return index == collection->Count;
34. }
35. cLecturer* cLecturerCollectionIterator::current_item()
36. {
37.     return collection->Lecturers[index];
38. }
39.
40.
41. /* Khai báo class cStudentCollectionIterator */
42. class cStudentCollectionIterator : public cIterator
43. {
44. public:
45.     cStudentCollectionIterator(const cStudentCollection* lecturerCollection);
46.     void first();
47.     void next();
48.     bool is_done();
49.     cStudent* current_item();
50.
51. private:
52.     const cStudentCollection* collection;
53.     cStudent* index;
54. };
55.
56. /* Định nghĩa class cStudentCollectionIterator */
57. cStudentCollectionIterator::cStudentCollectionIterator(const cStudentCollection* s
    tudentCollection)
58. {
59.     this->collection = studentCollection;
60.     this->index = collection->Head;
61. }
62.
63. void cStudentCollectionIterator::first()
64. {
65.     index = collection->Head;
66. }
67. void cStudentCollectionIterator::next()
68. {
69.     index = index->next();
70. }
71. bool cStudentCollectionIterator::is_done()
72. {
73.     return index == nullptr;
74. }
75. cStudent* cStudentCollectionIterator::current_item()
76. {
77.     return index;
78. }
79.
```

```
80.
81. /* Khai báo interface cIterator */
82. class cIterator
83. {
84. public:
85.     virtual void next() = 0;
86.     virtual void first() = 0;
87.     virtual bool is_done() = 0;
88.     virtual cUniversityPersonnel* current_item() = 0;
89. };
```

Bây giờ, ta sẽ thêm các phương thức để khởi tạo một iterator tương ứng với mỗi loại class tập hợp (***create_iterator()***). (Vì cách cài đặt 2 class **cLecturerCollection** và **cStudentCollection** tương tự như cách chưa sử dụng mẫu thiết kế Iterator – ngoài trừ thêm phương thức ***create_iterator()*** ở mỗi class, **cLecturerCollection** có friend class là **cLecturerCollectionIterator** và **cStudentCollection** có friend class là **cStudentCollectionIterator** – nên ta sẽ lướt qua.)

```
1.  cIterator* cLecturerCollection::create_iterator()
2.  {
3.      return new cLecturerCollectionIterator(this);
4.  }
5.
6.  cIterator* cStudentCollection::create_iterator()
7.  {
8.      return new cStudentCollectionIterator(this);
9.  }
```

Vậy là ta đã xây dựng xong các class có khả năng tạo ra loại đối tượng có thể duyệt từng phần tử trong các loại đối tượng tập hợp khác nhau nhưng xài cùng một giao diện – iterator. Phần sau đây sẽ thể hiện rõ ràng cách sử dụng loại đối tượng đặc biệt này (iterator) trong việc cài đặt class **cMinistry**:

```
1.  /* Khai báo class cMinistry */
2.  class cMinistry
3.  {
4.  public:
5.      cMinistry(cLecturerCollection* lecturerCollection, cStudentCollection* student
    Collection);
6.      void show_list_lecturers();
7.      void show_list_students();
8.      void add_one_lecturer(const cLecturer& lecturer);
9.      void add_one_student(const cStudent& student);
10.
11. private:
12.     cLecturerCollection* LecturerCollection;
13.     cStudentCollection* StudentCollection;
14.
15. private:
16.     void show_list(cIterator* iterator);
17. };
18.
19. /* Định nghĩa class cMinistry */
```

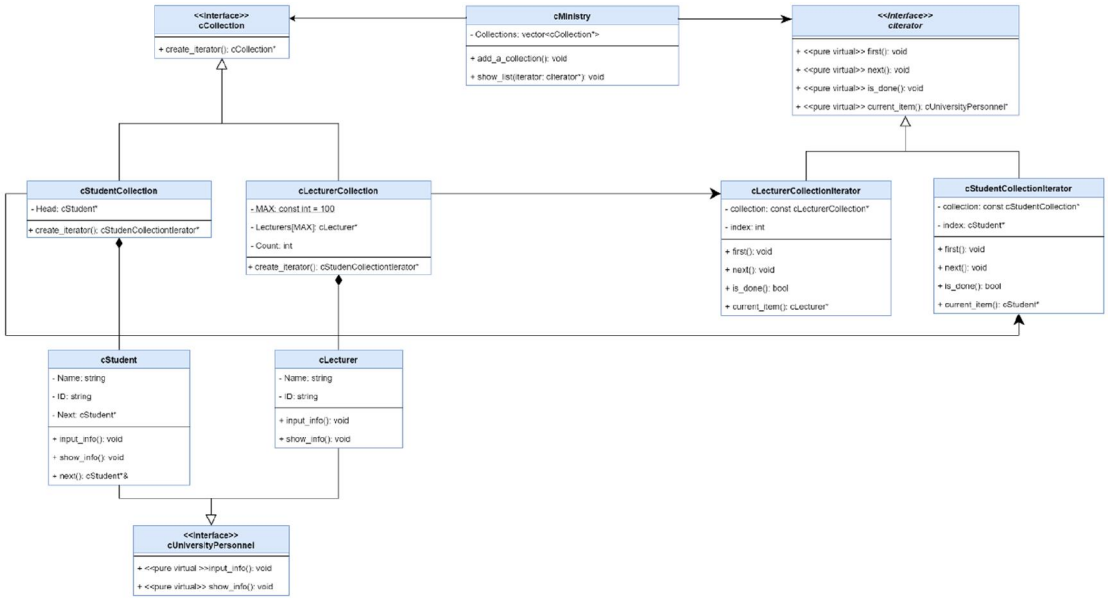
```
20. cMinistry::cMinistry(cLecturerCollection* lecturerCollection, cStudentCollection*
    studentCollection)
21. {
22.     this->LecturerCollection = lecturerCollection;
23.     this->StudentCollection = studentCollection;
24. }
25.
26. void cMinistry::show_list_lecturers()
27. {
28.     cIterator* iterator = this->LecturerCollection->create_iterator();
29.     cout << "~ List of lecturers ~" << endl;
30.     this->show_list(iterator);
31.     delete iterator;
32. }
33.
34. void cMinistry::show_list_students()
35. {
36.     cIterator* iterator = this->StudentCollection->create_iterator();
37.     cout << "~ List of students ~" << endl;
38.     this->show_list(iterator);
39.     delete iterator;
40. }
41.
42. void cMinistry::add_one_lecturer(const cLecturer& lecturer)
43. {
44.     if (this->LecturerCollection->Count == this->LecturerCollection->MAX)
45.         throw;  // Full
46.
47.     *this->LecturerCollection->Lecturers[this->LecturerCollection-
    >Count++] = lecturer;
48. }
49.
50. void cMinistry::add_one_student(const cStudent& student)
51. {
52.     cStudent* newStudent = new cStudent(student);
53.     newStudent->next() = this->StudentCollection->Head;
54.     this->StudentCollection->Head = newStudent;
55. }
56.
57. void cMinistry::show_list(cIterator* iterator)
58. {
59.     int i = 0;
60.     for (iterator->first(); !iterator->is_done(); iterator->next())
61.     {
62.         cout << " #" << i + 1 << endl;
63.         iterator->current_item()->show_info();
64.         ++i;
65.     }
66. }
```

Đến đây, ta có thể thấy rõ lợi ích mà iterator mang lại trong việc cài đặt class **cMinistry**:

- Tất cả các loại đối tượng tập hợp trong **cMinistry** đều xài chung một phương thức để duyệt qua từng phần tử trong tập hợp ấy mà không cần quan tâm cấu trúc sâu bên trong của từng loại đối tượng tập hợp.
- Ta có thể viết thêm một hàm ***show_list(...)*** với tham số truyền vào là loại iterator tương ứng (áp dụng tính đa hình) để rút gọn đoạn code khi cài đặt 2 hàm ***show_list_students()*** hay ***show_list_lecturers()***.

4. Cải tiến

Thật ra, để tối ưu hóa sự hiệu quả và linh hoạt hơn nữa cho cách giải quyết trên, ta có thể tạo thêm một class nữa là **cCollection** để làm giao diện chung cho 2 class tập hợp **cLecturerCollection** và **cStudentCollection**. Trong class **cCollection** sẽ có một phương thức ảo là ***create_iterator()***. Mục đích của việc làm này là lợi dụng tính đa hình để class **cMinistry** có thể quản lý đa dạng các loại tập hợp hơn, tùy thuộc vào cách mà ta khởi tạo cho nó.



Sơ đồ lớp của chương trình quản lý trường học sau khi cải tiến

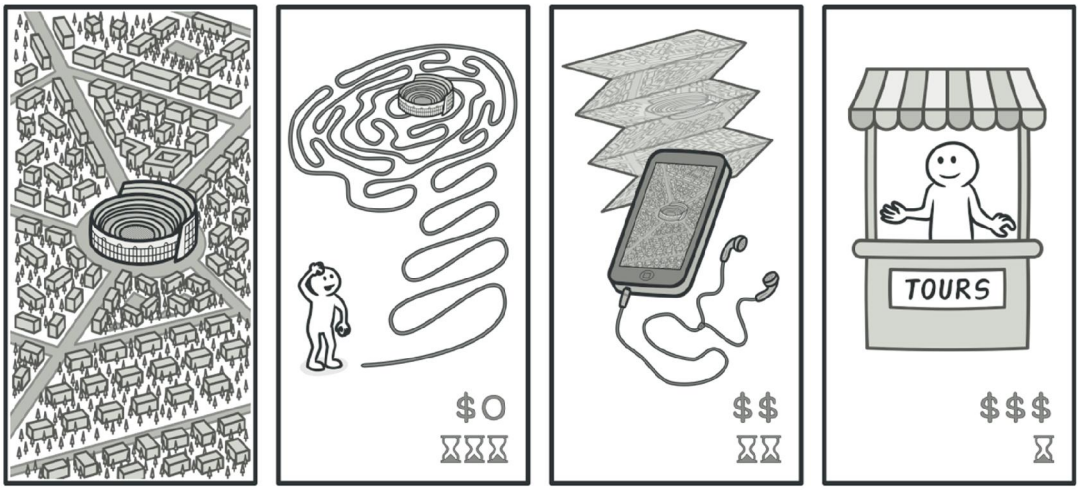
5. Khái niệm

Phía trên đã trình bày cách sử dụng mẫu thiết kế Iterator vào giải quyết vấn đề của nhóm sinh viên Harvard. Bây giờ ta sẽ tìm hiểu về khái niệm chính thống của mẫu thiết kế hành vi này:

"The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation." - GoF.

Tạm hiểu là: Mẫu thiết kế Iterator *(The Iterator Design Pattern)* cung cấp cách thức để truy cập vào các phần tử *(the elements)* của một đối tượng tập hợp *(an aggregate object)* một cách tuần tự mà không làm lộ cấu trúc bên dưới của đối tượng đó.

6. Liên tưởng thực tế



Có nhiều cách để tham quan thành Rome

Bạn đang lên kế hoạch cho chuyến du lịch vài ngày đến thành Rome và tham quan hết tất cả những danh lam thắng cảnh ở đó. Và bạn đang nhức đầu cân nhắc giữa 3 phương án mà bạn nghĩ ra để chuẩn bị cho chuyến tham quan này:

- Tham quan tự túc
  - Chi phí rẻ bèo.
  - Nhưng tốn nhiều thời gian vì bạn tìm hướng đi đến các danh lam theo cảm tính.
- Mua ứng dụng chỉ đường cho smart-phone
  - Tốn một ít tiền cho việc tải app.
  - Nhưng bù lại bạn sẽ có thiết bị chỉ đường nên sẽ tiết kiệm thời gian hơn phương án đầu tiên.
- Thuê hướng dẫn viên du lịch địa phương
  - Tốn kém.
  - Nhưng vừa tiết kiệm thời gian, vừa được hướng dẫn viên thuyết minh chi tiết về từng danh lam thắng cảnh. Chuyến đi sẽ thú vị hơn.

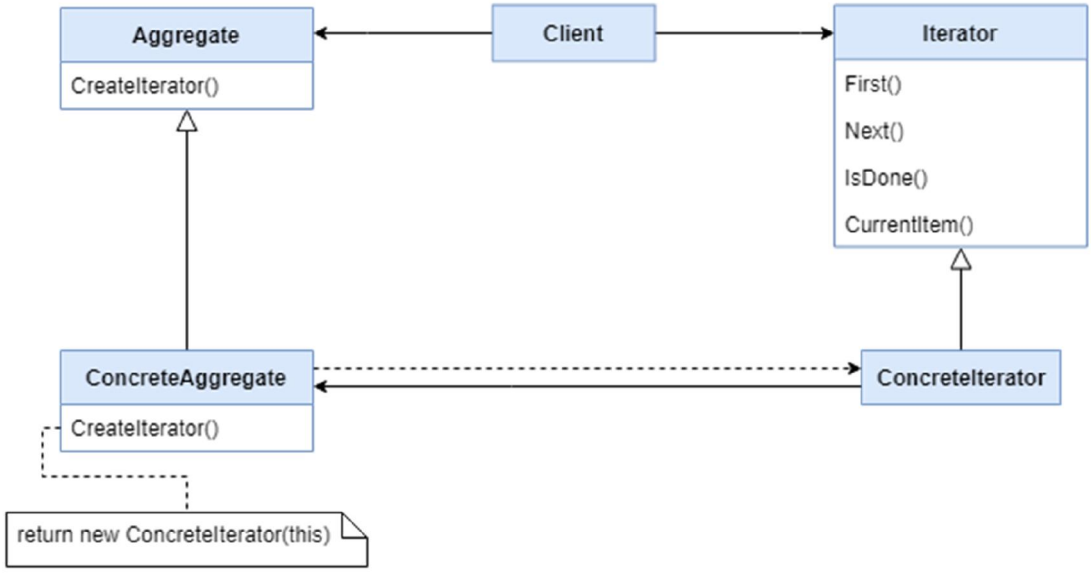
Cả 3 lựa chọn trên: hướng đi ngẫu nhiên phát sinh trong đầu bạn – ứng dụng chỉ đường trên smart-phone – hướng dẫn viên du lịch địa phương – đều hoạt động như các iterator duyệt qua các phần tử là các danh lam thắng cảnh của đối tượng tập hợp thành Rome.

Ví dụ này có nét tương đồng với vấn đề mà ta đặt ra ở chương 1. Ned đóng vai trò là người lên kế hoạch tham quan thành Rome, còn Kevin và Harry đóng vai trò là những hướng dẫn viên du lịch địa phương. Thành Rome ở đây chính là các đối tượng tập hợp

cStudentCollection và ***cLecturerCollection***, còn các danh lam thắng cảnh sẽ là các phần tử ***cStudent*** và ***cLecturer*** tương ứng với mỗi tập hợp.

- Đối với phương án 1, vì Ned không biết gì về 2 đối tượng tập hợp đó nên việc duyệt các phần tử trong tập hợp sẽ rất khó khăn và mất nhiều thời gian.
- Đối với phương án 2, Ned sẽ nhờ 2 "hướng dẫn viên du lịch địa phương" Kevin và Harry viết 2 "phần mềm chỉ đường" – tức là iterator – để duyệt các phần tử trong tập hợp.
- Đối với phương án 3, Ned sẽ nhờ Kevin và Harry review cụ thể đoạn code mà 2 người này đã cài đặt cho 2 đối tượng tập hợp ***cStudentCollection*** và ***cLecturerCollection*** cho mình, khi biết rõ được cấu trúc của 2 class tập hợp này, Ned có thể tự tin duyệt từng phần tử trong 2 tập hợp đó.

7. Cấu trúc tổng quát



Class Diagram thể hiện cấu trúc tổng quát của mẫu thiết kế Iterator

Trong đó:

- **Iterator**
  - Định nghĩa một giao diện cho việc truy cập và duyệt các phần tử trong tập hợp.
- **ConcreteIterator**
  - Cài đặt cho giao diện trên đối với mỗi loại Iterator tương ứng.
  - Lưu giữ vị trí hiện tại trong quá trình duyệt từng phần tử trong tập hợp.
- **Aggregate**
  - Định nghĩa một giao diện cho việc khởi tạo Iterator tương ứng với từng tập hợp.

- **ConcreteAggregate**
  - Cài đặt phần giao diện khởi tạo Iterator để trả về một thể hiện của đối tượng ConcreteAggregate phù hợp. (return new ConcreteIterator(this);)

8. Khả năng ứng dụng

Sử dụng mẫu thiết kế này để:

- Truy cập các phần tử của đối tượng tập hợp mà không làm lộ cấu trúc phức tạp bên trong của nó. (Tăng tính thuận tiện và bảo mật)
- Hỗ trợ nhiều cách thức để duyệt cho một đối tượng tập hợp. (Duyệt theo chiều xuôi, duyệt theo chiều ngược,…)
- Cung cấp một giao diện thống nhất để duyệt các phần tử duy nhất đối với mọi tập hợp có cấu trúc lưu trữ dữ liệu khác nhau. (Polymorphic Iteration)
- Giảm sự trùng lặp code cho các vòng lặp.

9. Lợi ích và hạn chế

Lợi ích mà mẫu thiết kế này mang lại:

- Đảm bảo nguyên tắc Đơn Trách Nhiệm (**Single Responsibility Principle**: *"Each class has only one responsibility, and therefore has only one reason to change."* – Tạm dịch: Mỗi class chỉ giữ một trách nhiệm, chính vì thế mỗi một class chỉ có duy nhất một lý do để thay đổi.): Tách rời thuật toán truy xuất phần tử của tập hợp khỏi lớp tập hợp. Như vậy lớp tập hợp chỉ gồm những phương thức thao tác trên các phần tử của tập hợp, lớp Iterator chỉ gồm những phương thức truy xuất tập hợp.
- Đảm bảo nguyên tắc Mở/Đóng (**Open/Closed Principle**: "*Software modules should be closed for modifications but open for extensions*" – Tạm dịch: Mỗi module hoặc class nên hạn chế thay đổi nhưng ưu tiên mở rộng.): Bằng việc tách rời kể trên, ta có thể thêm các chức năng mới, sử dụng cho tập hợp mà không cần thay đổi cấu trúc của lớp tập hợp. Theo nguyên tắc "ưu tiên mở rộng, hạn chế thay đổi".
- Ta có thể truy xuất cùng một tập hợp một cách song song. Vì mỗi iterator chứa trạng thái truy xuất riêng của nó. Tức là có thể có nhiều iterator liên kết đến một đối tượng tập hợp.
- Ta cũng có thể tạm dừng một công việc truy xuất và tiếp tục khi cần thiết.
- Iterator cung cấp một giao thức thống nhất để truy xuất những cấu trúc tập hợp khác nhau bằng cách hỗ trợ đa hình.

- Tránh thất thoát hoặc dư thừa dữ liệu khi thêm sửa xóa tập hợp.

Hạn chế của mẫu thiết này:

- Áp dụng mẫu thiết kế này có thể quá phức tạp nếu chỉ cần truy xuất một tập hợp đơn giản.
- Sử dụng Iterator có thể kém hiệu quả hơn so với việc truy xuất trực tiếp đối với một số tập hợp đặc biệt.

10. Các mẫu thiết kế liên quan

Mẫu thiết kế Iterator có thể kết hợp được với nhiều mẫu thiết kế khác để tăng sự linh động cho chương trình:

- Sử dụng **Iterator** để duyệt một tập hợp sử dụng mẫu thiết kế **Composite**. (Trees)
- Sử dụng mẫu thiết kế **Factory Method** cùng với **Iterator** để hỗ trợ đa hình.
- Sử dụng mẫu thiết kế **Memento** cùng với **Iterator** để lưu giữ trạng thái hiện tại của iterator và quay trở lại nếu cần thiết.

Sử dụng mẫu thiết kế **Visitor** với **Iterator** để duyệt các đối tượng tập hợp cấu trúc dữ liệu phức tạp và cài đặt một số tính năng tác động trực tiếp lên các phần tử của tập hợp đó, kể cả khi lớp tập hợp này quản lý nhiều tập hợp con khác nhau.

11. Một số bài toán khác có áp dụng mẫu thiết kế Iterator

Trong thực tế, mẫu thiết kế này được áp dụng khá rộng rãi và phổ biến:

- Chương trình có nhiều cấu trúc dữ liệu tập hợp như tree, list, array,… Để có thể thống nhất cách duyệt các tập hợp trên ta có thể áp dụng mẫu thiết kế này.
- Kiểu dữ liệu **vector** của C++ có tích hợp sẵn mẫu thiết kế Iterator, cho phép ta truy xuất dữ liệu một cách tuần tự mà không cần biết phương thức mà các phần tử được lưu trữ và liên kết
- Trong ngôn ngữ **Python**, mẫu thiết kế Iterator được sử dụng trong hầu hết các kiểu dữ liệu tập hợp (iterable), cho phép ta có thể truy xuất các phần tử bằng một cú pháp chung có dạng:

```
for element in iterable:
    # do something with element
```

PHỤ LỤC

TÀI LIỆU THAM KHẢO

[1] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software".

[2] Eric Freeman, Elisabeth Freeman, Bert Bates, Kathy Sierra, "Head First Design Patterns".

[3] Dmitri Nesteruk, "Design Patterns in Modern C++: Reusable Approaches for Object-Oriented Software Design".

[4] https://refactoring.guru/design-patterns/iterator