UNIVERSITY OF SCIENCE

ADVANCE PROGRAM IN COMPUTER SCIENCE

GROUP 9

VISITOR AND ITERATOR

SEMINAR CS202

Ho Chi Minh City, 12/2022

UNIVERSITY OF SCIENCEN

ADVANCE PROGRAM IN COMPUTER SCIENCE

VISITOR AND ITERATOR

SEMINAR REPORT

ADVISOR

Nguyễn Lê Hoàng Dũng

Hồ Tuấn Thanh

Đặng Minh Triết - 21125096

Nguyễn Hữu Trọng - 21125167

Vương Quốc Phong - 21125087

Nguyễn Đình Tùng - 21125171

Ho Chi Minh City, 12/2022

Contents

Chapter 1

Visitor Design Pattern

1.1 Real World Problem

Consider we are developing an application for a company. The app can manage in products of the company, each in there own categories. However, near the end, when our software is already in deployment for testing, the customer asks us to develop a new feature to help them manage the products from the distributor and products to be sold to customer. The details of the feature is following.

The company can buy products in each categories, and sell the products to the customers. Each time, when a distributor come, they will deliver an amount of goods in categories, the company will pay money and store the goods. Same thing with a customer, they will come and buy a list of items from each catgories. The company can maintain a list of products in each category, the total number of their remaining products in that category.

1.2 Naive Solution

As we give this job to a junior developer, they have not learn much about Object Oriented Programming, they presents the following UML diagram

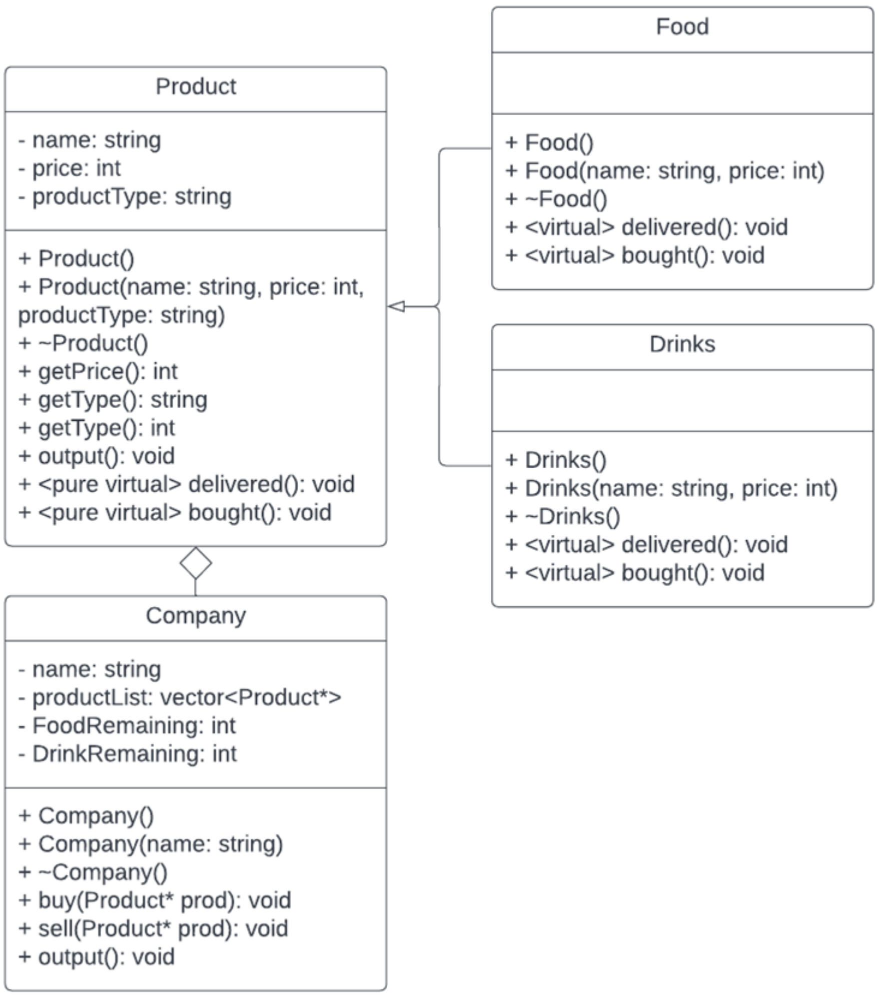

Product
- name: string
- price: int
- productType: string
+ Product()
+ Product(name: string, price: int, productType: string)
+ ~Product()
+ getPrice(): int
+ getType(): string
+ getType(): int
+ output(): void
+ <pure virtual> delivered(): void
+ <pure virtual> bought(): void
Food
+ Food()
+ Food(name: string, price: int)
+ ~Food()
+ <virtual> delivered(): void
+ <virtual> bought(): void
Drinks
+ Drinks()
+ Drinks(name: string, price: int)
+ ~Drinks()
+ <virtual> delivered(): void
+ <virtual> bought(): void
Company
- name: string
- productList: vector<Product*>
- FoodRemaining: int
- DrinkRemaining: int
+ Company()
+ Company(name: string)
+ ~Company()
+ buy(Product* prod): void
+ sell(Product* prod): void
+ output(): void

Fig 1.1: Naive solution

They propose a simple implementation by creating delivered() and bought() method, they'll be used in the buy and sell method in class Company to execute the task. Code implementations can be found in folder Code/Visitor/Example1/Naive .

1.3 Problem of the naive solution

The naive solution will create delivered and bought function for each children class. The solution works fine if you make everything from scratch. However, for a software product that has been already in producion, this solution tends to be bad. In depths, each of the class require an extra implementation if a new method is added, this might cause bugs in the software. The completed class is created in order to do specific types of job and may see the new extra implementation as an alien class. Further changes can also be modified in the future so the naïve solution would do really bad here. Thanks to the visitor design pattern we can solve the problem more efficient.

1.4 Introduction to the Visitor Pattern

Visitor is a behavioral design pattern that lets you separate algorithms from the objects on which they operate.

The original object that had to perform the behavior is now passed to one of the visitor's methods as an argument, providing the method access to all necessary data contained within the object.

The visitor class will define a set of methods, which takes argument of different types, as the example shown.

In order to call the proper method for the responding class, we will have to check the object classes.

Instead of using conditionals in the client class, we will use a technique called Double Dispatch. The main idea of the technique is, instead of letting the client select the proper method to call, we will delegate this choice to the objects we are passing to the vistor as an argument.

The objects will have an "accept" function in their interface, and will be implement in concrete classes to "accept" a visitor and tell it what visiting method should be used.

1.5 General Diagram

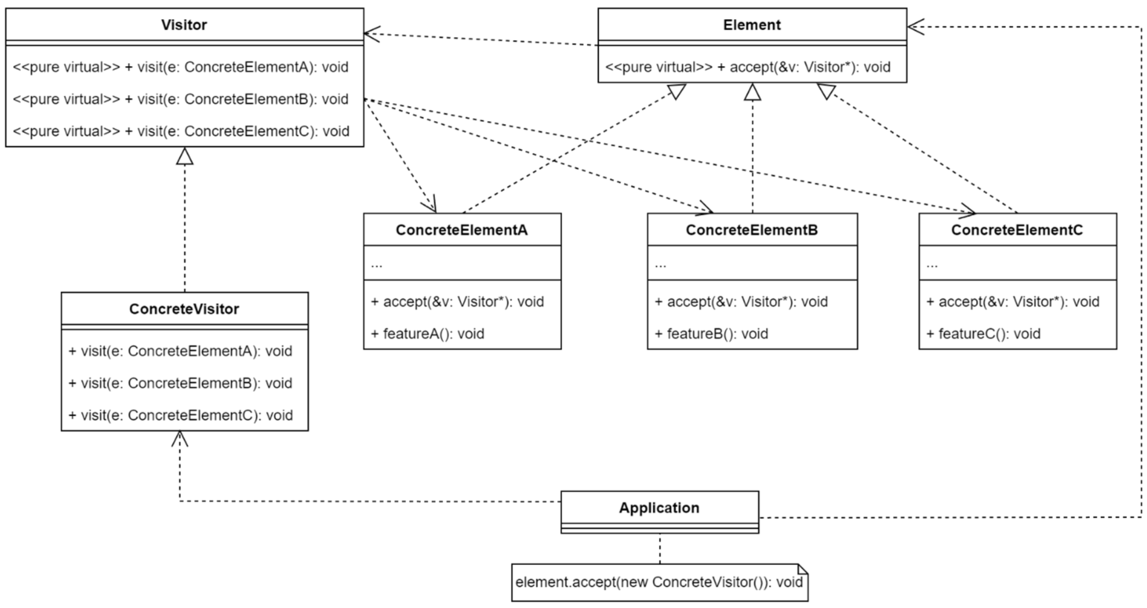

Visitor
<<pure virtual>> + visit(e: ConcreteElementA): void
<<pure virtual>> + visit(e: ConcreteElementB): void
<<pure virtual>> + visit(e: ConcreteElementC): void
Element
<<pure virtual>> + accept(&v: Visitor*): void
ConcreteElementA
...
+ accept(&v: Visitor*): void
+ featureA(): void
ConcreteElementB
...
+ accept(&v: Visitor*): void
+ featureB(): void
ConcreteElementC
...
+ accept(&v: Visitor*): void
+ featureC(): void
ConcreteVisitor
+ visit(e: ConcreteElementA): void
+ visit(e: ConcreteElementB): void
+ visit(e: ConcreteElementC): void
Application
element.accept(new ConcreteVisitor()): void

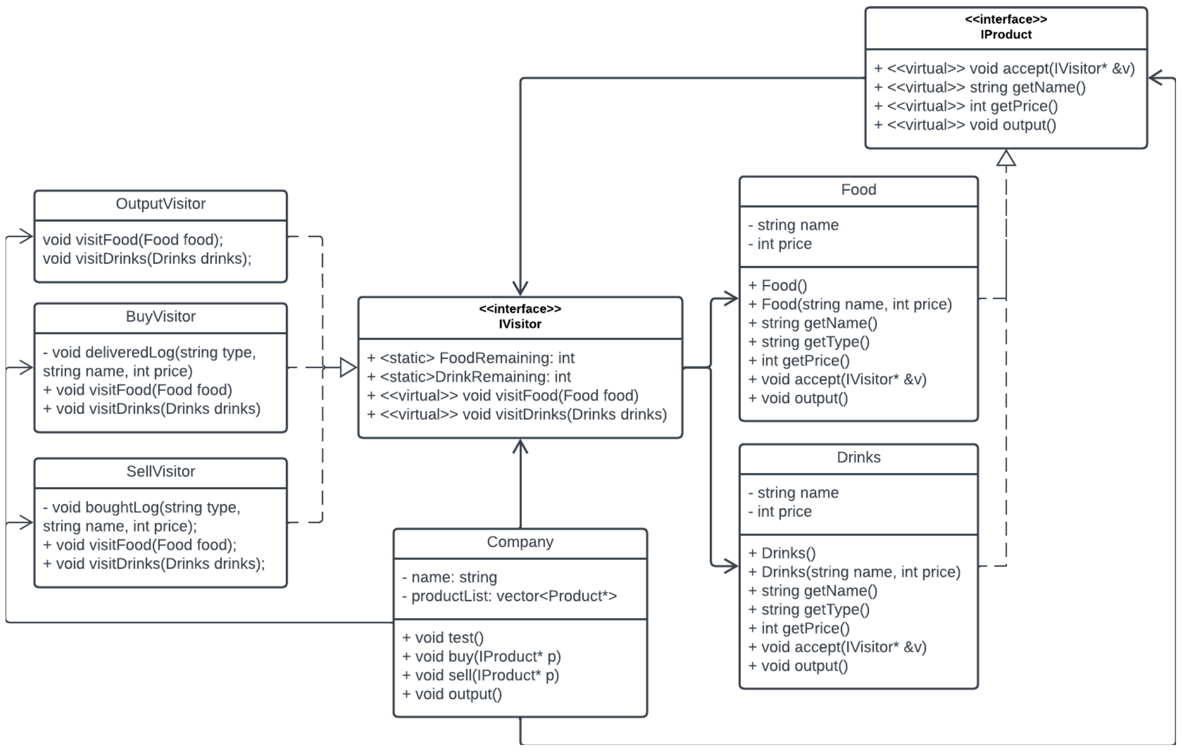
Fig 1.2: General implementation

1.6 Class diagram for the stated problem

We can see that with this implementation, Buy, Sell, Output operations can now be seperated from the Food, Drinks classes. The component classes only contains the most general methods like getter and output method. All other methods that requires the component classes can now be implemented seperately.

1.7 Implementations

Refer to folder Code/Visitor/Example1/VisitorSolution for the implementation. The problem that needs to be noted when implement this



<<interface>>
IProduct
+ <<virtual>> void accept(IVisitor* &v)
+ <<virtual>> string getName()
+ <<virtual>> int getPrice()
+ <<virtual>> void output()
OutputVisitor
void visitFood(Food food);
void visitDrinks(Drinks drinks);
BuyVisitor
- void deliveredLog(string type, string name, int price)
+ void visitFood(Food food)
+ void visitDrinks(Drinks drinks)
SellVisitor
- void boughtLog(string type, string name, int price);
+ void visitFood(Food food);
+ void visitDrinks(Drinks drinks);
<<interface>>
IVisitor
+ <static> FoodRemaining: int
+ <static>DrinkRemaining: int
+ <<virtual>> void visitFood(Food food)
+ <<virtual>> void visitDrinks(Drinks drinks)
Food
- string name
- int price
+ Food()
+ Food(string name, int price)
+ string getName()
+ string getType()
+ int getPrice()
+ void accept(IVisitor* &v)
+ void output()
Drinks
- string name
- int price
+ Drinks()
+ Drinks(string name, int price)
+ string getName()
+ string getType()
+ int getPrice()
+ void accept(IVisitor* &v)
+ void output()
Company
- name: string
- productList: vector<Product*>
+ void test()
+ void buy(IProduct* p)
+ void sell(IProduct* p)
+ void output()

Fig 1.3: Problem diagram

pattern is that you have to make foward declaration for class IVisitor in the IProduct.

1.8 Advantages and Disadvantages

1.8.1 Advantages

Visitor patter provide us with many advantages:

- New methods which can work with multiple complicated classes can be added without changing the original classes by adding a new visitor class.
- Multiple versions of the same behaviour can be added into the same class using the visitor corresponding to that mutual behaviour.

- Pattern is great for transversing through complex object structures like an object tree and perform operations via the visitor.

1.8.2 Disadvantages

- All of the visitors need updating when a new class is introduced or an existing class is removed from the hierachy.
- Private fields and methods of the elements might be inaccessible to the visitors, which is also an inconvenience.

1.9 Other real-world problems

Exporting the map/data of a city, simple code implementation at folder "Code/Visitor/Map Export". We can consider there are many different types of building, but they all implement the “Entity” interface. So, we can apply the visitor pattern here to have different “exporters” for different buildings. Moreover, when we wish to create more behaviors with the city, we just need to add the implementation in the “visitors” classes. It will be something look like this

This design pattern is good for general use of softwares where you need to process data with different classes. Good examples can be for developing graphic library with multiple types of shapes, colors, shadow,... and you need to update more feature in the future.

1.10 Quizzes

1.10.1 What problems can the Visitor pattern solve?

(a) Help to define a new operation for (some) classes of an object structure without changing the classes.

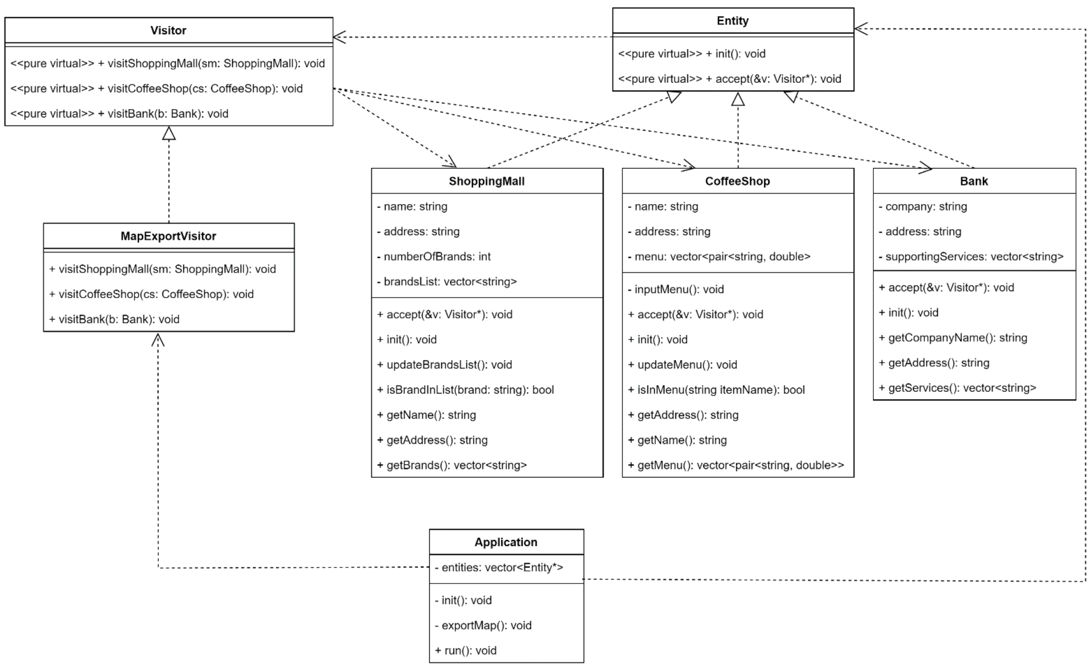

Visitor
<<pure virtual>> + visitShoppingMall(sm: ShoppingMall): void
<<pure virtual>> + visitCoffeeShop(cs: CoffeeShop): void
<<pure virtual>> + visitBank(b: Bank): void
Entity
<<pure virtual>> + init(): void
<<pure virtual>> + accept(&v: Visitor*): void
MapExportVisitor
+ visitShoppingMall(sm: ShoppingMall): void
+ visitCoffeeShop(cs: CoffeeShop): void
+ visitBank(b: Bank): void
ShoppingMall
- name: string
- address: string
- numberOfBrands: int
- brandsList: vector<string>
+ accept(&v: Visitor*): void
+ init(): void
+ updateBrandsList(): void
+ isBrandInList(brand: string): bool
+ getName(): string
+ getAddress(): string
+ getBrands(): vector<string>
CoffeeShop
- name: string
- address: string
- menu: vector<pair<string, double>
- inputMenu(): void
+ accept(&v: Visitor*): void
+ init(): void
+ updateMenu(): void
+ isInMenu(string itemName): bool
+ getAddress(): string
+ getName(): string
+ getMenu(): vector<pair<string, double>>
Bank
- company: string
- address: string
- supportingServices: vector<string>
+ accept(&v: Visitor*): void
+ init(): void
+ getCompanyName(): string
+ getAddress(): string
+ getServices(): vector<string>
Application
- entities: vector<Entity*>
- init(): void
- exportMap(): void
+ run(): void

Fig 1.4: Other application example

(b) Making changes to the changes to the classes to define new operations

(c) Help to define a new operation for (some) classes of an object structure while also changing the classes.

The answer is (a).

1.10.2 What techniques does the Visitor pattern base on?

(a) Single Dispatch

(b) Double Dispatch

(c) Triple Dispatch

The answer is (b).

1.10.3 Which is an advantage of Visitor pattern?

(a) New methods which can work with multiple complicated classes can be added without changing the original classes by adding a new visitor class.

(b) Multiple versions of the same behaviour can be added into the same class using the visitor corresponding to that mutual behaviour.

(c) Private fields and methods of the elements might be inaccessible to the visitors.

Then answer is (a) and (b)

1.10.4 Which is an disadvantage of Visitor pattern?

(a) All of the visitors need updating when a new class is introduced or an existing class is removed from the hierachy.

(b) Pattern is great for transversing through complex object structures like an object tree and perform operations via the visitor.

(c) Private fields and methods of the elements might be inaccessible to the visitors, which is also an inconvenience.

Answer is (a) and (c)

1.10.5 What method must the interface of elements have?

(a) Init method

(b) Visit method

(c) Accept method

Answer is (c)

Chapter 2

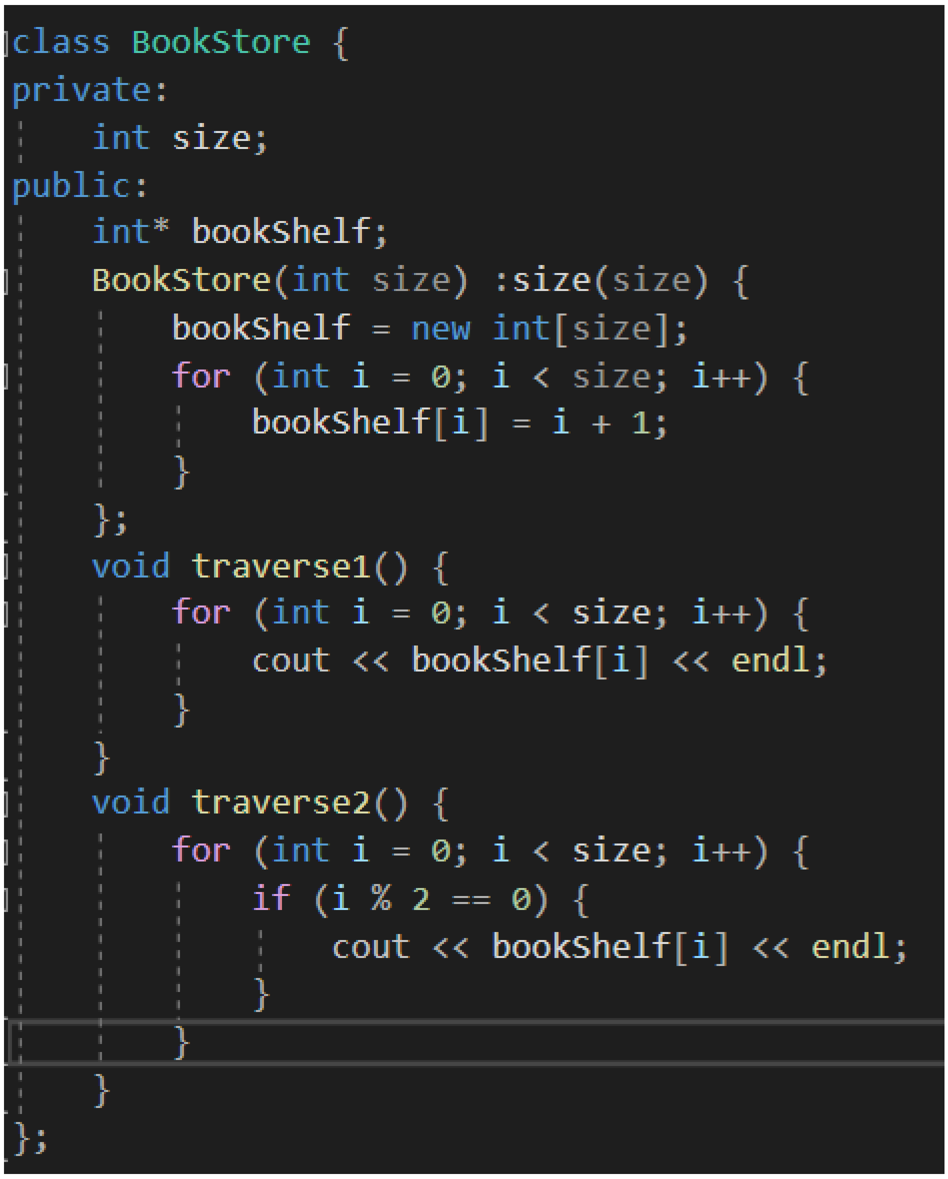
Iterator Design Pattern

2.1 Real World Problem

Assume that we need to develop an application to organize a bookstore with a lot of bookshelves. Each of the bookshelves has a specific number of books. Analogously, we also organize a chain of restaurants, each of them has a specific number of chairs, a classroom with a number of students... We would call all of the examples above "collections". Most collections store their elements in simple lists, and it should provide a way to go through each element.

2.2 Naive Solution

In order to transverse through a collection in different ways, we can define multiple methods to do this. Example: In this code, we want to display the number of books on all shelf(transverse 1) and display the number of books on even shelf (transverse 2)



```
class BookStore {
private:
    int size;
public:
    int* bookShelf;
    BookStore(int size) :size(size) {
        bookShelf = new int[size];
        for (int i = 0; i < size; i++) {
            bookShelf[i] = i + 1;
        }
    };
    void traverse1() {
        for (int i = 0; i < size; i++) {
            cout << bookShelf[i] << endl;
        }
    }
    void traverse2() {
        for (int i = 0; i < size; i++) {
            if (i % 2 == 0) {
                cout << bookShelf[i] << endl;
            }
        }
    }
};
```

Fig 2.1: Bookstore transverse

2.3 Problems with the naive solution

Based on that approach, when we want to traverse by a new way, we must add a new function to BookShelf class. At that time, we must know the underlying structure of the BookShelf. The collection gradually blurs its primary responsibility which is efficient data storage. With other entitys such as: restaurant, school, we will have a lot of traverse functions inside each of these classes. Therfore, it will contradict to the purpose of Object-

Oriented Programming.

2.4 Overview of the Iterator design pattern

C++ Iterator is an abstract notion which identifies an element of a sequence. A sequence could be anything that is connected and sequential, but not necessarily consecutive on the memory. C++ Iterator could be a raw pointer, or any data structure which satisfies the basics of a C++ Iterator. It has been widely used in C++ STL standard library, especially for the std containers and functions.

2.5 Genral class diagram

We have the class diagram for the design pattern in Figure 2.2

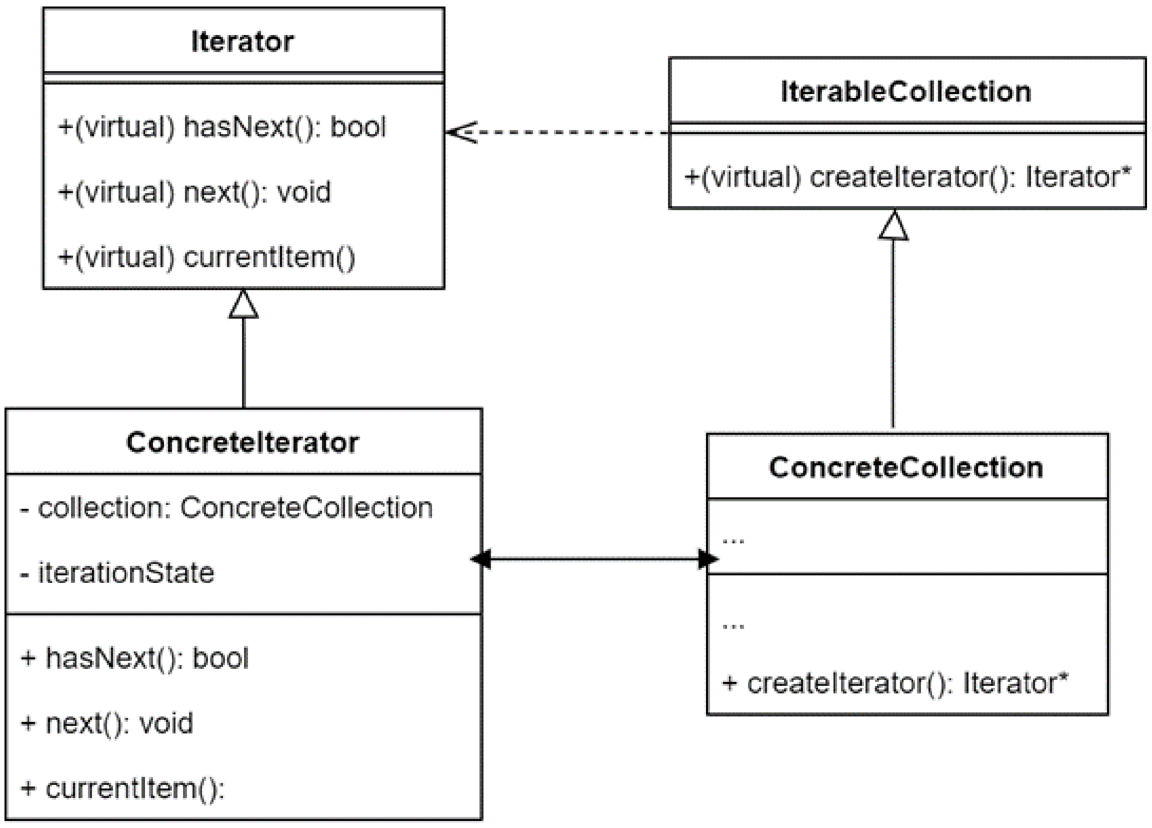


Fig 2.2: Iterator basic class diagram

2.6 Diagram for the problem

Detail diagram of the problem in Figure 2.3

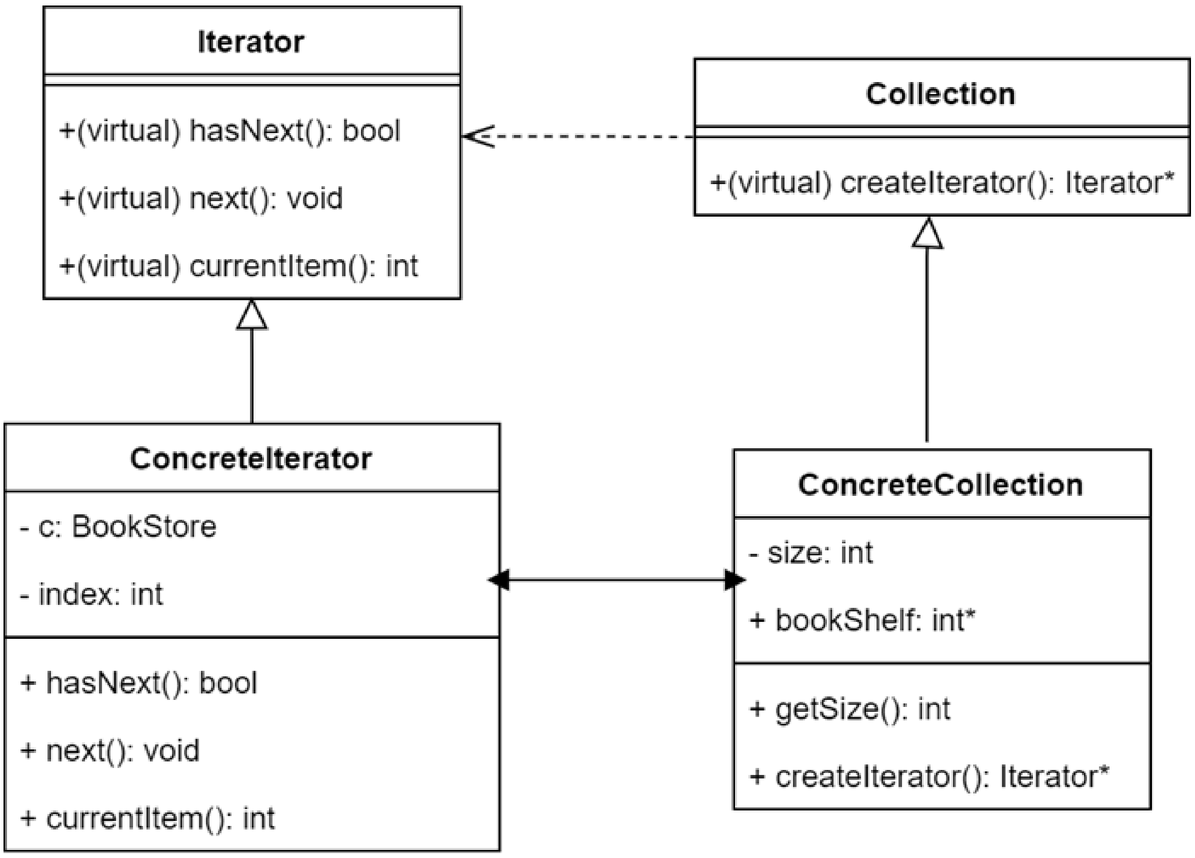


Fig 2.3: Example class diagram

2.7 Implementation of the pattern

Refer to Folder link at folder Code/Iterator/IteratorSolution.

In this implementation, we seperate the entity and the iterator, so that we don't care about the underlying of entity class. The same process can be applied for other child classes which is inherited from Collection

2.8 Advantages and Disadvantages

2.8.1 Advantages

Advantages:

- Single Responsibility Principle. You can clean up the client code and the collections by extracting bulky traversal algorithms into separate classes.
- Open/Closed Principle. You can implement new types of collections and iterators and pass them to existing code without breaking anything.
- You can iterate over the same collection in parallel because each iterator object contains its own iteration state.
- For the same reason, you can delay an iteration and continue it when needed.

Disadvantages:

- Applying the pattern can be overkill if your app only works with simple collections.
- Using an iterator may be less efficient than going through elements of some specialized collections directly.

2.9 Other real-world problems

Use the Iterator pattern when your collection has a complex data structure under the hood, but you want to hide its complexity from clients (either for convenience or security reasons).

The iterator encapsulates the details of working with a complex data structure, providing the client with several simple methods of accessing the collection elements. While this approach is very convenient for the client, it also protects the collection from careless or malicious actions which the client would be able to perform if working with the collection directly.

The code of non-trivial iteration algorithms tends to be very bulky. When placed within the business logic of an app, it may blur the responsibility of the original code and make it less maintainable. Moving the traversal code to designated iterators can help you make the code of the application leaner and cleaner.

Use the Iterator when you want your code to be able to traverse different data structures or when types of these structures are unknown beforehand. The pattern provides a couple of generic interfaces for both collections and iterators. Given that your code now uses these interfaces, it'll still work if you pass it various kinds of collections and iterators that implement these interfaces

2.10 Quizzes

2.10.1 What are Iterators?

(a) STL component used to point a memory address of a container

(b) STL component used for vectors

(c) STL component used to call functions efficiently

(d) STL component used to define template classes

The answer is (a). Iterators are STL components used to point a memory address of a container. They are used to iterate over container classes.

2.10.2 Pick the correct statement.

(a) Input iterator moves sequentially forward

(b) Input iterator moves sequentially backward

(c) Input iterator moves in both direction

(d) Input iterator moves sequentially downwards

The answer is (a). By definition Input iterators moves sequentially forward.

2.10.3 Which of the following is correct about Input Iterators?

(a) Input iterators can be used with all relational operators

(b) Input iterators can work with arithmetic operators

(c) No value can be assigned to the location pointed by Input Iterator

(d) Input iterators can work with sequence operators

The answer is (c). Values cannot be assigned to the location pointed by input operators. Input operators cannot be used with all relational and arithmetic operators.

2.10.4 Which of the following is correct about Input Iterators?

(a) They cannot be decremented

(b) Cannot be used in multi-pass algorithms

(c) Can only be incremented

(d) All of the mentioned

The answer is (d). Input iterators can only be incremented and as it cannot be decremented it can be used in single-pass algorithms only

2.10.5 Which of the following is correct?

(a) Input Iterators are used for assigning

(b) Output Iterators are used for assigning

(c) Both Input and Output Iterators are used for accessing

(d) Both Input and Output Iterators are used for assigning

The answer is (b). Input Iterators are used for accessing and Output Iterators are used for assigning.

References

[1] https://refactoring.guru/design-patterns/book