

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH



BÀI TẬP LỚN MÔN LẬP TRÌNH NÂNG CAO
ĐỀ TÀI:
FUNCTIONAL PROGRAMMING IN PYTHON
& DESIGN PATTERN
LỚP TN-01 _ HK 232
NGÀY NỘP: 02/06/2024

Giảng viên hướng dẫn: THẦY TRƯỞNG TUẤN ANH

Sinh viên thực hiện	Mã số sinh viên	Điểm số
ĐẶNG HOÀNG KHANG	2211422	

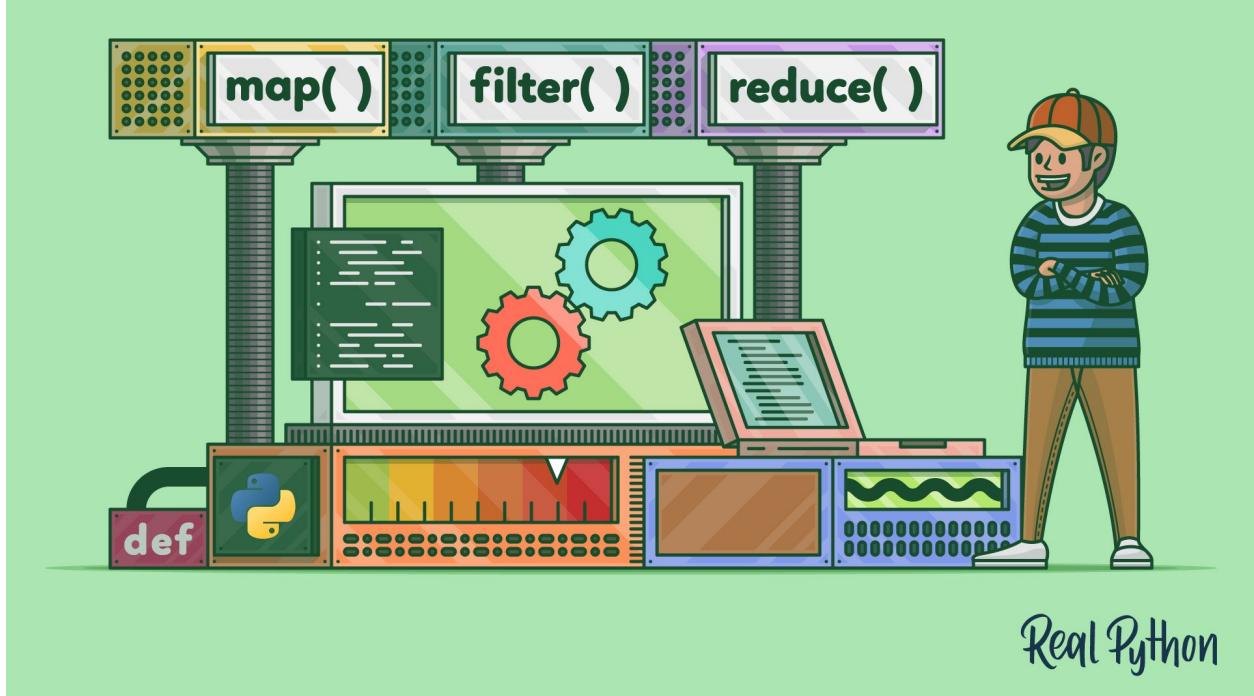
Thành phố Hồ Chí Minh – 2024

Table of Contents

LẬP TRÌNH HÀM TRONG PYTHON	4
1.Lập trình hàm trong Python	4
2.Python hỗ trợ lập trình chức năng tốt như thế nào?	4
3.Tạo một hàm.....	5
4.Gọi một hàm	5
5.Arguments	5
6.Parameters and Arguments	5
7.Number of Arguments	6
8.Arbitrary Arguments, *args	6
9.Keyword Arguments.....	7
10.Arbitrary Keyword Arguments, **kwargs	7
11.Default Parameter Value.....	7
12.Passing a List as an Argument.....	7
13.Return Values.....	8
14.The pass Statement	8
15.Positional-Only Arguments.....	8
16.Combine Positional-Only and Keyword-Only	9
17.Recursion	9
18.Python Lambda.....	9
19.Calling function again.....	11
20.High-order function	12
21.Map function	12
22.Filter function.....	14
23.Reduce function.....	15
24.Kết luận.....	17
DESIGN PATTERN.....	18
1.Định nghĩa.....	18
2.Lợi ích	18
3.Phân loại	19
4. Creational pattern – Factory Method	19

5. Creational pattern – Singleton.....	22
6. Creational pattern – Builder	25
7. Creational pattern – Abstract factory.....	30
8. Creational pattern – Prototype.....	34
9. Behavioral pattern – Chain of Responsibility	37
10. Behavioral pattern – Command	42
11. Behavioral pattern – Iterator.....	46
12. Behavioral pattern – Mediator	49
13. Behavioral pattern – Memento	53
14. Behavioral pattern – Observer	57
15. Behavioral pattern – State.....	59
16. Behavioral pattern – Strategy.....	62
17. Behavioral pattern – Template Method.....	65
18. Behavioral pattern – Visitor	68
19. Behavioral pattern – Interpreter	71
20. Structural pattern – Adapter	77
21. Structural pattern – Bridge.....	82
22. Structural pattern – Composite	85
23. Structural pattern – Facade	88
24. Structural pattern – Flyweight	91
25. Structural pattern – Proxy	94
26. Structural pattern – Decorator	97

LẬP TRÌNH HÀM TRONG PYTHON



Real Python

1. Lập trình hàm trong Python

Hàm **thuần túy** trong Python là hàm có giá trị đầu ra chỉ tuân theo các giá trị đầu vào của nó mà không có bất kỳ tác dụng phụ nào có thể quan sát được. Trong **lập trình hàm**, một chương trình hoàn toàn bao gồm việc đánh giá các hàm thuần túy. Quá trình tính toán được tiến hành bằng cách lệnh gọi hàm lồng nhau hoặc tổng hợp mà không thay đổi trạng thái hoặc dữ liệu có thể thay đổi.

Mô hình functional rất phổ biến vì nó mang lại một số lợi thế so với các mô hình lập trình khác. Functional code bao gồm:

- **High - level:** mô tả kết quả mình mong muốn thay vì chỉ định rõ ràng các bước cần thiết để đạt được kết quả đó. Những câu lệnh đơn lẻ nhưng kết hợp với nhau tạo thành một chương trình.
- **Transparent:** Hành vi của hàm thuần túy chỉ phụ thuộc vào đầu vào và đầu ra của nó, không có giá trị trung gian. Điều đó giúp loại bỏ khả năng xảy ra tác dụng phụ, tạo điều kiện thuận lợi cho việc chửa lỗi code.
- **Parallelizable:** Các routines không gây ra deadlock có thể dễ dàng chạy song song với nhau hơn.

Nhiều ngôn ngữ lập trình hỗ trợ một số mức độ lập trình chức năng. Trong một số ngôn ngữ, hầu như tất cả mã đều tuân theo mô hình chức năng. Haskell là một ví dụ như vậy. Ngược lại, Python không hỗ trợ lập trình chức năng nhưng cũng chứa các tính năng của các mô hình lập trình khác.

2. Python hỗ trợ lập trình chức năng tốt như thế nào?

Để hỗ trợ lập trình hàm, sẽ rất hữu ích nếu một hàm trong ngôn ngữ lập trình có hai khả năng sau: có thể nhận một hàm khác làm đối số và có thể trả về một hàm khác cho người gọi

Python hỗ trợ tốt cả hai khả năng này. Mọi thứ trong chương trình Python đều là đối tượng. Tất cả các đối tượng trong Python đều có vị trí tương đương nhau và hàm cũng không phải là ngoại lệ.

Trong Python, các hàm là những đối tượng hạng nhất. Điều đó có nghĩa là các hàm có cùng các đặc điểm như các giá trị như chuỗi và số. Bất kỳ điều gì ta mong đợi có thể làm với một chuỗi hoặc số, ta cũng có thể làm với một hàm.

3.Tạo một hàm

Trong Python, một hàm được định nghĩa bằng từ khóa def :

4.Gọi một hàm

Để gọi một hàm, hãy sử dụng tên hàm theo sau dấu ngoặc đơn:

```
def my_function():
    print("Hello from a function")  
my_function()
```

Hello from a function

5.Arguments

Thông tin có thể được chuyển vào các hàm dưới dạng đối số.

Các đối số được chỉ định sau tên hàm, bên trong dấu ngoặc đơn. Bạn có thể thêm bao nhiêu đối số tùy thích, chỉ cần phân tách chúng bằng dấu phẩy.

Ví dụ sau đây có một hàm có một đối số (fname). Khi hàm được gọi, chúng ta chuyển vào tên, tên này được sử dụng bên trong hàm để in tên đầy đủ:

```
def my_function(fname):
    print(fname + " Chonce")  
  
my_function("Emo")
my_function("Pentakill")
my_function("K/DA")
```

Emo Chonce
Pentakill Chonce
K/DA Chonce

6.Parameters and Arguments

Tham số (parameter): Là biến được liệt kê bên trong dấu ngoặc đơn trong định nghĩa hàm. Tham số là tên mà chúng ta đặt để đại diện cho dữ liệu sẽ được truyền vào hàm.

Đối số (argument): Là giá trị cụ thể được truyền vào hàm khi nó được gọi. Đối số là dữ liệu thực tế mà chúng ta truyền vào các tham số của hàm.

Tổng quát, tham số là biến được định nghĩa trong hàm để nhận giá trị, còn đối số là giá trị thực tế được truyền vào hàm khi gọi. Chung quy lại, chúng được sử dụng cho cùng một mục đích: thông tin được truyền vào hàm.

7. Number of Arguments

Theo mặc định, một hàm phải được gọi với số lượng đối số chính xác. Có nghĩa là nếu hàm của bạn cần có 2 đối số, phải gọi hàm với 2 đối số, không nhiều hơn cũng không ít hơn.

Nếu bạn cố gắng gọi hàm với 1 hoặc 3 đối số, sẽ gặp lỗi.

```
def my_function(fname, lname):
    print(fname + " " + lname)
```

```
my_function("Khang", "Dang")  Khang Dang
```

```
def my_function(fname, lname):
    print(fname + " " + lname)

my_function("Emil")
```

```
Traceback (most recent call last):
  File "demo_function_args_error.py", line 4, in <module>
    my_function("Emil")
TypeError: my_function() missing 1 required positional argument: 'lname'
```

8. Arbitrary Arguments, *args

Nếu ta không biết có bao nhiêu đối số sẽ được truyền vào hàm của mình, hãy thêm a *trước tên tham số trong định nghĩa hàm.

Bằng cách này, hàm sẽ nhận được một bộ đối số và có thể truy cập các mục tương ứng:

```
def my_function(*kids):
    print("The youngest child is " + kids[2])
my_function("Emil", "Tobias", "Linus")
```

```
The youngest child is Linus
```

9.Keyword Arguments

Ta cũng có thể gửi đối số bằng cú pháp *key = value*. Bằng cách này, thứ tự của các đối số không quan trọng.

```
def my_function(child3, child2, child1):
    print("The youngest child is " + child3)
my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

```
The youngest child is Linus
```

10.Arbitrary Keyword Arguments, **kwargs

Nếu ta không biết có bao nhiêu đối số từ khóa sẽ được truyền vào hàm của mình, hãy thêm hai dấu hoa thị: ****trước tên tham số trong định nghĩa hàm.

Bằng cách này, hàm sẽ nhận được một *từ điển* các đối số và có thể truy cập các mục tương ứng:

```
def my_function(**kid):
    print("His last name is " + kid["lname"])
my_function(fname = "Tobias", lname = "Refsnes") His last name is Refsnes
```

11.Default Parameter Value

Ví dụ sau đây cho thấy cách sử dụng giá trị tham số mặc định.

Nếu chúng ta gọi hàm mà không có đối số, nó sẽ sử dụng giá trị mặc định:

```
def my_function(country = "Norway"):
    print("I am from " + country)
my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

12.Passing a List as an Argument

Ta có thể gửi bất kỳ loại đối số dữ liệu nào tới một hàm (chuỗi, số, danh sách, từ điển, v.v.) và nó sẽ được coi là cùng một loại dữ liệu bên trong hàm.

Ví dụ: nếu ta gửi Danh sách làm đối số, nó vẫn sẽ là Danh sách khi đến hàm:

```
def my_function(food):
    for x in food:
        print(x)

fruits = ["apple", "banana", "cherry"]
my_function(fruits)
```

apple
banana
cherry

13.Return Values

Để cho phép hàm trả về một giá trị, hãy sử dụng câu return lệnh:

```
def my_function(x):
    return 5 * x

print(my_function(3)) 15
print(my_function(5)) 25
print(my_function(9)) 45
```

14.The pass Statement

Một function định nghĩa không được để trống, nhưng nếu vì lý do nào đó ta có function định nghĩa không có nội dung thì hãy đưa vào pass câu lệnh để tránh bị lỗi.

```
def myfunction():
    pass
```

15.Positional-Only Arguments

Ta có thể chỉ định rằng một hàm chỉ có thể có đối số vị trí hoặc chỉ đối số từ khóa.

Để chỉ định rằng một hàm chỉ có thể có các đối số vị trí, ta thêm vào , / sau các đối số:

```
def my_function(x, /):
    print(x)

my_function(3)
```

3

Nếu không có , / ta thực sự được phép sử dụng các đối số từ khóa ngay cả khi hàm mong đợi các đối số vị trí:

```
def my_function(x):
    print(x)

my_function(x = 3)
```

3

Nhưng khi thêm , / ta sẽ gặp lỗi nếu có gửi đối số từ khóa:

```

def my_function(x, /):
    print(x)

my_function(x = 3)
Traceback (most recent call last):
  File "./prog.py", line 4, in <module>
TypeError: my_function() got some positional-only arguments passed as keyword arguments: 'x'

```

16. Combine Positional-Only and Keyword-Only

Ta có thể kết hợp hai loại đối số trong cùng một hàm.

Bất kỳ đối số nào *trước* chỉ / , là vị trí và bất kỳ đối số nào sau *chi**, là từ khóa.

```

def my_function(a, b, /, *, c, d):
    print(a + b + c + d)

```

my_function(5, 6, c = 7, d = 8)	26
---------------------------------	----

17. Recursion

Python cũng chấp nhận đệ quy hàm, có nghĩa là một hàm được xác định có thể gọi chính nó.

Đệ quy là một khái niệm toán học và lập trình phổ biến. Nó có nghĩa là một hàm sẽ gọi chính nó. Điều này có lợi là bạn có thể lặp qua dữ liệu để đạt được kết quả.

Phết súc cẩn thận với đệ quy vì có thể khá dễ dàng viết một hàm không bao giờ kết thúc hoặc một hàm sử dụng quá nhiều bộ nhớ hoặc công sức bộ xử lý. Tuy nhiên, khi được viết chính xác, đệ quy có thể là một cách tiếp cận lập trình rất hiệu quả và tinh tế về mặt toán học.

Trong ví dụ này, tri_recursion() là một hàm mà chúng ta đã xác định để gọi chính nó ("recurse"). Chúng tôi sử dụng biến k làm dữ liệu, biến này sẽ giảm (-1) mỗi lần chúng ta lặp lại. Quá trình đệ quy kết thúc khi điều kiện không lớn hơn 0 (tức là khi nó bằng 0).

```

def tri_recursion(k):
    if(k > 0):
        result = k + tri_recursion(k - 1)
        print(result)
    else:
        result = 0
    return result

print("\n\nRecursion Example Results")
tri_recursion(6)

```

Recursion Example Results
1
3
6
10
15
21

18. Python Lambda

Hàm lambda là một hàm ẩn danh nhỏ.

Hàm lambda có thể nhận bất kỳ số lượng đối số nào nhưng chỉ có thể có một biểu thức.

Cú pháp

`lambda arguments : expression`

Biểu thức được thực thi và kết quả được trả về:

```
x = lambda a: a + 10  
print(x(5))          15
```

Hàm Lambda có thể nhận bất kỳ số lượng đối số nào:

```
x = lambda a, b: a * b  
print(x(5, 6))        30
```

```
x = lambda a, b, c: a + b + c  
print(x(5, 6, 2))    13
```

Tại sao nên sử dụng hàm Lambda?

Lợi ích của lambda được thể hiện rõ hơn khi ta sử dụng chúng như một hàm ẩn danh bên trong một hàm khác.

Giả sử ta có một định nghĩa hàm nhận vào một đối số và đối số đó sẽ được nhân với một số chưa xác định:

```
def myfunc(n):  
    return lambda a : a * n
```

Sử dụng định nghĩa hàm đó để tạo một hàm luôn nhân đối số mà chúng ta gửi vào:

```
def myfunc(n):  
    return lambda a : a * n  
  
mydoubler = myfunc(2)  
print(mydoubler(11))      22
```

Hoặc, sử dụng định nghĩa hàm tương tự để tạo một hàm luôn *nhân ba* số chúng ta gửi vào:

```
def myfunc(n):
    return lambda a : a * n

mytripler = myfunc(3)

print(mytripler(11))
```

33

Hoặc, sử dụng cùng một định nghĩa hàm để tạo cả hai hàm trong cùng một chương trình:

```
def myfunc(n):
    return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

22
33

19. Calling function again

Khi ta chuyển một hàm sang một hàm khác, hàm được truyền vào đôi khi được gọi là lệnh gọi lại vì lệnh gọi lại hàm bên trong có thể sửa đổi hành vi của hàm bên ngoài.

Một ví dụ điển hình về điều này là hàm Python sorted(). Thông thường, nếu ta chuyển danh sách các giá trị chuỗi tới sorted(), thì nó sẽ sắp xếp chúng theo thứ tự từ vựng:

```
animals = ["ferret", "vole", "dog", "gecko"]
sorted(animals)

['dog', 'ferret', 'gecko', 'vole']
```

Tuy nhiên, sorted() lấy một đối key số tùy chọn chỉ định hàm gọi lại có thể đóng vai trò là khóa sắp xếp. Vì vậy, ví dụ: ta có thể sắp xếp theo độ dài chuỗi:

```
animals = ["ferret", "vole", "dog", "gecko"]
sorted(animals, key=len)

['dog', 'vole', 'gecko', 'ferret']
```

`sorted()` cũng có thể lấy một đối số tùy chọn chỉ định việc sắp xếp theo thứ tự ngược lại. Nhưng bạn có thể quản lý điều tương tự bằng cách xác định hàm gọi lại của riêng bạn để đảo ngược ý nghĩa của `len()`:

```
animals = ["ferret", "vole", "dog", "gecko"]
sorted(animals, key=len, reverse=True)

def reverse_len(s):
    return -len(s)

sorted(animals, key=reverse_len)
['ferret', 'gecko', 'vole', 'dog']
```

20.High-order function

Ta có thể truyền một hàm cho một hàm khác làm đối số, một hàm cũng có thể chỉ định một hàm khác làm giá trị trả về của nó:

Ví dụ 1

```
def apply_function(func, value):
    return func(value)

def square(x):
    return x * x

result = apply_function(square, 5)
print(result) # Output: 25
```

Ví dụ 2

```
def create_multiplier(factor):
    def multiplier(x):
        return x * factor
    return multiplier

double = create_multiplier(2)
triple = create_multiplier(3)

print(double(5)) # Output: 10
print(triple(5)) # Output: 15
```

21.Map function

Hàm map trong Python là một hàm bậc cao được sử dụng để áp dụng một hàm cho từng phần tử trong một hoặc nhiều iterable (như danh sách, bộ, v.v.) và trả về một iterator với các kết quả.

Cú pháp cơ bản của map là:

map(function, iterable, ...)

Trong đó:

function: Là hàm sẽ được áp dụng cho mỗi phần tử của iterable.

iterable, ...: Một hoặc nhiều iterable.

Áp dụng một hàm cho một danh sách

```
def square(x):
    return x * x

numbers = [1, 2, 3, 4, 5]
squared_numbers = map(square, numbers)
print(list(squared_numbers)) # Output: [1, 4, 9, 16, 25]
```

Trong ví dụ này, hàm square được áp dụng cho từng phần tử của danh sách numbers, và map trả về một iterator với các kết quả.

Sử dụng lambda với map

```
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x * x, numbers)
print(list(squared_numbers)) # Output: [1, 4, 9, 16, 25]
```

Ở đây, thay vì định nghĩa một hàm riêng biệt, ta sử dụng một hàm lambda để tính bình phương của từng phần tử trong danh sách.

Áp dụng một hàm cho nhiều iterable

```
def add(x, y):
    return x + y

numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]
summed_numbers = map(add, numbers1, numbers2)
print(list(summed_numbers)) # Output: [5, 7, 9]
```

Trong ví dụ này, hàm add được áp dụng cho từng cặp phần tử từ hai danh sách numbers1 và numbers2.

Sử dụng map với các kiểu iterable khác nhau

```

def to_upper(s):
    return s.upper()

strings = ('hello', 'world', 'python')
upper_strings = map(to_upper, strings)
print(list(upper_strings)) # Output: ['HELLO', 'WORLD', 'PYTHON']

```

Ở đây, hàm `to_upper` được áp dụng cho từng phần tử trong một tuple các chuỗi.

→ `map` trả về một iterator, vì vậy để xem kết quả, cần chuyển đổi nó thành một danh sách hoặc một kiểu iterable khác bằng cách sử dụng `list()`, `tuple()`, v.v.

→ Hàm `map` không thay đổi các iterable gốc, mà tạo ra một iterator mới với các kết quả.

Tổng quát, hàm `map` rất hữu ích khi bạn muốn áp dụng một hàm nào đó cho toàn bộ phần tử trong một hoặc nhiều iterable một cách ngắn gọn và hiệu quả.

22. Filter function

Hàm `filter` trong Python là một hàm bậc cao được sử dụng để lọc các phần tử của một iterable (như danh sách, tuple, v.v.) bằng cách sử dụng một hàm điều kiện. Hàm này sẽ trả về một iterator chứa các phần tử của iterable gốc mà hàm điều kiện trả về `True`.

Cú pháp cơ bản

`filter(function, iterable)`

Trong đó:

`function`: Là hàm điều kiện sẽ được áp dụng cho từng phần tử của iterable. Hàm này phải trả về một giá trị boolean (`True` hoặc `False`).

`iterable`: Một iterable (như danh sách, tuple, v.v.) mà các phần tử của nó sẽ được lọc.

Ví dụ sử dụng hàm `filter`

Lọc các số chẵn từ một danh sách

```

def is_even(n):
    return n % 2 == 0

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = filter(is_even, numbers)
print(list(even_numbers)) # Output: [2, 4, 6, 8, 10]

```

Trong ví dụ này, hàm `is_even` được sử dụng để lọc các số chẵn từ danh sách `numbers`.

Sử dụng lambda với filter

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd_numbers = filter(lambda n: n % 2 != 0, numbers)
print(list(odd_numbers)) # Output: [1, 3, 5, 7, 9]
```

Ở đây, thay vì định nghĩa một hàm riêng biệt, chúng ta sử dụng một hàm lambda để lọc các số lẻ từ danh sách `numbers`.

Lọc các chuỗi có độ dài lớn hơn 3

```
def is_long_string(s):
    return len(s) > 3

strings = ["hi", "hello", "how", "are", "you"]
long_strings = filter(is_long_string, strings)
print(list(long_strings)) # Output: ['hello']
```

Hàm `is_long_string` được sử dụng để lọc các chuỗi có độ dài lớn hơn 3 từ danh sách `strings`.

Lọc các phần tử không phải là số None

```
data = [1, None, 2, None, 3, 4, None, 5]
non_none_data = filter(lambda x: x is not None, data)
print(list(non_none_data)) # Output: [1, 2, 3, 4, 5]
```

Trong ví dụ này, chúng ta sử dụng một hàm lambda để lọc các phần tử không phải là `None` từ danh sách `data`.

→ `filter` trả về một iterator, vì vậy để xem kết quả, cần chuyển đổi nó thành một danh sách hoặc một kiểu iterable khác bằng cách sử dụng `list()`, `tuple()`, v.v.

→ Hàm `filter` không thay đổi các iterable gốc, mà tạo ra một iterator mới chứa các phần tử được lọc.

Tổng quát, hàm `filter` rất hữu ích khi muốn lọc các phần tử từ một iterable dựa trên một điều kiện nào đó một cách ngắn gọn và hiệu quả.

23. Reduce function

Hàm reduce trong Python là một hàm bậc cao từ module functools được sử dụng để áp dụng một hàm lên từng cặp phần tử của một iterable (như danh sách, tuple, v.v.) và kết hợp chúng lại thành một giá trị duy nhất.

Cú pháp cơ bản

```
functools.reduce(function, iterable[, initializer])
```

Trong đó:

function: Là hàm sẽ được áp dụng cho từng cặp phần tử của iterable. Hàm này phải nhận hai đối số và trả về một giá trị.

iterable: Một iterable (như danh sách, tuple, v.v.) mà các phần tử của nó sẽ được kết hợp.

initializer (tùy chọn): Một giá trị khởi đầu, nếu được cung cấp thì sẽ được sử dụng làm giá trị khởi đầu cho quá trình kết hợp. Nếu không có giá trị khởi đầu, phần tử đầu tiên của iterable sẽ được sử dụng làm giá trị khởi đầu, và quá trình kết hợp sẽ bắt đầu từ phần tử thứ hai của iterable.

Ví dụ sử dụng hàm reduce

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
sum_of_numbers = reduce(lambda x, y: x + y, numbers)
print(sum_of_numbers) # Output: 15

numbers = [1, 2, 3, 4, 5]
product_of_numbers = reduce(lambda x, y: x * y, numbers, 1)
print(product_of_numbers) # Output: 120
```

→ Hàm reduce trả về một giá trị duy nhất, không phải một iterator như map và filter.

→ Nếu iterable trống và không có giá trị khởi đầu được cung cấp, reduce sẽ gây ra một TypeError.

→ Hàm reduce nên được sử dụng khi muốn kết hợp các phần tử của một iterable thành một giá trị duy nhất một cách ngắn gọn và hiệu quả. Tuy nhiên, do độ phức tạp của việc sử dụng reduce, nên lưu ý rằng nó không luôn là lựa chọn tốt nhất, và trong nhiều trường hợp, việc sử dụng các phương pháp khác như vòng lặp for có thể dễ dàng hiểu hơn.

$$\begin{array}{r}
 \begin{matrix} 1 + 2 \\ f(1, 2) = 3 \end{matrix} \\
 \downarrow \quad \quad \quad \begin{matrix} 3 + 3 \\ f(3, 3) = 6 \end{matrix} \\
 \downarrow \quad \quad \quad \begin{matrix} 6 + 4 \\ f(6, 4) = 10 \end{matrix} \\
 \downarrow \quad \quad \quad \begin{matrix} 10 + 5 \\ f(10, 5) = 15 \end{matrix}
 \end{array}$$

24.Kết luận

Lập trình hàm là một mô hình lập trình trong đó phương pháp tính toán chính là đánh giá các hàm thuần túy. Mặc dù Python về cơ bản không phải là một ngôn ngữ chức năng, nhưng nên làm quen với lambda, map(), filter() và reduce() và các hàm chức năng khác được python hỗ trợ vì chúng có thể giúp ta viết mã ngắn gọn, cấp cao, có thể song song hóa.

DESIGN PATTERN

1. Định nghĩa

Design pattern là một mô hình được sử dụng để giải quyết các vấn đề phổ biến trong thiết kế phần mềm. Chúng không phải là một giải pháp cụ thể hoàn chỉnh cho một vấn đề cụ thể, mà thay vào đó là các mẫu thiết kế đã được kiểm chứng và sử dụng rộng rãi trong lĩnh vực phát triển phần mềm. Mục tiêu của việc sử dụng design pattern là tạo ra các thiết kế dễ hiểu, dễ bảo trì và dễ mở rộng.

Design pattern cung cấp một cách tiếp cận chuẩn hóa cho việc giải quyết các vấn đề phức tạp trong thiết kế phần mềm. Ví dụ về các design pattern phổ biến bao gồm Singleton, Factory Method, Observer, Strategy, và MVC (Model-View-Controller).

2. Lợi ích

Tính tái sử dụng

Design pattern đã được kiểm chứng và thử nghiệm trong các tình huống thực tế, giúp tăng khả năng tái sử dụng mã nguồn và giảm thiểu việc viết mã mới.

Tính linh hoạt và mở rộng

Các design pattern thường được thiết kế để làm cho mã nguồn dễ dàng mở rộng và thay đổi, giúp phát triển phần mềm linh hoạt hơn khi cần phải thay đổi yêu cầu hoặc mở rộng tính năng.

Tính bảo trì

Bằng cách áp dụng các design pattern, mã nguồn thường trở nên dễ hiểu hơn và dễ bảo trì hơn. Các pattern thường tạo ra các cấu trúc rõ ràng và tổ chức mã nguồn một cách logic.

Tăng hiệu suất phát triển

Sử dụng design pattern giúp giảm thời gian cần để thiết kế và phát triển phần mềm, vì chúng cung cấp các mẫu thiết kế đã được kiểm chứng và sẵn có để sử dụng.

Tiêu chuẩn hóa

Bằng cách sử dụng design pattern, các nhà phát triển có thể nắm bắt các phương pháp thiết kế chung và tiêu chuẩn hóa hơn, giúp dễ dàng truyền đạt và hiểu các phần của mã nguồn cho các thành viên trong nhóm phát triển.

Tăng tính di động và tái sử dụng mã nguồn

Các design pattern thường tạo ra các cấu trúc linh hoạt và dễ dàng tái sử dụng, giúp cho mã nguồn có thể di động và tái sử dụng trong các dự án khác nhau.

3. Phân loại

Design pattern thường được phân loại vào ba loại chính: Creational (khởi tạo), Structural (cấu trúc), và Behavioral (hành vi). Mỗi loại pattern có một mục đích và ứng dụng riêng, giúp giải quyết các vấn đề thiết kế phần mềm từ các góc độ khác nhau. Bằng cách sử dụng và kết hợp các pattern khác nhau, nhà phát triển có thể tạo ra các hệ thống phần mềm linh hoạt, dễ bảo trì và dễ mở rộng.

Creational Patterns (Mẫu Khởi Tạo)

Creational patterns tập trung vào cách tạo ra đối tượng một cách linh hoạt, tiết kiệm và phù hợp.

Các mẫu trong nhóm này bao gồm Singleton, Factory Method, Abstract Factory, Builder, Prototype, và Object Pool.

Behavioral Patterns (Mẫu Hành Vi)

Behavioral patterns tập trung vào cách các đối tượng tương tác với nhau và phân phối trách nhiệm giữa chúng.

Các mẫu trong nhóm này bao gồm Observer, Strategy, Command, Iterator, State, Template Method, Mediator, Memento, và Visitor.

Structural Patterns (Mẫu Cấu Trúc)

Structural patterns liên quan đến cách sắp xếp các đối tượng và lớp trong một cấu trúc lớn hơn.

Các mẫu trong nhóm này bao gồm Adapter, Bridge, Composite, Decorator, Facade, Flyweight, và Proxy.

4. Creational pattern – Factory Method

Vấn đề

Khi chúng ta tạo ra các đối tượng mà không cần phải chỉ định cụ thể lớp cụ thể của đối tượng được tạo ra. Ví dụ, chúng ta cần tạo ra một hệ thống sở thú để quản lý các con vật.



Hướng giải quyết

Khi này chúng ta sẽ sử dụng design pattern factory method để giải quyết vấn đề trên. Factory Method cho phép tạo ra đối tượng mà không cần chỉ định rõ lớp cụ thể của chúng.

Các thành phần của design pattern này trong c++ như sau:

Product interface hoặc lớp cơ sở cho các đối tượng mà Factory Method tạo ra. Tất cả các sản phẩm phải triển khai chung một interface hoặc kế thừa từ cùng một lớp cơ sở.

ConcreteProduct là các lớp cụ thể mà Factory Method tạo ra. Mỗi ConcreteProduct tương ứng với một loại đối tượng cụ thể.

Creator là abstract class hoặc interface chứa phương thức Factory Method, là phương thức để tạo ra các đối tượng. Factory Method có thể là một phương thức abstract hoặc được triển khai sẵn với một cài đặt mặc định. Trong một số trường hợp, Creator có thể chứa các phương thức trợ giúp hoặc các phương thức khác để làm việc với các sản phẩm.

ConcreteCreator là các lớp cụ thể thực hiện Factory Method để tạo ra các đối tượng cụ thể. Mỗi ConcreteCreator tạo ra một loại cụ thể của sản phẩm bằng cách triển khai Factory Method.

Hiện thực code

```
#include <iostream>
#include <vector>
using namespace std;
class animal{
public:
    virtual void callname()=0;
};
class dog:public animal {
public:
    void callname(){
        cout<<"It's Dog"=<<endl;
    }
};
class cat:public animal {
public:
    void callname(){
        cout<<"It's Cat"=<<endl;
    }
};
class butterfly:public animal {
public:
    void callname(){
        cout<<"It's Butterfly"=<<endl;
    }
};
enum type{Dog,Cat,Butterfly};
class factory{
public:
    animal* callit(type t){
        if(t==Dog) return new dog();
        else if(t==Cat) return new cat();
        else return new butterfly();
    }
};
```

```
int main(){
vector<animal*> a;
factory f;
a.push_back(f.callit(Dog));
a.push_back(f.callit(Cat));
a.push_back(f.callit(Butterfly));
for(animal*& s:a){
    s->callname();
}
for(animal*& s:a){
    delete s;
}
a.clear();
return 0;
}
```

- Các lớp animal, dog, cat, butterfly:

animal: là lớp cơ sở, định nghĩa một phương thức thuần ảo callname().

dog, cat, butterfly: là các lớp dẫn xuất từ animal, mỗi lớp đều triển khai phương thức callname() để hiển thị tên của loài động vật tương ứng.

- Lớp factory:

Cung cấp phương thức callit() để tạo ra đối tượng tương ứng với loại động vật được chỉ định bằng enum type.

Phương thức này sẽ kiểm tra giá trị của type và trả về một con trỏ đến đối tượng động vật tương ứng.

- *Hàm main():*

Tạo một vector a chứa con trỏ đến các đối tượng animal.

Khởi tạo một đối tượng của lớp factory.

Sử dụng factory để tạo các đối tượng dog, cat, butterfly và thêm chúng vào vector a.

Duyệt qua vector a để gọi phương thức callname() của mỗi đối tượng.

Lợi ích

- Tính linh hoạt cao: Factory Method cho phép tạo ra các đối tượng mà không cần chỉ định rõ lớp cụ thể của chúng, giúp linh hoạt trong việc quản lý và mở rộng mã nguồn.
- Tính tái sử dụng: Bằng cách sử dụng Factory Method, bạn có thể dễ dàng tạo ra các đối tượng mới mà không cần phải sửa đổi mã nguồn hiện có.
- Tách biệt giao diện và triển khai: Factory Method giúp tách biệt phần giao diện (interface) của các đối tượng và phần triển khai (implementation) của chúng, giúp dễ dàng thay đổi hoặc mở rộng các loại đối tượng mà không ảnh hưởng đến các thành phần khác.

Hạn chế

- Tăng độ phức tạp của mã nguồn: Factory Method có thể tạo ra nhiều lớp Factory con, dẫn đến tăng độ phức tạp của mã nguồn.
- Khó hiểu khi số lượng lớp Factory con lớn: Nếu có quá nhiều lớp Factory con, việc quản lý và hiểu mã nguồn có thể trở nên khó khăn.
- Khó khăn trong việc debug: Khi xảy ra lỗi, việc debug có thể trở nên phức tạp hơn do sự phụ thuộc vào các Factory con để tạo ra đối tượng.

5. Creational pattern – Singleton

Vấn đề

Đôi khi, chúng ta làm việc với một chương trình lớn mà trong đó có những struct mỗi khi khởi tạo sẽ tồn rất nhiều tài nguyên, nhưng những hành vi

và phương thức của các đối tượng đó đều giống nhau trong mỗi lần sử dụng. Ví dụ là những object giao tiếp với DB là database, những object đọc ghi các I/O file, những object dùng để log file trong các ứng dụng...

Việc đó dẫn đến một yêu cầu chúng ta chỉ cần tạo ra một object để chạy xuyên suốt trong chương trình.

Hướng giải quyết

Để giải quyết được nhu cầu đó người ta sẽ ứng dụng một design pattern khá nổi tiếng trong ngôn ngữ lập trình đó là *Singleton*.

Design pattern Singleton được thiết kế để đảm bảo rằng chỉ có một đối tượng duy nhất của một lớp được tạo ra và cung cấp một cách để truy cập đối tượng đó từ bất kỳ điểm nào trong ứng dụng.

Các thành phần của design pattern Singleton trong C++ như sau:

Constructor protected: Constructor của lớp Singleton được đặt là protected, nghĩa là chỉ có thể truy cập từ bên trong lớp hoặc từ các lớp con của nó. Điều này ngăn chặn việc tạo ra các đối tượng Singleton từ bên ngoài lớp.

Static method : Lớp Singleton cung cấp một static method để trả về một tham chiếu đến đối tượng Singleton duy nhất. Trong phương thức này, một biến static được khai báo bên trong phương thức để lưu trữ đối tượng Singleton. Biến này được khai báo là static, do đó nó chỉ được khởi tạo một lần duy nhất khi phương thức này được gọi lần đầu tiên.



Hiện thực code

```

#include <iostream>
using namespace std;
class Singleton{
protected:
    Singleton(){}
public:
    int data;
    static Singleton& get_in(){
        static Singleton i;
        return i;
    }
};

int main(){
    Singleton& s1=Singleton::get_in();
    s1.data=10;

    cout<<"s1.data="<<s1.data<<endl;

    Singleton& s2=Singleton::get_in();

    cout<<"s2.data="<<s2.data<<endl;

    s2.data=20;

    cout<<"s1.data="<<s1.data<<endl;
    cout<<"s2.data="<<s2.data<<endl;

    Singleton s_copy=s1;
    s_copy.data=30;

    cout<<"s_copy.data="<<s_copy.data<<endl;
    cout<<"s1.data="<<s1.data<<endl;
    cout<<"s2.data="<<s2.data<<endl;

    return 0;
}

```

- *Class Singleton:*

Lớp Singleton được định nghĩa với một constructor bảo vệ (protected) để không cho phép tạo đối tượng từ bên ngoài lớp.

Lớp có một biến dữ liệu public là data để lưu trữ dữ liệu.

Phương thức get_in() được khai báo là static, nó trả về một tham chiếu (reference) đến một đối tượng Singleton. Phương thức này là điểm truy cập duy nhất đến đối tượng Singleton.

- *Phần main():*

Trong hàm main(), đầu tiên ta gọi phương thức get_in() từ lớp Singleton để lấy một tham chiếu đến đối tượng Singleton duy nhất. Biến s1 và s2 là hai tham chiếu đến cùng một đối tượng Singleton.

Ta gán giá trị 10 cho s1.data và in ra giá trị của s1.data. Sau đó, ta in ra giá trị của s2.data, thấy rằng s1.data và s2.data cùng mang giá trị 10.

Tiếp theo, ta gán giá trị 20 cho s2.data và in ra giá trị của cả s1.data và s2.data. Kết quả thấy rằng cả hai đều mang giá trị 20, do chúng đều trỏ tới cùng một đối tượng.

Tiếp theo, ta tạo một đối tượng mới từ việc copy s1 và gán giá trị 30 cho s_copy.data. Sau đó, in ra giá trị của s_copy.data, s1.data, và s2.data. Kết quả thấy rằng s_copy.data có giá trị là 30, trong khi s1.data và s2.data vẫn giữ nguyên giá trị là 20. Điều này chứng tỏ rằng việc copy không tạo ra một đối tượng mới, mà chỉ tạo ra một tham chiếu mới trỏ đến cùng một đối tượng Singleton.

Lợi ích

- Tạo ra một đối tượng duy nhất.
- Tiết kiệm bộ nhớ.
- Dễ dàng truy cập.
- Khả năng kiểm soát.

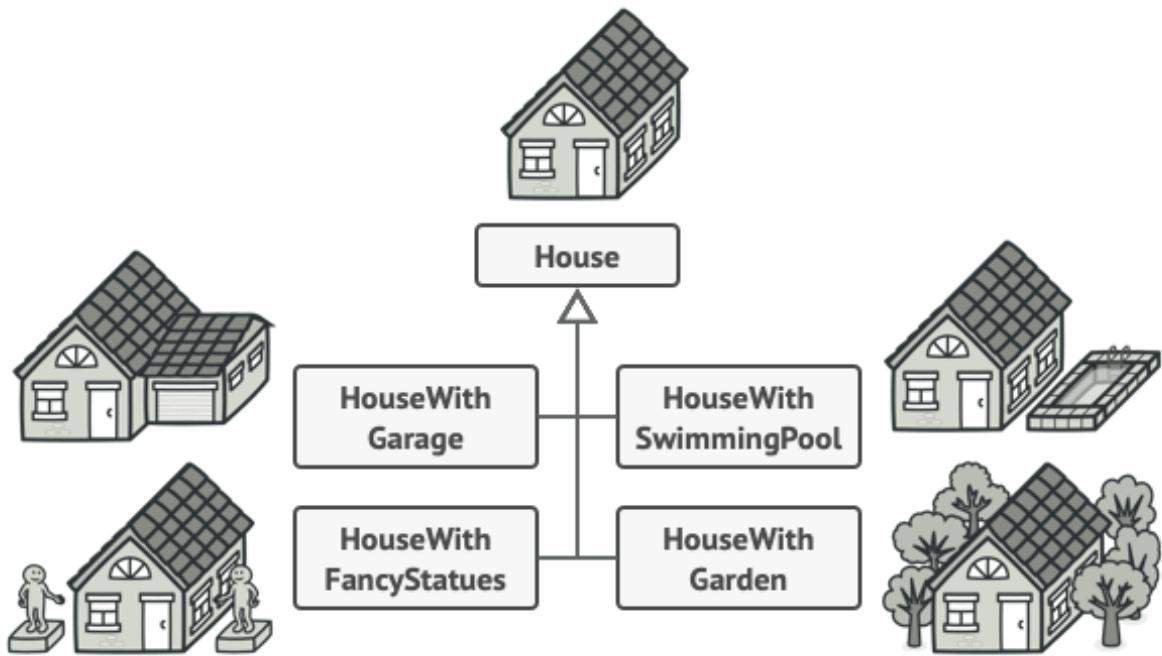
Hạn chế

- Thách thức trong việc kiểm thử.
- Khó khăn trong việc mở rộng.
- Tính đa luồng khi chạy go routine.
- Gây ra sự phụ thuộc.

6.Creational pattern – Builder

Vấn đề

Khi chúng ta cần xây dựng cấu trúc dữ liệu như ngôi nhà mà ở đó các thuộc tính như cửa ra vào, cửa sổ, số tầng của ngôi nhà được biểu diễn dưới dạng các kiểu dữ liệu khác nhau và chúng ta cần nắm rõ cách truyền vào của các tham số đó cũng như kiểu trả về sao cho phù hợp. Từ đó chúng ta xây dựng ra được những đối tượng ngôi nhà cụ thể hơn như ngôi nhà gỗ, ngôi nhà băng...



Hướng giải quyết

Khi đó, chúng ta sẽ sử dụng design pattern *Builder* để hiện thực vấn đề trên. Design pattern Builder được sử dụng để tách biệt quá trình xây dựng đối tượng từ việc biểu diễn của nó, cho phép tạo ra các đối tượng với các thuộc tính khác nhau một cách linh hoạt.

Các thành phần của design pattern này trong c++ như sau:

Class Product: sản phẩm cuối cùng cần xây dựng cụ thể ở đây là ngôi nhà.

Abstract class ibuilder: định nghĩa các phương thức xây dựng một ngôi để các class concrete kế thừa.

Các class concrete builder 1: xây dựng cụ thể nhà gỗ, concrete builder 2: xây dựng cụ thể nhà bê tông.

Class director: để hiện thực các đối tượng cụ thể trên thông qua một builder cụ thể.

Hiện thực code

```
#include <iostream>

using namespace std;

class House {
private:
    string window, door;
    int floor;
public:
    void setwindow(string typewindow){
        window=typewindow;
    }
    void setdoor(string typedoor){
        door=typedoor;
    }
    void setfloor(int num){
        floor=num;
    }
    void getinf(){
        cout<<"Window:"<<window<<endl<<"Door:"<<door<<endl<<"Floor:"<<floor<<endl;
    }
};

class iBuilder {
protected:
    House * mybuilder;
public:
    void createBuilder(){
        mybuilder=new House;
    }
    House* getBuilder() {
        return mybuilder;
    }
    virtual void getwindow() =0;
    virtual void getdoor() =0;
    virtual void getfloor() =0;
};
```

```

class WoodHouse: public iBuilder {
public:
    void getwindow(){
        mybuilder->setwindow("red wood");
    }
    void getdoor(){
        mybuilder->setdoor("brown wood");
    }
    void getfloor(){
        mybuilder->setfloor(2);
    }
};

class IceHouse: public iBuilder {
public:
    void getwindow(){
        mybuilder->setwindow("ice-cream");
    }
    void getdoor(){
        mybuilder->setdoor("ice flower");
    }
    void getfloor(){
        mybuilder->setfloor(1);
    }
};

class Director {
private:
    iBuilder * newhouse;
public:
    void createHouse(iBuilder *tam) {
        newhouse=tam;
    }
    House* getHouse(){
        return newhouse->getBuilder();
    }
    void sethouse(){
        newhouse->createBuilder();
        newhouse->getwindow();
        newhouse->getdoor();
        newhouse->getfloor();
    }
};
}

int main(){
    Director build;
    WoodHouse whouse;
    IceHouse ihouse;
    //Create WoodHouse
    build.createHouse(&whouse);
    build.sethouse();
    House* woodhouse=build.getHouse();
    woodhouse->getinf();

    //Create IceHouse
    build.createHouse(&ihouse);
    build.sethouse();
    House* icehouse=build.getHouse();
    icehouse->getinf();
}

```

- *Class House:*

Là lớp đại diện cho các đối tượng cần xây dựng.

Có các thuộc tính là window (cửa sổ), door (cửa ra vào), và floor (số tầng).

Cung cấp các phương thức để thiết lập các thuộc tính và in thông tin của nhà (getinf()).

- *Abstract Builder Class (iBuilder):*

Là lớp trừu tượng định nghĩa các phương thức cần thiết để xây dựng một House.

Có một con trỏ đến đối tượng House để lưu trữ đối tượng đang được xây dựng.

Có phương thức để tạo mới một đối tượng House (createBuilder()).

Cung cấp phương thức trả về đối tượng House đã được xây dựng (getBuilder()).

Các phương thức thuần ảo getwindow(), getdoor(), và getfloor() để thiết lập các thuộc tính của nhà.

- *Concrete Builder Classes (WoodHouse và IceHouse):*

Là các lớp con của iBuilder, triển khai các phương thức để xây dựng một loại nhà cụ thể.

Mỗi lớp con cài đặt các phương thức getwindow(), getdoor(), và getfloor() để thiết lập các thuộc tính của nhà theo cách đặc trưng cho loại nhà đó.

- *Director Class:*

Là lớp quản lý quá trình xây dựng House.

Có một con trỏ đến một đối tượng iBuilder.

Cung cấp phương thức createHouse() để thiết lập builder cho loại nhà cần xây dựng.

Cung cấp phương thức sethouse() để thực hiện quá trình xây dựng nhà.

Cung cấp phương thức getHouse() để trả về đối tượng House đã được xây dựng.

- *Hàm main():*

Tạo một đối tượng Director.

Tạo hai đối tượng WoodHouse và IceHouse.

Sử dụng Director để xây dựng nhà gỗ và nhà bằng đá lạnh.

In ra thông tin của các nhà đã xây dựng.

Lợi ích

- Tách rời quá trình xây dựng ra khỏi biểu diễn trong class của nó, giúp cho việc tạo ra các đối tượng có cấu trúc phức tạp trở nên dễ dàng hơn.
- Được thêm mới các bước xây dựng mà không cần thay đổi cấu trúc tổng thể của sản phẩm.
- Tạo ra các đối tượng có cấu trúc phức tạp.
- Phân tầng các bước xây dựng đối tượng.

Hạn chế

- Độ phức tạp lớn.

7. Creational pattern – Abstract factory

Vấn đề

Khi chúng ta muốn tạo ra một hay nhiều cấu trúc dữ liệu khác nhau từ những cách khác nhau. Ví dụ như việc tạo ra sản phẩm shoe hoặc short từ hai hãng đồ thể thao khác nhau là Adidas và Nike.



Hướng giải quyết

Để giải quyết vấn đề trên chúng ta có thể dùng design pattern *Abstract factory*, điều này cho phép chúng ta tạo ra các sản phẩm của các nhà sản xuất khác nhau mà không cần quan tâm đến cách chúng được tạo ra.

Các thành phần của design pattern này trong c++ là:

Các class abstract product cụ thể ở đây là class shoe và class short.

Các class concrete product để hiện thực 2 class trừu tượng trên.

Các class abstract factory bao gồm class Adidas và class Nike.

Cũng như các class concrete để hiện thực 2 class trên.

Hiện thực code

```
#include <iostream>
using namespace std;

class Shoe {
public:
    virtual void introduce() = 0;
    virtual ~Shoe() {}
};

class Short {
public:
    virtual void introduce() = 0;
    virtual ~Short() {}
};

class AdidasShoe : public Shoe{
public:
    void introduce() {
        std::cout << "Introducing on a Adidas style Short." << std::endl;
    }
};

class AdidasShort : public Short {
public:
    void introduce() {
        std::cout << "Introducing on a Adidas style Short." << std::endl;
    }
};

class NikeShoe : public Shoe {
public:
    void introduce(){
        std::cout << "Introducing on a Nike style Short." << std::endl;
    }
};

class NikeShort : public Short {
public:
    void introduce() {
        std::cout << "Introducing on a Nike style Short." << std::endl;
    }
};
```

```

class Factory {
public:
    virtual Shoe* createShoe() = 0;
    virtual Short* createShort() = 0;
    virtual ~Factory() {}
};

class Adidas : public Factory {
public:
    Shoe* createShoe() {
        return new AdidasShoe();
    }
    Short* createShort(){
        return new AdidasShort();
    }
};

class Nike : public Factory {
public:
    Shoe* createShoe() {
        return new NikeShoe();
    }
    Short* createShort() {
        return new NikeShort();
    }
};

int main() {
    Factory* adidas = new Adidas();
    Shoe* ShoesShoe = adidas->createShoe();
    Short* ShoesShort = adidas->createShort();
    ShoesShoe->introduce();
    ShoesShort->introduce();

    Factory* nike = new Nike();
    Shoe* linuxShoe = nike->createShoe();
    Short* linuxShort = nike->createShort();
    linuxShoe->introduce();
    linuxShort->introduce();

    delete ShoesShoe;
    delete ShoesShort;
    delete adidas;
    delete linuxShoe;
    delete linuxShort;
    delete nike;

    return 0;
}

```

- *Abstract Base Classes:*

Shoe và Short là hai lớp trừu tượng định nghĩa các phương thức thuận ảo introduce(). Các lớp cụ thể sẽ triển khai phương thức này để giới thiệu các sản phẩm của họ.

- *Concrete Classes:*

AdidasShoe, AdidasShort, NikeShoe, và NikeShort là các lớp cụ thể triển khai các phương thức introduce() cho các sản phẩm của Adidas và Nike.

- *Abstract Factory:*

Factory là lớp trừu tượng định nghĩa các phương thức tạo ra các sản phẩm (createShoe() và createShort()). Các lớp cụ thể sẽ triển khai các phương thức này để tạo ra các sản phẩm của họ.

- *Concrete Factories:*

Adidas và Nike là hai lớp cụ thể triển khai các phương thức tạo ra các sản phẩm của Adidas và Nike.

- *Hàm main():*

Trong hàm main(), chúng ta tạo ra một đối tượng Factory cho Adidas và Nike.

Sau đó, chúng ta sử dụng các phương thức createShoe() và createShort() để tạo ra các sản phẩm của Adidas và Nike.

Lợi ích

- Tách rời việc tạo ra đối tượng.
- Khả năng mở rộng ra nhiều class khác và giảm sự phụ thuộc giữa các class.
- Tính linh hoạt cao khi muốn tạo ra một đối tượng mới chỉ cần thêm một class con và class Factory.

Hạn chế

- Phức tạp do số lượng class nhiều.
- Phải viết thêm class Factory.

8. Creational pattern – Prototype

Vấn đề

Khi chúng ta muốn tạo ra một đối tượng có những đặc điểm và hành vi giống y hệt với một đối tượng nào đó đã có sẵn, ví như cơ chế copy giữa các file và folder trong máy tính thường ngày.



Hướng giải quyết

Khi đó, ta có thể dùng design pattern *Prototype* để giải quyết vấn đề trên. Mẫu này cho phép tạo ra các đối tượng mới bằng cách sao chép (clone) các đối tượng đã tồn tại thay vì tạo mới hoàn toàn từ đầu. Hữu ích khi việc tạo một đối tượng mới từ đầu là tốn kém hoặc phức tạp.

Các thành phần của design pattern này trong c++ như sau:

Abstract class prototype chứa phương thức ít nhất một phương thức copy().

Các concrete class đại diện cho những đối tượng cụ thể để hiện thực phương thức copy() trong hàm trừu tượng, ở vấn đề này bao gồm class File và class Folder.

Hiện thực code

```

class Inode {
public:
    virtual Inode* clone() = 0;
    virtual void print() = 0;
};

class File : public Inode {
private:
    string name;
public:
    File (string n):name(n){}
    Inode* clone(){
        return new File(this->name);
    }
    void print(){
        cout<<this->name;
    }
};

class Folder : public Inode{
private:
    string name;
    Inode* a[100];
    int m;
public:
    Folder(string n):name(n),m(0){}
    void InsertFile(Inode *f){
        a[m++]=f;
    }
    Inode* clone(){
        Folder *tam=new Folder(this->name);
        for(int i=0;i<this->m;i++){
            tam->InsertFile(this->a[i]->clone());
        }
        return tam;
    }
    void print(){
        cout<<this->name<<" :"<<endl;
        for(int i=0;i<m;i++){
            a[i]->print();
            cout<<endl;
        }
    }
};

```

```

int main(){
    File *file1=new File("file 1");
    File *file2=new File("file 2");
    File *file3=new File("file 3");
    Folder *folder1=new Folder("folder 1");
    folder1->InsertFile(file1);
    Folder *folder2=new Folder("folder 2");
    folder2->InsertFile(folder1);
    folder2->InsertFile(file2);
    folder2->InsertFile(file3);
    Inode *copyfolder=folder2->clone();
    copyfolder->print();
    return 0;
}

```

- *Abstract Base Class Inode:*

Là lớp cơ sở trừu tượng định nghĩa hai phương thức thuần ảo: clone() để tạo ra một bản sao của đối tượng và print() để in thông tin của đối tượng.

- *Concrete Class File:*

Là một lớp con của Inode, đại diện cho các tệp tin.

Có một thuộc tính là name để lưu trữ tên của tệp tin.

Triển khai phương thức clone() để tạo ra một bản sao của đối tượng File.

Triển khai phương thức print() để in tên của tệp tin.

- *Concrete Class Folder:*

Là một lớp con của Inode, đại diện cho các thư mục.

Có hai thuộc tính là name để lưu trữ tên của thư mục và a[] là một mảng chứa các con trỏ đến các đối tượng Inode.

Có một biến m để theo dõi số lượng đối tượng trong mảng a[].

Triển khai phương thức InsertFile() để chèn một đối tượng Inode vào mảng a[].

Triển khai phương thức clone() để tạo ra một bản sao của đối tượng Folder, bao gồm việc tạo ra bản sao của các đối tượng con trong mảng a[].

Triển khai phương thức print() để in tên của thư mục và gọi phương thức print() của các đối tượng con trong mảng a[].

- *Hàm main():*

Tạo các đối tượng File và Folder.

Chèn các đối tượng File và Folder vào Folder khác.

Tạo ra một bản sao của Folder sử dụng phương thức clone().

In thông tin của thư mục bản sao.

Lợi ích

- Giảm tài nguyên tạo đối tượng.
- Tăng hiệu suất nhờ việc copy đối tượng đang tồn tại.
- Dễ mở rộng mà không cần thay đổi code hiện tại.
- Đối tượng không phụ thuộc vào lớp cụ thể bằng việc dùng một class chung nhất thay cho “new”.

Hạn chế

- Phức tạp khi đối tượng có liên kết với đối tượng khác như danh sách liên kết đơn, đôi.
- Phương thức clone() phức tạp.

9. Behavioral pattern – Chain of Responsibility

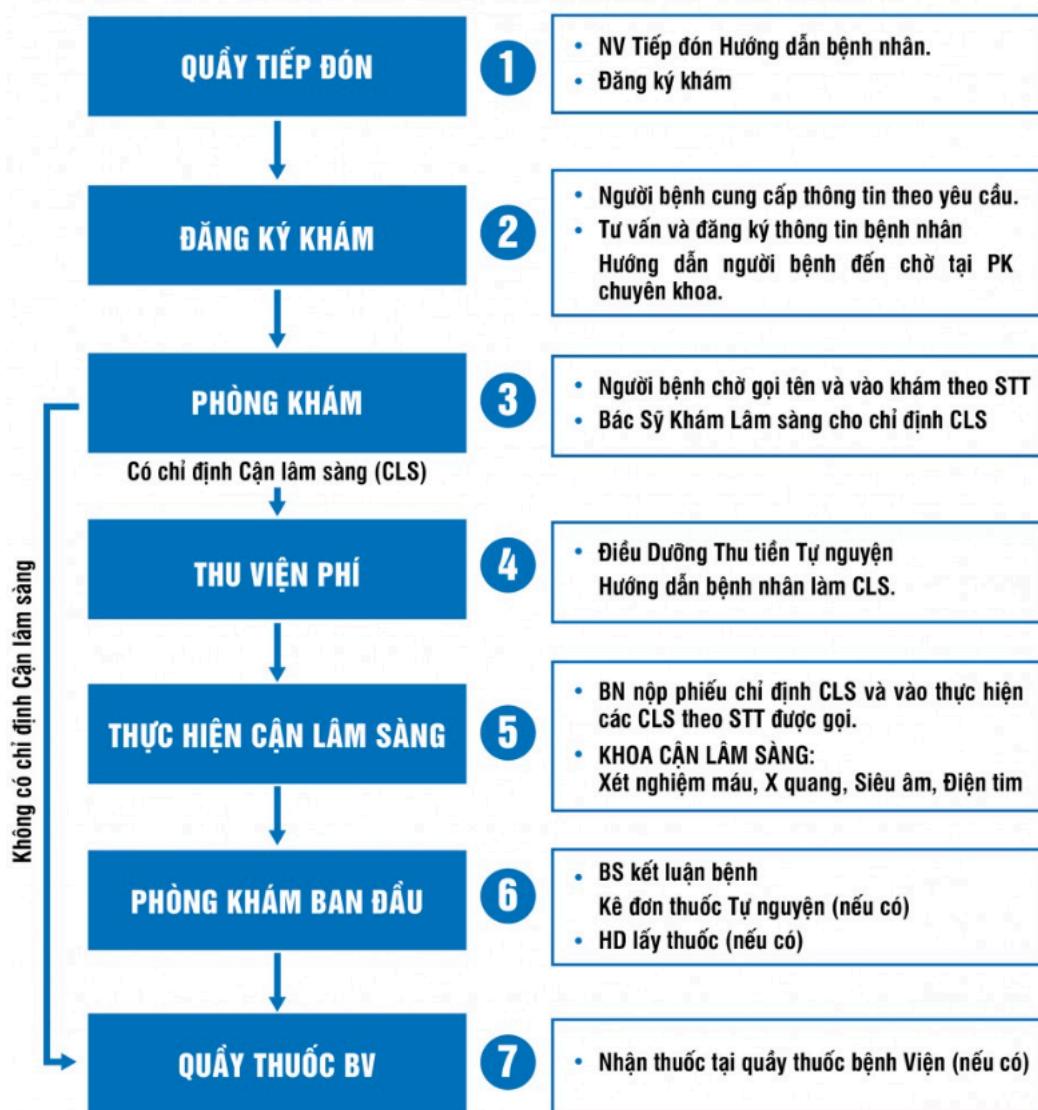
Vấn đề

Khi chúng ta muốn xử lý đối tượng nào đó với các hành vi được diễn ra tuần tự và cuối cùng sẽ kết thúc xử lý đối tượng bằng việc hoàn thành chuỗi hành vi tuần tự

đó. Ví dụ trong một hệ thống cấu trúc dữ liệu quản lý hoạt động khám bệnh của các bệnh nhân trong một bệnh viện lớn, bắt đầu bằng việc nhận hồ sơ đăng ký ở quầy lễ tân, sau đó được bác sĩ khám kiểm tra, tiếp đến là việc nhận thuốc và cuối cùng là việc thanh toán.

QUY TRÌNH KHÁM BỆNH NGOẠI TRÚ

(Dành cho bệnh nhân khám Tự nguyện)



Hướng giải quyết

Khi đó, chúng ta sẽ dùng design pattern *Chain of Responsibility* để hiện thực vấn đề trên. Mẫu này cho phép gửi yêu cầu qua một chuỗi các đối tượng mà không cần biết đối tượng nào sẽ xử lý yêu cầu đó. Mỗi đối tượng trong chuỗi có cơ hội để xử lý yêu cầu hoặc chuyển tiếp nó cho đối tượng kế tiếp trong chuỗi.

Các thành phần của design pattern này trong c++ như sau:

Class client để biểu diễn đối tượng cần xử lý: class Patient.

Class handle trùu tượng chứa các phương thức rỗng cần thiết và ít nhất có một phương thức SetNext để di chuyển đến hành động tiếp theo: class Chain

Các class concrete handler để hiện thực các hành vi cụ thể: class Reception, class Doctor, class Medical, class Cashier.

Hiện thực code

```

#include <iostream>
using namespace std;

class Patient {
public:
    bool doctorcheck;
    bool receptioncheck;
    bool medicalcheck;
    bool cashiercheck;
};

class Chain {
public:
    virtual void check(Patient *p) = 0;
    virtual void setnext(Chain *next) = 0;
};

class Reception : public Chain{
private:
    Chain *next;
public:
    void check(Patient *p){
        if(p->receptioncheck){
            cout<<"Resgiter done!";
            this->next->check(p);
            return;
        }
        else {
            p->receptioncheck=true;
        }
        this->next->check(p);
    }
    void setnext(Chain* next){
        this->next=next;
    }
};

class Doctor : public Chain{
private:
    Chain *next;
public:
    void check(Patient *p){
        if(p->doctorcheck){
            cout<<"Doctor done!";
            this->next->check(p);
            return;
        }
        else {
            p->doctorcheck=true;
        }
        this->next->check(p);
    }
    void setnext(Chain* next){
        this->next=next;
    }
};

class Medical : public Chain{
private:
    Chain *next;
public:
    void check(Patient *p){
        if(p->medicalcheck){
            cout<<"Medical done!";
            this->next->check(p);
            return;
        }
        else {
            p->medicalcheck=true;
        }
        this->next->check(p);
    }
    void setnext(Chain* next){
        this->next=next;
    }
};

```

```

class Cashier : public Chain{
private:
    Chain *next;
public:
    void check(Patient *p){
        if(p->cashiercheck){
            cout<<"Pay done!";
            this->next->check(p);
            return;
        }
        else {
            p->cashiercheck=true;
        }
        cout<<"Done!";
    }
    void setnext(Chain* next){}
};

int main(){
    Cashier* pay=new Cashier();
    Medical* med=new Medical();
    med->setnext(pay);
    Doctor* doc=new Doctor();
    doc->setnext(med);
    Reception* reg=new Reception();
    reg->setnext(doc);
    Patient* p=new Patient();
    p->receptioncheck=false;
    p->doctorcheck=false;
    p->medicalcheck=false;
    p->cashiercheck=false;
    reg->check(p);
}

```

- *Class Patient:*

Định nghĩa một lớp để đại diện cho bệnh nhân. Có các biến để đánh dấu việc kiểm tra từng bước của quy trình điều trị.

- *Abstract Class Chain:*

Định nghĩa một giao diện chung cho các bước trong quy trình kiểm tra của bệnh nhân.

Có hai phương thức thuần ảo là check() để kiểm tra bệnh nhân và setnext() để thiết lập bước tiếp theo trong chuỗi.

- *Concrete Classes (Reception, Doctor, Medical, Cashier):*

Mỗi lớp triển khai phương thức check() để xác định xem bệnh nhân đã hoàn thành bước kiểm tra hay chưa. Nếu chưa, nó sẽ thực hiện bước kiểm tra và chuyển tiếp đến bước tiếp theo trong chuỗi.

Mỗi lớp cũng triển khai phương thức setnext() để thiết lập bước tiếp theo trong chuỗi.

- *Hàm main():*

Tạo các đối tượng của các bước kiểm tra (Reception, Doctor, Medical, Cashier) và thiết lập chuỗi liên kết giữa chúng bằng cách sử dụng phương thức setnext().

Tạo một đối tượng Patient đại diện cho bệnh nhân và đánh dấu rằng bệnh nhân chưa hoàn thành bất kỳ bước kiểm tra nào.

Gọi phương thức check() trên bước kiểm tra đầu tiên (Reception) và bắt đầu quá trình kiểm tra chuỗi. Khi một bước kiểm tra hoàn thành, nó chuyển tiếp sang bước tiếp theo trong chuỗi.

Lợi ích

- Giảm sự phụ thuộc giữa người gửi và người nhận.
- Dễ dàng thay đổi vị trí hoặc thêm mới một hay nhiều hành vi.
- Phân loại các yêu cầu.
- Có thể thêm các class handler mới để mở rộng nhiều chuỗi hành vi.

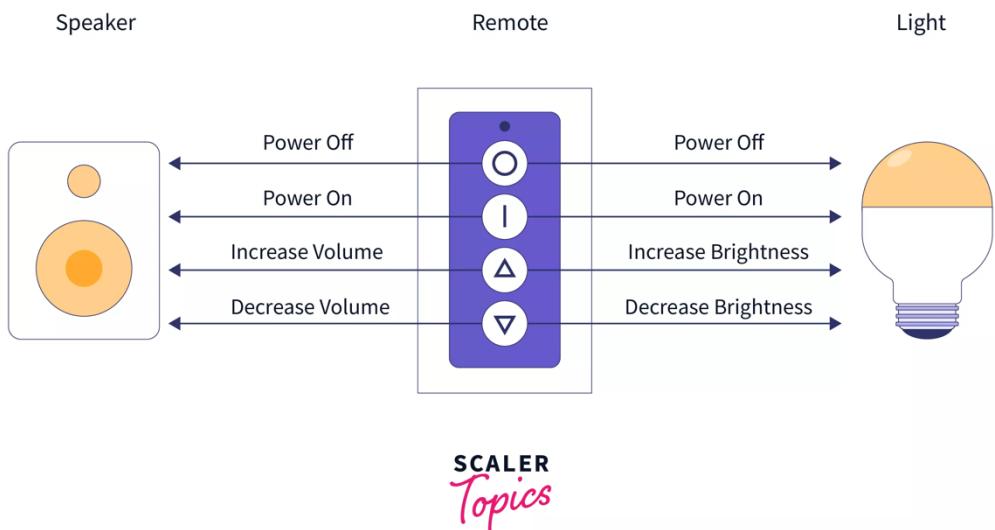
Hạn chế

- Hành vi sẽ không được xử lý nếu lệch khỏi chuỗi liên kết các hành vi.
- Hiệu suất giảm do đi qua nhiều khâu.
- Khó debug
- Quản lý khó khăn nếu chuỗi các hành vi quá dài hoặc có quá nhiều handler.

10.Behavioral pattern – Command

Vấn đề

Khi chúng ta cần đưa ra một hay nhiều yêu cầu đến một hay nhiều đối tượng mà chúng ta muốn giấu đi hành động phía sau để thực hiện các yêu cầu đó hoặc giấu đi các đối tượng nhận để thực hiện các yêu cầu đó. Giả dụ chúng ta cần thực hiện một cấu trúc dữ liệu remote để bật tắt tivi mà ta không muốn cho mọi người nhìn thấy tín hiệu được truyền đi từ remote gửi đến tivi khi bấm nút bật/tắt trên remote.



SCALER
Topics

Hướng giải quyết

Khi đó, chúng ta sẽ sử dụng design pattern *Command* để thực hiện vấn đề trên. Command trong lập trình hướng đối tượng là một kỹ thuật mà trong đó một đối tượng được sử dụng để đóng gói tất cả thông tin cần thiết để thực hiện một hành động hoặc kích hoạt một sự kiện sau này. Thông tin này bao gồm tên phương thức, đối tượng mà phương thức thuộc về, và các giá trị đối số của phương thức.

Các thành phần của design pattern này trong c++ như sau:

Class Command thuần ảo: Lớp trừu tượng mà mọi Command cụ thể phải tuân theo. Thường có một phương thức để thực thi lệnh, như `execute()`.

Concrete Command: Các lớp triển khai giao diện Command, mỗi lớp đại diện cho một hành động cụ thể.

Invoker: Đối tượng yêu cầu command thực thi. Invoker không cần biết chi tiết cách thực thi hành động như thế nào.

Receiver: Đối tượng chứa logic thực tế để thực hiện hành động. Command có thể gọi một hoặc nhiều phương thức trên Receiver.

Client: Tạo ra các Concrete Command và thiết lập các đối tượng Receiver của chúng.

Hiện thực code

```

#include <iostream>
using namespace std;

class Command{
public:
    virtual void execute ()= 0;
};

class Device {
public:
    virtual void on () = 0;
    virtual void off() = 0;
};

class Button {
private:
    Command* c;
public:
    Button(Command *co):c(co){}
    void press(){
        this->c->execute();
    }
};

class Oncommand : public Command{
private:
    Device * d;
public:
    Oncommand(Device *de):d(de){}
    void execute() {
        d->on();
    }
};

class Offcommand : public Command{
private:
    Device * d;
public:
    Offcommand(Device *de):d(de){}
    void execute() {
        d->off();
    }
};

```

```

class TV : public Device {
private:
    bool isTurn;
public:
    void on(){
        isTurn=true;
        cout<<"TV on" << endl;
    }
    void off(){
        isTurn=false;
        cout<<"TV off" << endl;
    }
};

int main(){
    TV* t=new TV();
    Oncommand* on=new Oncommand(t);
    Button* OnButton=new Button(on);
    OnButton->press();
    Offcommand* off=new Offcommand(t);
    Button* OffButton=new Button(off);
    OffButton->press();
}

```

- *Command (Abstract)*

Đây là interface cho các lệnh. Nó chỉ định phương thức execute() mà mọi lệnh cụ thể phải cài đặt.

- *Device (Abstract)*

Interface cho các thiết bị, với các phương thức on() và off() để bật và tắt thiết bị.

- *Button*

Đại diện cho một nút bấm vật lý hoặc ảo. Nó chứa một tham chiếu đến đối tượng Command mà nó sẽ kích hoạt khi được bấm.

Phương thức press() khi được gọi, sẽ gọi phương thức execute() của Command được liên kết với nó.

- *OnCommand*

Là một lớp cụ thể của Command dùng để bật thiết bị.

Khi phương thức execute() được gọi, nó sẽ kích hoạt phương thức on() của thiết bị được liên kết.

- *OffCommand*

Là một lớp cụ thể khác của Command dùng để tắt thiết bị.

Tương tự như OnCommand, khi phương thức execute() được gọi, nó sẽ kích hoạt phương thức off() của thiết bị được liên kết.

- *TV*

Là một lớp cụ thể của Device. Cài đặt cách thiết bị TV phản ứng khi bật và tắt.

Có trạng thái isTurn để theo dõi trạng thái bật/tắt của TV.

- *Main()*

Một đối tượng TV được tạo ra.

Đối tượng OnCommand được tạo ra với đối tượng TV làm đối số. Điều này liên kết OnCommand với TV để khi execute() được gọi, nó sẽ bật TV.

Một Button được tạo với OnCommand. Khi nút này được bấm (gọi press()), nó kích hoạt phương thức on() của TV thông qua OnCommand.

Tương tự, một đối tượng OffCommand và một Button được tạo cho việc tắt TV.

Khi nút tắt được bấm, TV sẽ được tắt.

Lợi ích

- Tách rời phần code khỏi phần thực thi.
- Dễ dàng thêm các lệnh mới.
- Cơ sở của undo, redo, logfile.

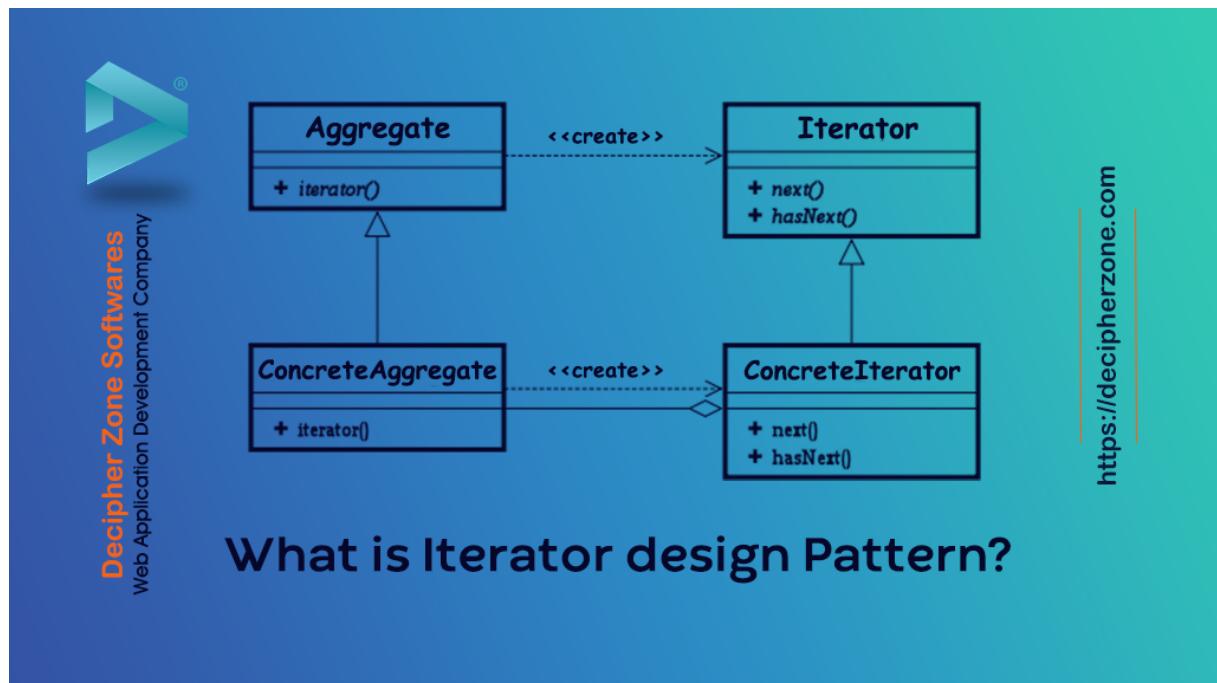
Hạn chế

- Phức tạp do cần triển khai nhiều lớp trừu tượng.
- Cần quản lý nhiều trạng thái.

11. Behavioral pattern – Iterator

Vấn đề

Khi chúng ta cần một đối tượng để duyệt qua các cấu trúc dữ liệu gồm có các tập hợp cụ thể mà không cần quan tâm đến bên trong cấu trúc dữ liệu đó có gì nhằm những mục đích như xoá, thêm, thay đổi hay lấy giá trị một vài đối tượng con có tính tuần tự trong tập hợp đó.



Hướng giải quyết

Khi đó, ta sử dụng design pattern *Iterator* để biểu diễn vấn đề trên. *Iterator* được sử dụng để cung cấp một cách để truy cập tuần tự các phần tử của một đối tượng tập hợp mà không cần phải biết đến cách thức lưu trữ nội bộ của nó. Mẫu này giúp tách rời các thuật toán trên các tập hợp khỏi cách thức thực tế của cách tập hợp này

được lưu trữ và quản lý. Điều này không chỉ làm giảm sự phụ thuộc giữa các lớp mà còn tăng cường tính mô-đun và tái sử dụng mã.

Các thành phần của design pattern này trong c++ bao gồm:

Class client là cấu trúc dữ liệu thành phần của tập hợp: Class Node.

Class Collection thuần ảo chứa tập hợp của class client và có chứa phương thức tạo ra iterator: Class List.

Class concrete để hiện thực cụ thể class thuần ảo trên: Class NodeList.

Class Iterator thuần ảo chứa ít nhất hai phương thức ảo là setNext() và hasNext(): Class Iterator.

Class concrete iterator để hiện thực class thuần ảo trên: Class NodeIterator.

Thực hiện code

```

class Node {
public:
    int value;
    Node(int val) : value(val) {}
};

class Iterator {
public:
    virtual Node* getNext () =0;
    virtual bool hasNext () =0;
};

class List {
public:
    virtual Iterator* createI() = 0;
};

class NodeIterator :public Iterator{
private:
    Node *a[100];
    int index;
    int num;
public:
    NodeIterator(Node* node[],int m){
        num=m;
        for(int i=0;i<m;i++){
            a[i]=node[i];
        }
        index=0;
    }
    Node* getNext(){
        return a[index++];
    }
    bool hasNext(){
        if(index<num) return true;
        else return false;
    }
};
class NodeList : public List{
private:
    Node* a[100];
    int num;
public:
    void add(Node* node){
        a[num++]=node;
    }
    Iterator* createI(){
        return new NodeIterator(a,num);
    }
};

int main(){
    NodeList *list=new NodeList();
    list->add(new Node(1));
    list->add(new Node(2));
    list->add(new Node(3));
    Iterator*i =list->createI();
    while(i->hasNext()){
        cout<<i->getNext()->value<<endl;
    }
}

```

- *Node Class:*

Lớp này đơn giản đại diện cho một nút trong danh sách. Mỗi nút chứa một giá trị nguyên.

- *Iterator Interface:*

Là một interface định nghĩa các phương thức cần thiết cho một Iterator, bao gồm `getNext()` để lấy nút tiếp theo và `hasNext()` để kiểm tra xem có nút tiếp theo không.

- *List Interface:*

Interface này định nghĩa một phương thức `createI()` để tạo ra một Iterator cho danh sách.

- *NodeIterator Class:*

Là một lớp cụ thể của Iterator interface. Nó duyệt qua một mảng các nút được cung cấp trong constructor và triển khai phương thức `getNext()` và `hasNext()` để trả về nút tiếp theo và kiểm tra xem có nút tiếp theo không.

- *NodeList Class:*

Là một lớp cụ thể của List interface. Nó duy trì một mảng các nút và triển khai phương thức `add()` để thêm nút vào danh sách và phương thức `createI()` để tạo ra một Iterator cho danh sách.

- *Trong hàm main():*

Một NodeList được tạo ra và một số nút được thêm vào danh sách. Sau đó, một Iterator được tạo ra từ danh sách và dùng để duyệt qua danh sách các nút, in giá trị của mỗi nút ra màn hình.

Lợi ích

- Tính tách biệt khi duyệt qua các phần tử.
- Dễ sử dụng khi iterator cung cấp phương pháp duyệt mà không cần quan tâm đến chi tiết.
- Cho phép thay đổi cách lưu trữ mà không cần thay đổi mã duyệt.
- Tạo ra nhiều iterator cho các loại cấu trúc dữ liệu khác nhau.

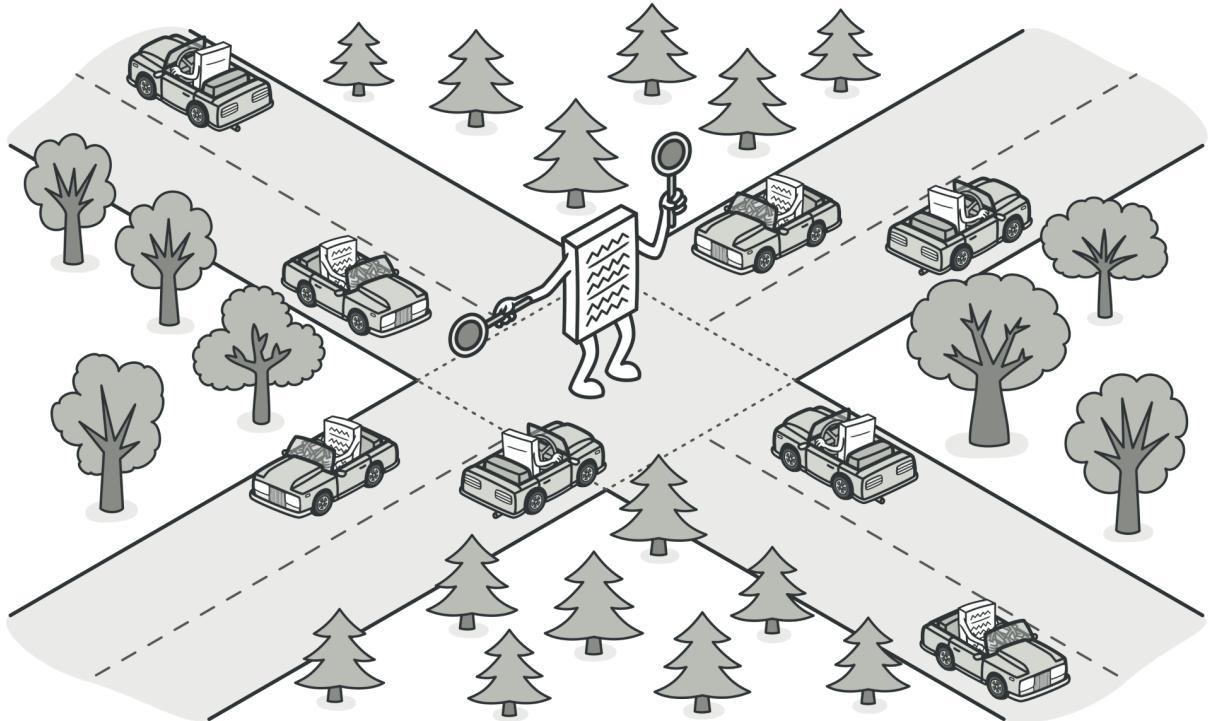
Hạn chế

- Dễ gây overhead.
- Phức tạp nếu muốn đi ngược lại hoặc ứng dụng trong các cấu trúc dữ liệu phức tạp.
- Không phù hợp cho tất cả các loại dữ liệu, đặc biệt là dữ liệu có cấu trúc phức tạp hoặc dữ liệu được tổ chức theo cách không tuân tự.

12. Behavioral pattern – Mediator

Vấn đề

Khi có hai hay nhiều đối tượng cần giao tiếp với nhau để giải quyết hành vi mà chúng không nhất thiết phải truy cập vào nhau. Giả như có 2 đoàn tàu Đỏ và Xanh cùng dùng chung một đường ray và cần phải có một trạm điều phối để đảm bảo 2 đoàn tàu trên không lúc nào chạy cùng lúc với nhau để tránh xảy ra va chạm.



Hướng giải quyết

Khi đó, chúng ta cần design pattern *Mediator* như một người trung gian để giao tiếp giữa các đối tượng. Mediator được sử dụng để giảm sự phụ thuộc giữa các thành phần trong hệ thống, thay vì các thành phần giao tiếp trực tiếp với nhau, chúng giao tiếp thông qua một đối tượng trung gian gọi là Mediator. Mediator giúp giảm sự phức tạp và sự phụ thuộc trong mã nguồn bằng cách kiểm soát và tăng tính mô-đun hóa của các thành phần trong hệ thống.

Các thành phần của design pattern này trong c++ như sau:

Class thuần ảo client : class train.

Các concrete class hiện thực cho class trên: class Redtrain và class Bluetrain.

Class thuần ảo mediator: class Mediator.

Class concrete cho class trên: class Station.

Hiện thực code

```
#include <iostream>
#include<queue>
using namespace std;

class train{
public:
    virtual void checktogo() = 0;
};

class Mediator{
    virtual bool PermitGo(train* tr)=0;
    virtual bool Waiting(train *tr)=0;
    virtual void Changetrain(train *tr) = 0;
};

class Station:public Mediator{
private:
    queue<train*> q;
public:
    bool PermitGo(train* tr){
        if(q.empty()) {
            q.push(tr);
            return true;
        }
        return false;
    }
    bool Waiting(train* tr){
        if(q.front()!=tr) return true;
        return false;
    }
    void Changetrain(train*tr){
        q.pop();
        q.push(tr);
    }
};
```

```

class Redtrain:public train{
private:
    Station *med;
public:
    Redtrain(Station *m ):med(m){}
    void checktogo(){
        if(med->PermitGo(this)) cout<<"Red train ready to go" << endl;
        else if(med->Waiting(this)){
            med->Changetrain(this);
            cout<<"The blue train will finish" << endl;
        }
        else{
            cout<<"The red train is running" << endl;
        }
    }
};

class Bluetrain:public train{
private:
    Station *med;
public:
    Bluetrain(Station *m ):med(m){}
    void checktogo(){
        if(med->PermitGo(this)) cout<<"Blue train ready to go" << endl;
        else if(med->Waiting(this)){
            med->Changetrain(this);
            cout<<"The red train will be finish" << endl;
        }
        else{
            cout<<"The blue train is running" << endl;
        }
    }
};

int main(){
    Station * sta=new Station();
    Redtrain *r=new Redtrain(sta);
    Bluetrain *b=new Bluetrain(sta);
    r->checktogo();
    b->checktogo();
    b->checktogo();
}

```

- *Mediator (Trung gian):*

Là một lớp trùu tượng định nghĩa các phương thức cần thiết để điều phối giao tiếp giữa các đối tượng. Trong trường hợp này, Mediator có các phương thức PermitGo, Waiting, và Changetrain để điều phối việc điều khiển các đoàn tàu.

- *Station:*

Là một lớp cụ thể của Mediator. Nó triển khai các phương thức của Mediator để quản lý việc điều khiển các đoàn tàu tại trạm. Trong đoạn code, Station sử dụng một hàng đợi (queue) để xếp các đoàn tàu.

- *Train:*

Là một lớp trùu tượng đại diện cho các loại tàu.

- *Redtrain và Bluetrain:*

Là các lớp con của train. Mỗi lớp này triển khai phương thức checktogo, trong đó tương tác với Mediator để kiểm tra xem tàu có thể đi tiếp được hay không.

- *Cách hoạt động của chương trình:*

Trong hàm main, một đối tượng Station được tạo ra để đại diện cho trạm.

Một đoàn tàu màu đỏ (Redtrain) và một đoàn tàu màu xanh (Bluetrain) được tạo ra và chúng được liên kết với trạm thông qua tham số truyền vào constructor.

Sau đó, các phương thức checktogo được gọi để kiểm tra xem các đoàn tàu có thể tiếp tục đi tiếp hay không.

Trong quá trình kiểm tra, các đoàn tàu giao tiếp với trạm thông qua phương thức được triển khai từ Mediator, và dựa vào kết quả trả về từ trạm để quyết định hành động tiếp theo.

- *Kết quả:*

Khi chạy chương trình, đoàn tàu màu đỏ được phép đi trước vì không có đoàn tàu nào khác ở trạm.

Sau đó, đoàn tàu màu xanh cố gắng đi, nhưng phải chờ đợi vì đoàn tàu màu đỏ đang đi.

Khi đoàn tàu màu xanh cố gắng đi lần thứ hai, đoàn tàu màu đỏ đã kết thúc, nên nó được phép đi tiếp.

Lợi ích

- Các đối tượng không cần giao tiếp trực tiếp.
- Tái sử dụng.
- Giảm độ phức tạp.
- Mediator cho phép thêm các loại đối tượng mới vào hệ thống mà không cần sửa code nhiều.

Hạn chế

- Phụ thuộc nhiều vào Mediator, nếu class này có vấn đề thì cả hệ thống cũng vậy.
- Quá tải do xử lý nhiều phép logic.
- Trong vài trường hợp, khó hiểu hết các lớp cấu trúc để duy trì các đối tượng qua việc giao tiếp bằng Mediator.

13. Behavioral pattern – Memento

Vấn đề

Khi chúng ta muốn lưu trữ các trạng thái đã xuất hiện của một đối tượng và có thể truy xuất đến trạng thái bất kì bất cứ lúc nào để cập nhật lại trạng thái cho đối tượng đó. Ví dụ ở đây ta sẽ giải quyết một cấu trúc dữ liệu thể hiện cảm xúc trên khuôn mặt của người máy.



Hướng giải quyết

Khi này, chúng ta sẽ sử dụng design pattern *Memento* để hiện thực vấn đề trên. Mẫu thiết kế này được sử dụng để lưu trữ trạng thái bên trong của một đối tượng mà không vi phạm tính đóng gói (encapsulation). Memento cho phép khôi phục trạng thái đã lưu trước đó của đối tượng.

Các thành phần của design pattern này trong c++ như sau:

Class Originator là lớp đối tượng cần lưu trạng thái: class Face.

Class Memento là lớp trạng thái tại một thời điểm nào đó: class Emoji.

Class Caretaker là lớp lưu trữ các trạng thái và là cầu nối giữa 2 class trên.

Hiện thực code

```

class Emoji {
public:
    string state;
    Emoji(string s):state(s){}
};

class Face {
private:
    string state;
public:
    Face(string s):state(s){}
    Emoji * create(){
        return new Emoji(state);
    }
    string getstate(){ return state;}
    void setstate(Emoji *e){
        this->state=e->state;
    }
    void setstate(string st){
        this->state=st;
    }
};

class caretaker{
private:
    Emoji* states[100];
    int m;
public:
    caretaker():m(0){}
    Emoji* get(int index){
        return states[index];
    }
    void add(Emoji* e){
        states[m++]=e;
    }
};

```

```

int main(){
    caretaker* a=new caretaker();
    Face *face=new Face("Happy");
    a->add(face->create());
    face->setstate("Sad");
    a->add(face->create());
    face->setstate("Angry");
    face->setstate(a->get(0));
    cout<<face->getstate();
}

```

- *Emoji:*

Là lớp đại diện cho trạng thái của một biểu tượng cảm xúc (emoji). Trong đoạn mã này, trạng thái được đại diện bằng một chuỗi (string).

- *Face:*

Là lớp đại diện cho một khuôn mặt có thể thay đổi trạng thái của nó.

Lớp này có một trường dữ liệu state để lưu trữ trạng thái hiện tại của khuôn mặt.

Phương thức `create()` được sử dụng để tạo một đối tượng Emoji từ trạng thái hiện tại của khuôn mặt.

Phương thức `setstate()` được sử dụng để thiết lập trạng thái mới của khuôn mặt, cả từ một chuỗi hoặc từ một đối tượng Emoji.

- *Caretaker:*

Là một lớp để quản lý lịch sử của các trạng thái đã lưu của khuôn mặt.

Nó lưu trữ các đối tượng Emoji trong một mảng và cung cấp các phương thức để thêm mới đối tượng Emoji vào mảng và lấy đối tượng Emoji từ mảng theo chỉ mục.

- *Trong hàm main:*

Một đối tượng `Caretaker a` được tạo ra để quản lý lịch sử trạng thái của khuôn mặt.

Một đối tượng Face `*face` được tạo ra với trạng thái ban đầu là "Happy".

Đối tượng Emoji tạo ra từ trạng thái hiện tại của khuôn mặt được thêm vào `Caretaker a`.

Trạng thái của khuôn mặt được thiết lập là "Sad", sau đó được thêm vào `Caretaker a` dưới dạng một đối tượng Emoji mới.

Trạng thái của khuôn mặt tiếp theo được thiết lập là "Angry".

Trạng thái của khuôn mặt được thiết lập lại từ trạng thái đã lưu trong `Caretaker a` ở vị trí đầu tiên trong mảng. Điều này làm cho trạng thái của khuôn mặt trở lại là "Happy".

Kết quả cuối cùng là xuất trạng thái của khuôn mặt, mà lúc này là "Happy".

Lợi ích

- Khôi phục được trạng thái trước.
- Tính bao đóng.
- Giảm độ phức tạp khi các đối tượng có thể tự quản lý trạng thái của chính mình.
- Ghi lại lịch sử trạng thái.

Hạn chế

- Tiêu tốn bộ nhớ và chi phí quản lý trạng thái.
- Thiết kế Memento có thể làm phức tạp thiết kế của system.

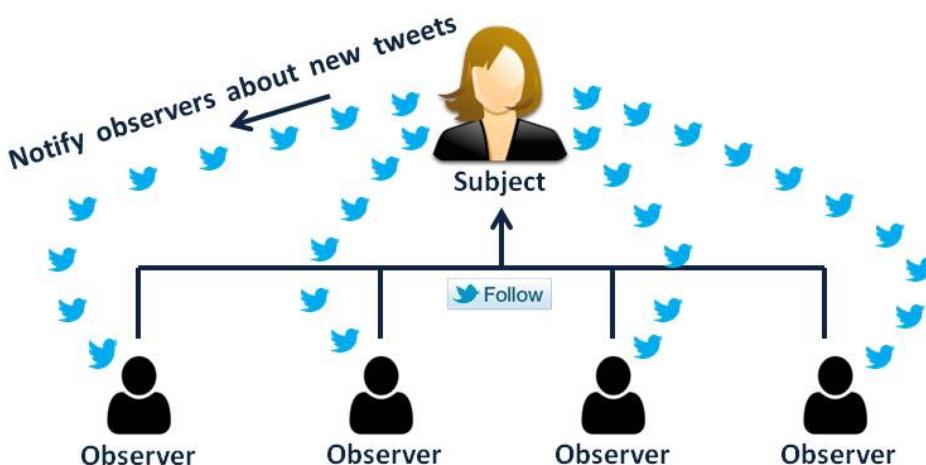
- Phải đảm bảo tính riêng tư để chính chủ mới có thể truy cập vào trạng thái của mình.

14. Behavioral pattern – Observer

Vấn đề

Khi chúng ta có một đối tượng mà chúng ta muốn nó gửi thông báo đến các đối tượng khác đã gửi đăng ký cho nó. Ví dụ nếu chúng ta đã mua đồ ở các cửa hàng lớn và để lại e-mail thì chắc chắn nếu có sản phẩm gì mới ra mắt thì sẽ có thông báo gửi đến email chúng ta. Nếu dưới gốc nhìn của server thì vấn đề đó là phải gửi một loạt các thông báo đến các e-mail đã đăng ký trước đó.

Observer Design Pattern



Hướng giải quyết

Để giải quyết nhu cầu trên chúng ta có thể dùng design pattern *Observer*. Observer cho phép các đối tượng quan sát thay đổi trong một đối tượng cụ thể mà không cần biết cụ thể về cấu trúc của nó, và giúp tăng tính tái sử dụng và phân chia logic giữa các thành phần của hệ thống.

Các thành phần của design pattern này trong c++ như sau:

Class thuần ảo subject chỉ server để nhận đăng ký và gửi tin.

Class concrete để hiện thực class thuần ảo trên: class Item.

Class thuần ảo observer chỉ các đối tượng được nhận thông tin.

Class concrete để hiện thực class thuần ảo trên: class Customer.

Hiện thực code

```
#include <iostream>
#include <vector>
using namespace std;

class Subject {
public:
    virtual void ReceiveInf() = 0;
};

class Customer : public Subject{
private:
    string name;
public:
    Customer(string st):name(st){}
    void ReceiveInf(){
        cout<<"Given mail to "<<name<<endl;
    }
};

class Observer {
    virtual void Register(Customer* c) = 0;
    virtual void DeRegister(Customer* c) = 0;
    virtual void NotifyAll() = 0;
};

class Item :public Observer{
private:
    vector<Customer*> cus;
    bool isStock;
    string name;
public:
    Item(string st,bool b):name(st), isStock(b){}
    void Register(Customer* c){
        cus.push_back(c);
    }
    void DeRegister(Customer* c){
        for(int i=0;i<cus.size();i++){
            if(cus[i]==c) cus.erase(cus.begin()+i);
        }
    }
    void NotifyAll(){
        for(int i=0;i<cus.size();i++) cus[i]->ReceiveInf();
    }
    void Check(){
        if(isStock) this->NotifyAll();
    }
};

int main(){
    Item * shoe=new Item("Adidas shoe",true);
    shoe->Register(new Customer("Brocklyn"));
    shoe->Register(new Customer("Dan"));
    shoe->Register(new Customer("Bronya"));
    shoe->Check();
}
```

- *Subject:*

Là lớp cơ sở định nghĩa các phương thức cho việc đăng ký, hủy đăng ký và thông báo cho các đối tượng Observer.

- *Customer:*

Là một đối tượng Observer cụ thể - một khách hàng.

Khi được thông báo, khách hàng nhận thông tin.

- *Observer:*

Là một giao diện hoặc lớp cơ sở định nghĩa các phương thức cho việc đăng ký, hủy đăng ký và thông báo cho các đối tượng Observer.

Item:

Là một đối tượng cụ thể mà các Observer quan tâm đến.

Có khả năng đăng ký, hủy đăng ký các Observer.

Khi có thông báo, Item gửi thông báo đến tất cả các Observer đã đăng ký.

- *Trong hàm main:*

Một đối tượng Item được tạo ra với tên là "Adidas shoe" và trạng thái là có hàng (isStock = true).

Ba đối tượng Customer được tạo ra và đăng ký vào Item "Adidas shoe".

Hàm Check() được gọi để kiểm tra trạng thái của Item, và nếu hàng có sẵn (isStock = true), thì Item gửi thông báo đến tất cả các Observer đã đăng ký thông qua hàm NotifyAll().

Trong trường hợp này, tất cả các Observer sẽ nhận được thông báo rằng "Given mail to ..." với tên của mỗi khách hàng.

Lợi ích

- Cho phép quan sát đối tượng mà không cần biết cấu trúc bên trong chúng.
- Tái sử dụng bằng cách tạo các class Observer mới.
- Cho phép các đối tượng quan sát và được quan sát độc lập với nhau.
- Cập nhật thông tin cho các đối tượng đã đăng ký.

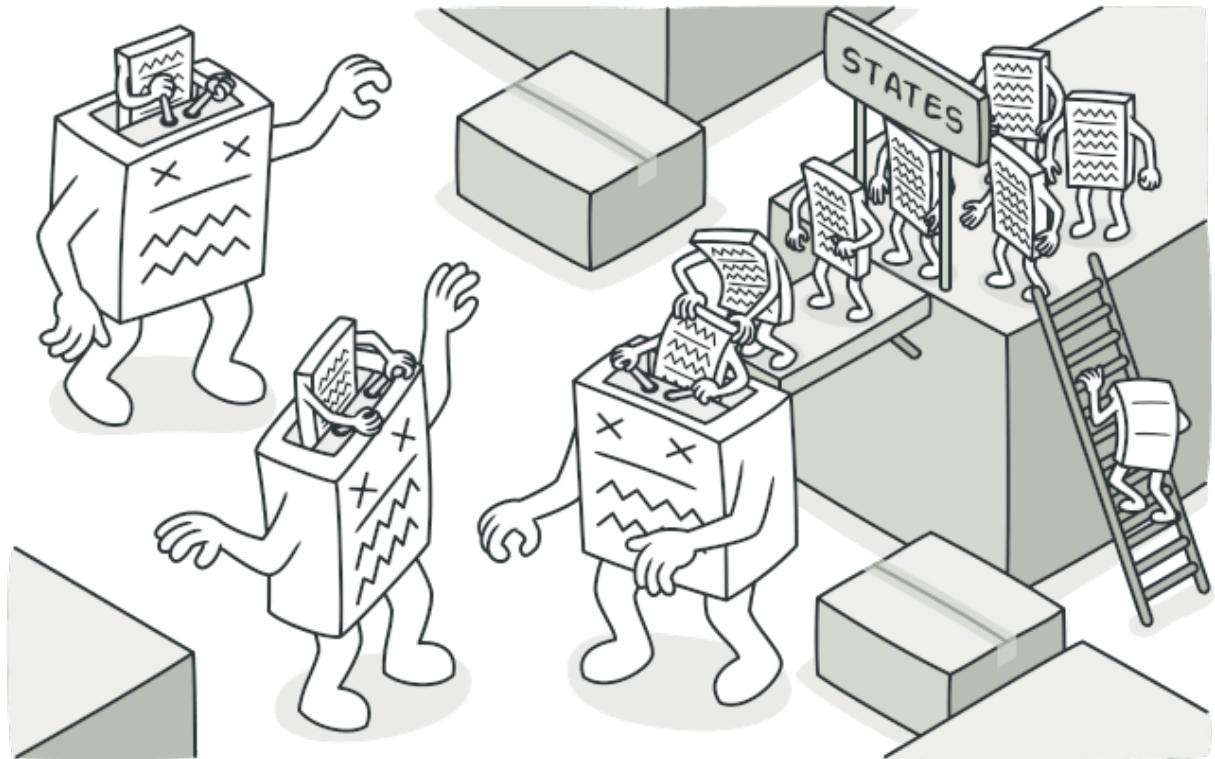
Hạn chế

- Leak memory bởi các đối tượng được quan sát không được free() đúng cách.
- Nếu hệ thống nhiều đối tượng đăng ký và thông báo thường xuyên thì hiệu suất không cao.
- Khó đọc code.

15. Behavioral pattern – State

Vấn đề

Khi một đối tượng có nhiều trạng thái, mà các trạng thái đó có cùng các kiểu hành vi nhưng cách thức thực hiện các hành vi đó lại khác nhau. Ví dụ, một công tắc có hai trạng thái On và Off thì các kiểu hành vi này có thể thấy là tương tự nhau nhưng cách thức hoạt động bên trong nó thì lại khác nhau.



Hướng giải quyết

Để giải quyết nhu cầu này, chúng ta có thể dùng design pattern *State* cho phép thay đổi hành vi khi trạng thái nội bộ của nó thay đổi.

Các thành phần của design pattern này trong c++ như sau:

Class context chỉ đỗi tượng chủ của các trạng thái: class Switch.

Class cơ sở thuần ảo State.

Các class concrete hiện thực class cơ sở trên: class On/Off.

Hiện thực code

```

class State {
public:
    virtual void On(Switch *s) = 0;
    virtual void Off(Switch *s) = 0;
};

class Switch {
private:
    State *s;
public:
    Switch (State *state):s(state){}
    void SetState(State *state){
        s=state;
    }
    void On(){
        s->On(this);
    }
    void Off(){
        s->Off(this);
    }
};

class off: public State{
public:
    void On(Switch*s) {
        cout<<"Switch return from Off to On";
        delete s;
        s->SetState(new on());
    }
    void Off(Switch*s){
        cout<<"Switch already Off";
    }
};

class on: public State{
public:
    void On(Switch*s){
        cout<<"Switch already On";
    }
    void Off(Switch*s){
        cout<<"Switch return from On to Off";
        delete s;
        s->SetState(new off());
    }
};

```

```

int main(){
    on* o= new on();
    Switch* s =new Switch(o);
    s->On();
    s->Off();
}

```

- *Class State:*

Đây là một class trừu tượng (abstract class) định nghĩa hai phương thức ảo On và Off mà mọi trạng thái cụ thể phải thực thi.

Class Switch:

Đại diện cho "context" trong mẫu State, lưu giữ một tham chiếu đến đối tượng State hiện tại và cung cấp các phương thức để thay đổi trạng thái cũng như gọi các hành vi tương ứng của trạng thái.

- *Class off và on:*

Hai class này kế thừa từ State và cài đặt các phương thức On và Off. off chứa logic chuyển đổi từ trạng thái "Tắt" sang "Bật", và on chứa logic chuyển đổi ngược lại từ "Bật" sang "Tắt".

- *Trong main():*

Tạo một đối tượng on và khởi tạo Switch với trạng thái bật.

Gọi s → On(): Do công tắc đã ở trạng thái bật, thông điệp "Switch already On" được in ra.

Gọi s → Off(): Hành vi này kích hoạt chuyển đổi trạng thái từ bật sang tắt, và thông điệp "Switch return from On to Off" được in ra.

Lợi ích

- Tách biệt rõ ràng giữa trạng thái và hành vi.
- Dễ sửa code để phù hợp với yêu cầu mới.
- Bằng cách tập trung hành vi liên quan đến từng trạng thái vào các lớp riêng biệt, State giảm sự phức tạp của việc quản lý nhiều điều kiện và chuyển đổi trạng thái trong cùng một lớp.

Hạn chế

- Tăng số lượng class phải thiết kế.
- Cần phải lý trạng thái chặt chẽ để không xảy ra lỗi logic khi các trạng thái đụng độ với nhau.

16. Behavioral pattern – Strategy

Vấn đề

Khi chúng ta có một đối tượng dùng để xử lý vấn đề và chúng ta muốn nó có thể xử lý theo nhiều chiến lược khác nhau, các chiến lược này sẽ có cùng kiểu hành vi nhưng bên trong đó thì cách xử lý sẽ khác nhau. Ở đây, chúng ta sẽ lấy ví dụ về các giải thuật Scheduling ở CPU trong hệ điều hành, ta sẽ thay đổi luân phiên các chiến lược xử lý khác nhau như FCFS, SJF, RR.

CPU Scheduling Types

Types
<ul style="list-style-type: none">• FCFS• SJF (non-preemptive)• SJF/SRTF(preemptive)• Priority• Round Robin

Hướng giải quyết

Ta sẽ áp dụng design pattern *Strategy* để giải quyết nhu cầu này, theo mô tả của vấn đề, có thể thấy design pattern này gần giống với design pattern *State* nhưng điểm khác biệt là ở *State* khi gọi hành vi chuyển trạng thái thì trạng thái bên trong đối tượng sẽ tự động thay đổi, còn ở *Strategy* chúng ta phải chủ động thay đổi chiến lược được sử dụng trước rồi mới thực hiện hành vi sau.

Các thành phần của design pattern này trong c++ như sau:

Class context chỉ đối tượng: class CPU.

Class cơ sở Strategy thuần ảo :Class Arlo.

Các class concrete để hiện thực cụ thể class cơ sở trên: class FCFS, class SJF, class RR.

Hiện thực code

```

class CPU;

class Arlo {
public:
    virtual void quit(CPU* c) = 0;
};

class FCFS :public Arlo {
public:
    void quit(CPU* c){
        cout<<"Call FCFS";
    }
};

class SJF :public Arlo {
public:
    void quit(CPU* c){
        cout<<"Call SJF";
    }
};

class RR :public Arlo {
public:
    void quit(CPU* c){
        cout<<"Call RR";
    }
};

```

```

class CPU {
private:
    vector<string> queue;
    Arlo *a;
    int cap;
    int maxcap;
public:
    CPU(int a, int b, Arlo *c):cap(a),maxcap(b),a(c){}
    void add(string st){
        queue.push_back(st);
        cap++;
    }
    void setArlo(Arlo* ar){
        a=ar;
    }
    void checkcap(){
        if(cap>=maxcap){
            a->quit(this);
            cap--;
        }
    }
};

int main(){
    FCFS *arlo1= new FCFS();
    CPU* cpu= new CPU(0,2,arlo1);
    RR *arlo2=new RR();
    cpu->setArlo(arlo2);
    cpu->add("a");
    cpu->add("b");
    cpu->checkcap();
}

```

- *Arlo (Lớp cơ sở trừu tượng):*

Lớp trừu tượng này định nghĩa một hàm thuần ảo duy nhất quit(CPU* c), mà mỗi lớp dẫn xuất phải triển khai. Hàm này nhằm mô phỏng hành động được thực hiện khi hàng đợi CPU đạt đến công suất tối đa.

- *Các lớp dẫn xuất (FCFS, SJF, RR):*

Các lớp này đại diện cho các thuật toán lập lịch khác nhau:

FCFS: Nguyên tắc đến trước, phục vụ trước

SJF: Công việc ngắn nhất trước

RR: Vòng tròn

Mỗi lớp ghi đè phương thức quit() để xuất thông báo cho biết thuật toán lập lịch nào đang được gọi. Phương thức này được cho là nơi logic lập lịch sẽ được triển khai.

- *Lớp CPU:*

Thuộc tính: vector<string> queue: Một hàng đợi chứa các chuỗi biểu diễn các tác vụ.

Arlo* a: Con trỏ đến một đối tượng Arlo, đại diện cho chiến lược lập lịch hiện tại.

int cap: Theo dõi số lượng tác vụ hiện tại trong hàng đợi.

int maxcap: Công suất tối đa của hàng đợi trước khi cần gọi hàm lập lịch.

setArlo(Arlo* ar): Cho phép thay đổi chiến lược lập lịch một cách động.

checkcap(): Kiểm tra xem số lượng tác vụ hiện tại có vượt quá công suất tối đa hay không. Nếu có, nó sẽ gọi phương thức quit() trên chiến lược lập lịch hiện tại và giảm số lượng tác vụ đi một (có vẻ như mô phỏng việc xử lý một tác vụ).

- *Hàm Main:*

Tạo một thể hiện của FCFS và đặt nó làm thuật toán lập lịch ban đầu cho CPU.
Chuyển thuật toán sang RR (Vòng tròn).

Thêm hai tác vụ vào hàng đợi của CPU.

Gọi checkcap() để kiểm tra xem công suất hàng đợi có bị vượt quá và xử lý nó một cách thích hợp.

Lợi ích

- Linh hoạt thay đổi chiến lược khi cần thiết.
- Giảm sự phụ thuộc vào chiến lược cụ thể.
- Dễ dàng mở rộng khi muốn thêm vào một chiến lược mới.
- Khi code thì chỉ cần tập trung vào lớp cụ thể mà không cần ảnh hưởng đến các lớp khác.

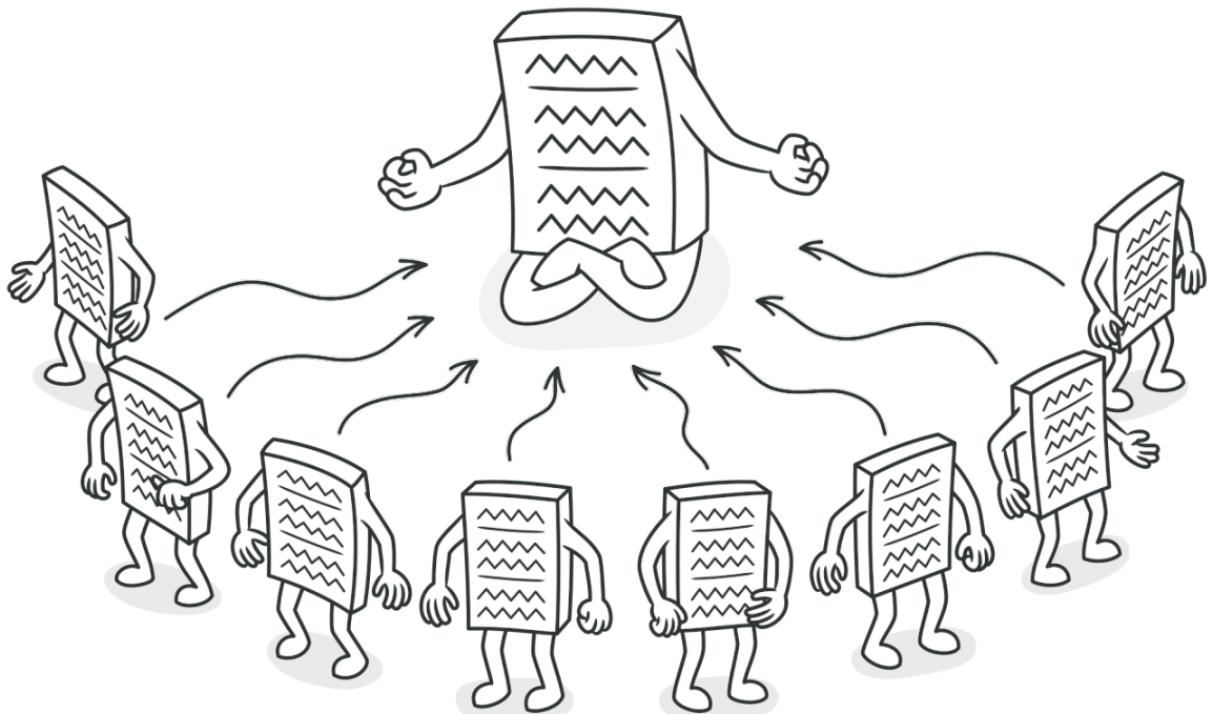
Hạn chế

- Độ phức tạp tăng khi tạo ra quá nhiều lớp.
- Phải liên tục free() khi đổi qua chiến lược mới khiến quản lý tài nguyên trở nên khó khăn.
- Việc apply nhiều chiến lược vào một hệ thống đòi hỏi hiểu biết sâu về chuyên môn để có thể đồng bộ hệ thống đó.

17. Behavioral pattern – Template Method

Vấn đề

Khi chúng ta cần thực hiện hai hay nhiều đối tượng giống nhau về kiểu hành vi cũng như cách thức hoạt động của chúng là hoàn toàn giống nhau. Giống như việc gửi mã OTP qua tin nhắn SMS hoặc e-mail đều có các hoạt động giống nhau và thực hiện theo thứ tự giống nhau.



Hướng giải quyết

Khi đó, chúng ta sẽ sử dụng design pattern *Template Method* để giải quyết vấn đề này. *Template Method* cho phép các lớp con cụ thể hóa các bước của thuật toán mà không thay đổi cấu trúc tổng thể của thuật toán.

Các thành phần cơ bản của design pattern này trong C++ như sau:

Class thuần ảo method chứa các phương thức được dùng chung: class OTP.

Các concrete class để hiện thực class trên: class SMS, class Email.

Class TemplateMethod là class viết ra giải thuật trên các phương thức trên class method.

Hiện thực code

```

class OTP {
public:
    virtual void getandsaveOTP(int num) = 0;
    virtual void sendOTP() = 0;
    virtual bool enterOTP()= 0;
};

class Email : public OTP{
private:
    int otp;
public:
    void getandsaveOTP(int num){
        int mod=pow(10,num);
        otp=rand()%mod;
    }
    void sendOTP(){
        cout<<"Send message OTP to email: "<<otp;
    }
    bool enterOTP(){
        int n;
        cout<<"Enter OTP:";
        cin>>n;
        if(n==otp) return true;
        else return false;
    }
};

class SMS : public OTP{
private:
    int otp;
public:
    void getandsaveOTP(int num){
        int mod=pow(10,num);
        otp=rand()%mod;
    }
    void sendOTP(){
        cout<<"Send message OTP to SMS: "<<otp;
    }
    bool enterOTP(){
        int n;
        cout<<"Enter OTP:";
        cin>>n;
        if(n==otp) return true;
        else return false;
    }
};

class iOTP {
private:
    OTP *o;
public:
    iOTP(OTP*otp):o(otp){}
    void getandcheckOTP(int num){
        o->getandsaveOTP(num);
        o->sendOTP();
        cout<<endl;
        if(o->enterOTP()) cout<<"successful";
        else cout<<"unsuccessful";
    }
};

int main(){
    OTP *sms=new SMS();
    iOTP *i=new iOTP(sms);
    i->getandcheckOTP(4);
}

```

- *Lớp trừu tượng OTP:*

Đây là lớp cơ sở trừu tượng có ba phương thức ảo là getandsaveOTP(int num), sendOTP(), và enterOTP(). Các lớp con phải triển khai đầy đủ các phương thức này.

- *Lớp Email:*

Triển khai cụ thể cho việc gửi OTP qua Email.

getandsaveOTP(int num): Tạo một số ngẫu nhiên làm OTP với số chữ số xác định bởi num.

sendOTP(): Hiển thị OTP qua màn hình giả định là gửi qua email.

`enterOTP()`: Cho phép người dùng nhập OTP từ bàn phím và kiểm tra nếu đúng.

- *Lớp SMS:*

Tương tự như lớp Email nhưng giả định là gửi OTP qua SMS.

`getandsaveOTP(int num)`, `sendOTP()`, `enterOTP()`: Tương tự như lớp Email nhưng thông báo gửi qua SMS.

- *Lớp iOTP:*

Lớp này sử dụng một đối tượng của lớp OTP để thực thi các phương thức liên quan đến OTP.

Constructor nhận vào một đối tượng OTP.

`getandcheckOTP(int num)`: Thực hiện quá trình tạo, gửi và kiểm tra OTP. Đầu tiên gọi `getandsaveOTP()` để tạo OTP, sau đó `sendOTP()` để gửi OTP và cuối cùng là `enterOTP()` để nhận và kiểm tra OTP từ người dùng. Nếu đúng, in ra "successful", sai thì in ra "unsuccessful".

- *Main():*

Chương trình tạo ra một instance của lớp SMS, sau đó dùng nó để khởi tạo một instance của lớp iOTP. Tiếp theo, chương trình yêu cầu iOTP xử lý một quá trình xác thực OTP bốn chữ số.

Lợi ích

- Có thể dễ dàng chỉnh sửa giải thuật.
- Lớp TemplateMethod không phải quan tâm đến chi tiết cách một đối tượng được gửi hay được tạo ra. Nó chỉ tương tác với interface Method.
- Khả năng mở rộng ra các đối tượng mới, giảm lặp lại code.

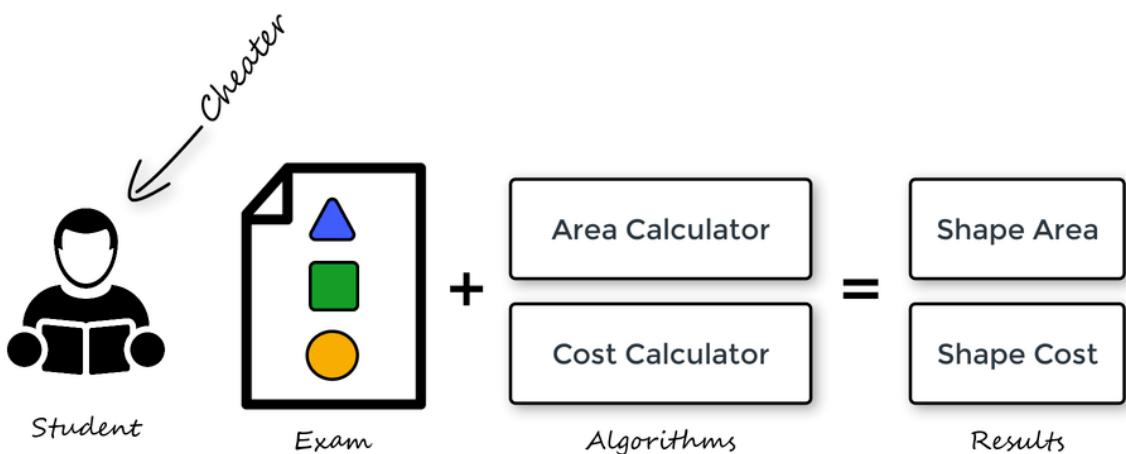
Hạn chế

- Phức tạp khi thiết kế.
- Gây overfitting.
- Khi một phần lớn các lớp con đã triển khai và phụ thuộc vào một cấu trúc cụ thể của template method, việc thay đổi cấu trúc template method này có thể ảnh hưởng cấu trúc của các lớp con đó.

18. Behavioral pattern – Visitor

Vấn đề

Khi chúng ta muốn thêm vào một hành vi hay một phương thức mới cho một đối tượng mà chúng ta không muốn thay đổi cấu trúc hiện có của lớp đối tượng đó. Ví dụ, chúng ta có một lớp đối tượng các loại hình học và chúng ta muốn tính toán các thông số trên các loại hình học đó, có thể là tính chu vi, tính diện tích và mỗi khi chúng ta muốn tính một thông số mới chúng ta chỉ cần thêm phương thức mới vào mà không cần thay đổi cấu trúc gì bên trong các lớp hình học đó cả.



Hướng giải quyết

Khi này, chúng ta có thể dùng design pattern *Visitor* để giải quyết yêu cầu trên. *Visitor* cho phép thêm các chức năng hoặc phương thức mới vào các lớp không thay đổi, mà không làm thay đổi cấu trúc của chúng.

Các thành phần cơ bản của design pattern này trong c++ như sau:

Class element thuần ảo chứa phương thức chung nhất để thực hiện các hành vi:
class Shape.

Các class concrete để hiện thực cụ thể các thông tin của các loại đối tượng:
class Circle, class Rectangle.

Class visitor trừu tượng chứa các phương thức để đại diện các hành vi cho các loại đối tượng khác nhau.

Các class concrete visitor để hiện thực class cơ sở trên: class AreaCalculator, class PerimeterCalculator.

Hiện thực code

```
class Circle;
class Rectangle;

class ShapeVisitor {
public:
    virtual void visitCircle(Circle *circle) = 0;
    virtual void visitRectangle(Rectangle *rectangle) = 0;
};

class Shape {
public:
    virtual void accept(ShapeVisitor *visitor) = 0;
};

class Circle : public Shape {
private:
    double radius;
public:
    Circle(double r) : radius(r) {}
    double getRadius() { return radius; }
    void accept(ShapeVisitor *visitor) {
        visitor->visitCircle(this);
    }
};

class Rectangle : public Shape {
private:
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double getWidth() { return width; }
    double getHeight() { return height; }
    void accept(ShapeVisitor *visitor) {
        visitor->visitRectangle(this);
    }
};

class AreaCalculator : public ShapeVisitor {
private:
    double totalArea;
public:
    AreaCalculator() : totalArea(0) {}
    void visitCircle(Circle *circle) {
        cout<<"Call area circle";
    }
    void visitRectangle(Rectangle *rectangle) {
        cout<<"Call area rectangle";
    }
};

class PerimeterCalculator : public ShapeVisitor {
private:
    double totalPerimeter;
public:
    PerimeterCalculator() : totalPerimeter(0) {}
    void visitCircle(Circle *circle) {
        cout<<"Call perimeter circle";
    }
    void visitRectangle(Rectangle *rectangle) {
        cout<<"Call perimeter rectangle";
    }
};

int main() {
    vector<Shape*> shapes;
    shapes.push_back(new Circle(5));
    shapes.push_back(new Rectangle(5,10));
    shapes.push_back(new Circle(5));

    AreaCalculator areaCalculator;
    PerimeterCalculator perimeterCalculator;

    for (Shape *shape : shapes) {
        shape->accept(&areaCalculator);
        shape->accept(&perimeterCalculator);
    }
}
```

- *ShapeVisitor:*

Là interface cho các visitor, định nghĩa hai phương thức visitCircle và visitRectangle.

- *Shape:*

Là lớp gốc cho các hình học, định nghĩa phương thức accept để chấp nhận một visitor.

- *Circle* và *Rectangle*:

Các lớp cụ thể của hình học, mỗi lớp triển khai phương thức accept để gửi visitor đến.

- *AreaCalculator* và *PerimeterCalculator*:

Là các lớp thực hiện việc tính diện tích và chu vi của các hình học. Mỗi lớp triển khai phương thức visitCircle và visitRectangle để tính toán dựa trên loại hình học.

- *Trong hàm main:*

Chúng ta tạo một danh sách các hình học gồm các hình tròn và hình chữ nhật, sau đó chấp nhận các visitor areaCalculator và perimeterCalculator để tính toán diện tích và chu vi của từng hình học.

Lợi ích

- Visitor cho phép thêm các phương thức hoặc chức năng mới mà không cần phải sửa đổi cấu trúc của lớp đang tồn tại.
- Giảm sự phụ thuộc bằng cách chuyển các chức năng vào lớp visitor.
- Dễ dàng sửa code khi đã cách biệt cách quản lý dữ liệu.

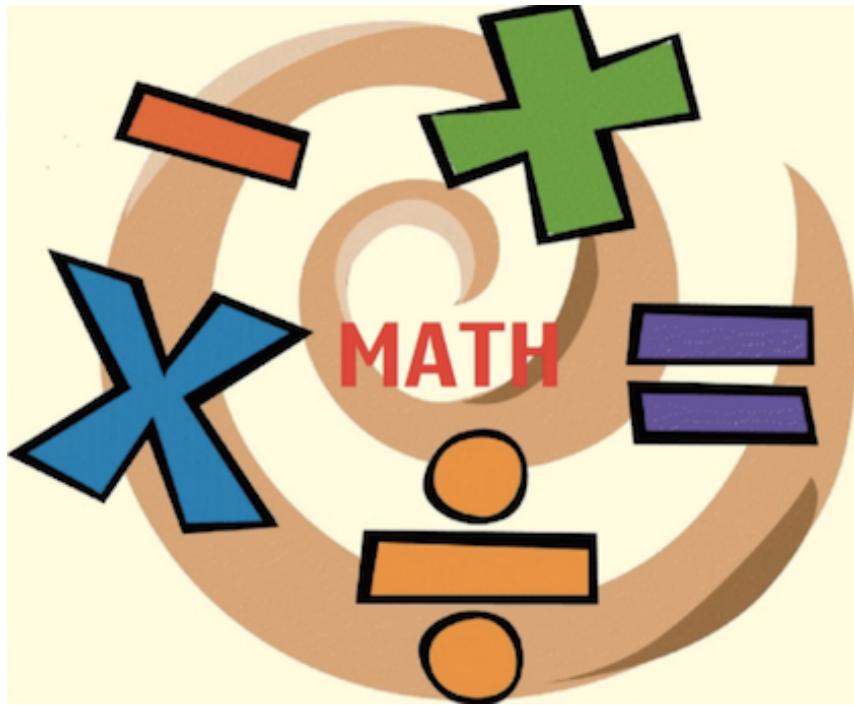
Hạn chế

- Phức tạp khi có nhiều lớp và nhiều loại visitor.
- Gây khó hiểu khi phải apply nhiều bước để tính một vấn đề.

19.Behavioral pattern – Interpreter

Vấn đề

Khi ta cần giải quyết các vấn đề liên quan đến ngôn ngữ đơn giản, như đánh giá biểu thức toán học, phân tích cú pháp, hoặc xử lý các câu lệnh trong một ngôn ngữ tự nhiên. Ví dụ, ta cần đánh giá biểu thức toán học đơn giản gồm các biến và các toán tử +, -, *, /.



Hướng giải quyết

Khi này, chúng ta sẽ sử dụng design pattern Interpreter để giải quyết vấn đề này. Interpreter được sử dụng khi bạn cần đánh giá hoặc thông dịch các câu lệnh hoặc biểu thức trong một ngôn ngữ cụ thể.

Các thành phần của design pattern này trong c++:

Abstract Expression là giao diện hoặc lớp cơ sở khai báo phương thức interpret, mà sẽ được triển khai bởi các lớp con cụ thể.

Terminal Expression là biểu thức cơ bản không thể phân chia thêm, thường đại diện cho các đối tượng cuối cùng trong ngôn ngữ.

Non-terminal Expression là biểu thức phức tạp, được xây dựng từ các biểu thức khác (có thể là Terminal hoặc Non-terminal).

Context chứa thông tin toàn cục mà các biểu thức sử dụng để đánh giá hoặc phân tích.

Client xây dựng (hoặc được cung cấp) một câu lệnh hoặc biểu thức trong ngôn ngữ cụ thể, sau đó gọi phương thức interpret để đánh giá nó.

Hiện thực code

```
#include <iostream>
#include <map>
#include <stack>
#include <string>

using namespace std;

class Expression {
public:
    virtual int interpret(map<string, int>& variables) = 0;
    virtual ~Expression() {}
};

class Number : public Expression {
private:
    int number;
public:
    Number(int num) : number(num) {}
    int interpret(map<string, int>&) { return number; }
};

class Variable : public Expression {
private:
    string name;
public:
    Variable(string varName) : name(varName) {}
    int interpret(map<string, int>& variables) {
        return variables[name];
    }
};
```

```

class Operator : public Expression {
private:
    char op;
    Expression *leftOperand;
    Expression *rightOperand;
public:
    Operator(char opr, Expression *left, Expression *right) : op(opr), leftOperand(left), rightOperand(right) {}
    ~Operator() {
        delete leftOperand;
        delete rightOperand;
    }
    int interpret(map<string, int>& variables) {
        switch(op) {
            case '+': return leftOperand->interpret(variables) + rightOperand->interpret(variables);
            case '-': return leftOperand->interpret(variables) - rightOperand->interpret(variables);
            case '*': return leftOperand->interpret(variables) * rightOperand->interpret(variables);
            case '/': return leftOperand->interpret(variables) / rightOperand->interpret(variables);
            default: return 0;
        }
    }
};

class Parser {
protected:
    stack<Expression *> expressions;
public:
    void parse(string expression) {
        for (int i = 0; i < expression.length(); ++i) {
            if (isalpha(expression[i])) {
                expressions.push(new Variable(1, expression[i])));
            } else if (isdigit(expression[i])) {
                expressions.push(new Number(expression[i] - '0'));
            } else if (expression[i] == '+' || expression[i] == '-' || expression[i] == '*' || expression[i] == '/') {
                Expression *left = expressions.top();
                expressions.pop();
                Expression *right = expressions.top();
                expressions.pop();
                expressions.push(new Operator(expression[i], left, right));
            }
        }
    }
    int evaluate(map<string, int>& variables) {
        return expressions.top()->interpret(variables);
    }
};

```

```

class SimpleParser : public Parser {
public:
    int evaluate(string expression, map<string, int>& variables) {
        parse(expression);
        return Parser::evaluate(variables);
    }
};

int main() {
    string expression = "a+b*c-d";
    SimpleParser parser;
    map<string, int> variables;
    variables["a"] = 5;
    variables["b"] = 10;
    variables["c"] = 2;
    variables["d"] = 3;
    cout << "Result: " << parser.evaluate(expression, variables) << endl;
    return 0;
}

```

- *Class Expression:*

Là interface cơ sở cho tất cả các biểu thức, khai báo phương thức `interpret` để được triển khai bởi các lớp con cụ thể.

- *Class Number:*

Biểu thức số hằng, triển khai phương thức `interpret` để trả về giá trị số của nó.

- *Class Variable:*

Biểu thức biến, triển khai phương thức `interpret` để trả về giá trị của biến từ bản đồ `variables`.

- *Class Operator:*

Biểu thức toán tử nhị phân, triển khai phương thức `interpret` để thực hiện phép toán dựa trên các toán tử +, -, *, /.

- *Class Parser:*

Lớp cơ sở để phân tích cú pháp biểu thức. Nó chứa một ngăn xếp (stack) để lưu trữ các biểu thức đã phân tích.

- *Class SimpleParser:*

Lớp cụ thể kế thừa từ `Parser`, cung cấp phương thức `evaluate` để phân tích và đánh giá biểu thức.

Lớp Parser chứa một ngăn xếp để lưu trữ các biểu thức trong quá trình phân tích cú pháp.

Phương thức parse của lớp Parser duyệt qua từng ký tự của biểu thức, tạo đối tượng biểu thức tương ứng (Number, Variable, hoặc Operator), và đẩy chúng vào ngăn xếp.

Khi gặp một toán tử, nó lấy hai biểu thức từ ngăn xếp, tạo một biểu thức toán tử mới với hai toán hạng đó, và đẩy biểu thức toán tử mới vào lại ngăn xếp.

Lớp SimpleParser kế thừa từ Parser và cung cấp phương thức evaluate để phân tích và đánh giá biểu thức.

Phương thức evaluate gọi phương thức parse để phân tích biểu thức và sau đó gọi phương thức interpret trên biểu thức đầu ngăn xếp để tính giá trị cuối cùng.

- *Trong main*

Biểu thức "a+b*c-d" được phân tích và đánh giá với các giá trị biến: a = 5, b = 10, c = 2, d = 3

Kết quả:

$$a + (b * c) - d = 5 + (10 * 2) - 3 = 5 + 20 - 3 = 22$$

Kết quả được in ra là Result: 22.

Lợi ích

- Dễ dàng thêm các quy tắc ngữ pháp mới vào hệ thống bằng cách tạo thêm các lớp biểu thức mới mà không cần thay đổi cấu trúc hiện có.
- Hệ thống có thể mở rộng để hỗ trợ các biểu thức phức tạp hơn bằng cách kế thừa và mở rộng các lớp cơ sở.
- Mỗi lớp biểu diễn một phần cụ thể của ngữ pháp, giúp cho mã nguồn dễ hiểu và bảo trì.
- Các biểu thức được tổ chức thành các đối tượng độc lập, làm cho việc quản lý mã trở nên rõ ràng và có tổ chức hơn.
- Có thể dễ dàng thay đổi hoặc cập nhật quy tắc ngữ pháp mà không ảnh hưởng đến các phần khác của hệ thống.
- Cấu trúc mã theo kiểu đối tượng giúp dễ dàng thay đổi và mở rộng chức năng mà không làm thay đổi mã hiện tại.
- Các lớp biểu thức và các thành phần khác có thể được tái sử dụng trong các dự án khác có yêu cầu tương tự.

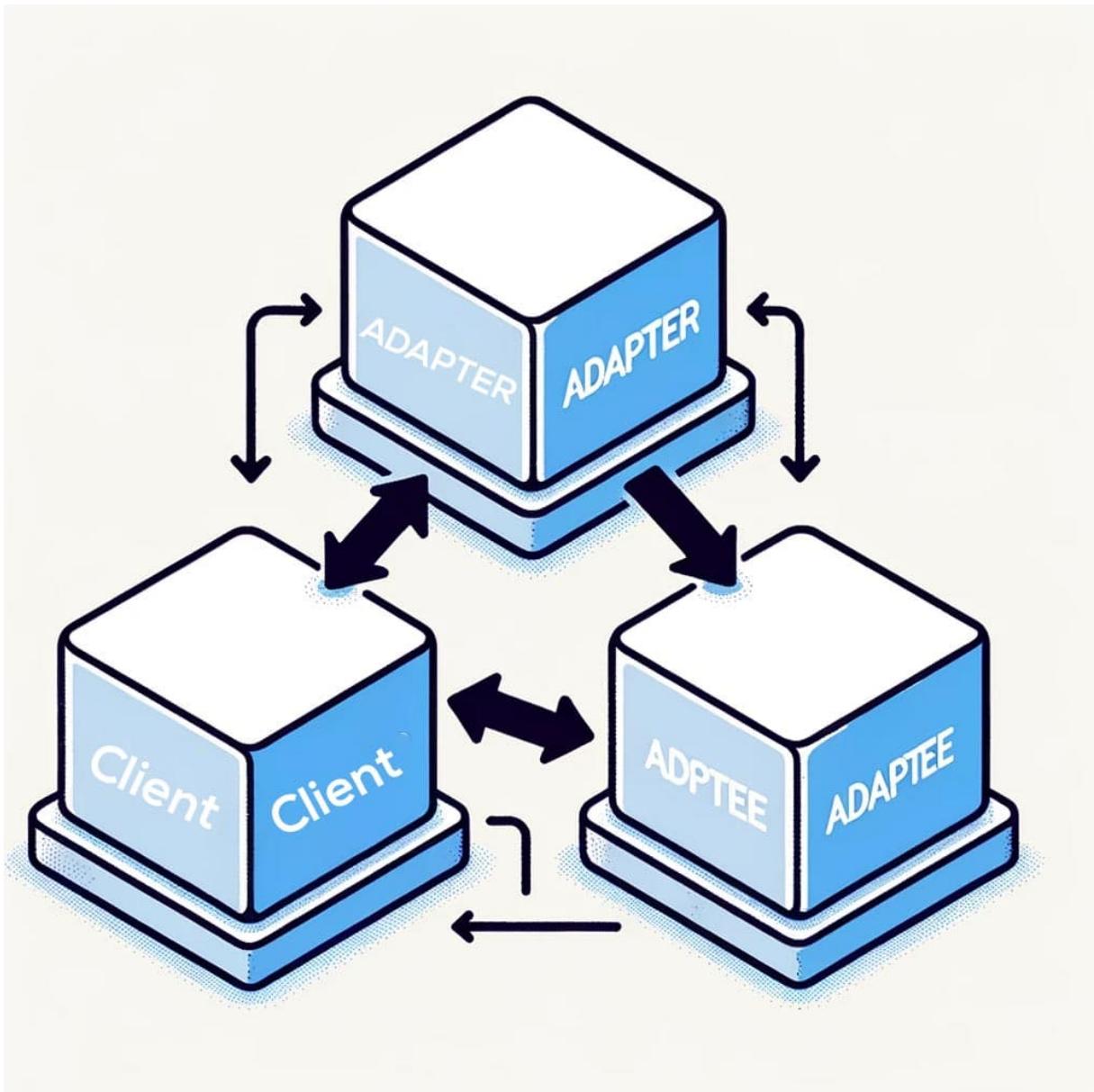
Hạn chế

- Nếu biểu thức phức tạp hoặc có số lượng lớn các biểu thức, việc tạo và quản lý nhiều đối tượng có thể làm giảm hiệu suất của hệ thống.
- Mỗi lần thực hiện biểu thức có thể dẫn đến việc gọi đệ quy nhiều lần, làm tăng chi phí xử lý.
- Khi số lượng quy tắc ngữ pháp tăng lên, số lượng lớp biểu thức cũng tăng theo, dẫn đến việc quản lý và điều hướng mã trở nên phức tạp.
- Hệ thống có thể trở nên khó hiểu đối với những người mới bắt đầu hoặc những người không quen thuộc với mẫu thiết kế này.
- Mặc dù mẫu thiết kế này dễ bảo trì trong các hệ thống nhỏ, nhưng với các hệ thống lớn và phức tạp, việc quản lý và bảo trì các lớp biểu thức có thể trở nên khó khăn.
- Các thay đổi trong một phần của ngữ pháp có thể yêu cầu thay đổi nhiều lớp khác nhau, gây ra rủi ro lỗi cao.

20. Structural pattern – Adapter

Vấn đề

Khi ta muốn kết nối hai hệ thống hoặc lớp có các giao diện không tương thích hoặc không tương đồng với nhau.



Hướng giải quyết

Để giải quyết vấn đề này ta sẽ sử dụng design pattern Adapter, Adapter hữu ích khi bạn cần kết nối giữa các thành phần, lớp, hệ thống hoặc API có giao diện không tương thích hoặc không đồng nhất với nhau.

Các thành phần của design pattern này trong c++ như sau:

Target Interface hoặc lớp mà client mong muốn tương tác với. Adapter sẽ triển khai giao diện này để client có thể sử dụng các phương thức của nó mà không cần biết về chi tiết bên dưới.

Lớp Adapter triển khai giao diện của Target và chuyển đổi yêu cầu từ client thành các yêu cầu tương ứng đối với Adaptee. Adapter này có thể được triển khai dưới dạng lớp hoặc đối tượng.

Adaptee là lớp hoặc đối tượng có sẵn và cung cấp các phương thức hoặc tính năng mà client mong muốn sử dụng, nhưng không tương thích trực tiếp với giao diện của Target. Adaptee sẽ được Adapter chuyển đổi để phù hợp với giao diện của Target.

Client là thành phần sử dụng Adapter để giao tiếp với Adaptee thông qua giao diện của Target. Client không cần biết chi tiết về cách Adapter thực hiện chuyển đổi, chỉ cần gọi các phương thức của Target như thường.

Hiện thực code

```
#include <iostream>
#include <memory>
using namespace std;
enum color{red, blue, yellow, green};
class remote{
public:
    void ContractRed(string st) const;
    void ContractBlue(string st) const;
    void ContractYellow(string st) const;
    void ContractGreen(string st) const;
};
void remote::ContractRed(string st ) const{
    cout<<"R: "<<st<<endl;
}
void remote::ContractBlue(string st ) const{
    cout<<"B: "<<st<<endl;
}
void remote::ContractYellow(string st ) const{
    cout<<"Y: "<<st<<endl;
}
void remote::ContractGreen(string st ) const{
    cout<<"G: "<<st<<endl;
}
class local{
public:
    virtual void messages(color c,string st)=0;
};
class adapter: public remote, public local{
public:
    void messages(color c, string st){
        if(c==red) ContractRed(st);
        else if(c==blue) ContractBlue(st);
        else if(c==yellow) ContractYellow(st);
        else ContractGreen(st);
    }
};
int main(){
    unique_ptr<local> displayptr(new adapter);
    displayptr->messages(red,"Hello, I am Red");
    displayptr->messages(blue,"Hello, I am Blue");
    displayptr->messages(yellow,"Hello, I am Yellow");
    displayptr->messages(green,"Hello, I am Green");
    return 0;
}
```

Adapter được sử dụng để chuyển đổi giao diện từ lớp remote sang lớp local.

- *remote:*

Là lớp hiện có với các phương thức ContractRed, ContractBlue, ContractYellow, và ContractGreen để hiển thị các thông điệp với màu tương ứng.

- *local:*

Là giao diện mà client mong muốn sử dụng để hiển thị thông điệp với màu tương ứng.

- *adapter:*

Là lớp triển khai giao diện local và kế thừa từ remote. Nó triển khai phương thức messages để chuyển đổi các yêu cầu từ giao diện local sang giao diện remote bằng cách gọi các phương thức tương ứng.

- *Trong hàm main*

Một đối tượng adapter được tạo ra và sử dụng để gọi phương thức messages với các màu khác nhau, nhưng client chỉ cần gọi phương thức messages với các mã màu red, blue, yellow, green. Adapter sẽ chuyển đổi yêu cầu này thành các cuộc gọi tương ứng tới lớp remote.

Với cách triển khai này, client không cần biết về cấu trúc bên trong của lớp remote, chỉ cần sử dụng giao diện local mà nó mong muốn. Điều này giúp giảm sự phụ thuộc của client vào cấu trúc nội bộ của lớp remote, và cho phép thay đổi nội dung của lớp remote mà không ảnh hưởng đến client.

Lợi ích

- Tính tái sử dụng cao: Adapter cho phép sử dụng các lớp hiện có mà không cần phải sửa đổi chúng, giúp tái sử dụng mã dễ dàng hơn.
- Tính linh hoạt: Adapter cung cấp một giải pháp linh hoạt cho việc tương thích giữa các giao diện không tương thích, giúp dễ dàng tích hợp các thành phần mới vào hệ thống hiện tại.
- Tăng tính mở rộng: Bằng cách sử dụng Adapter, bạn có thể mở rộng chức năng của các thành phần hiện có mà không cần thay đổi mã gốc của chúng.
- Độc lập giữa các thành phần: Client không cần biết về Adaptee, chỉ cần tương tác với Adapter thông qua giao diện chuẩn. Điều này giảm sự phụ thuộc và làm cho hệ thống dễ bảo trì hơn.

- Tương thích với các thư viện cũ: Khi bạn cần sử dụng các thư viện cũ hoặc mã cũ không tương thích với mã hiện tại, Adapter giúp bạn làm điều này mà không cần sửa đổi mã cũ.

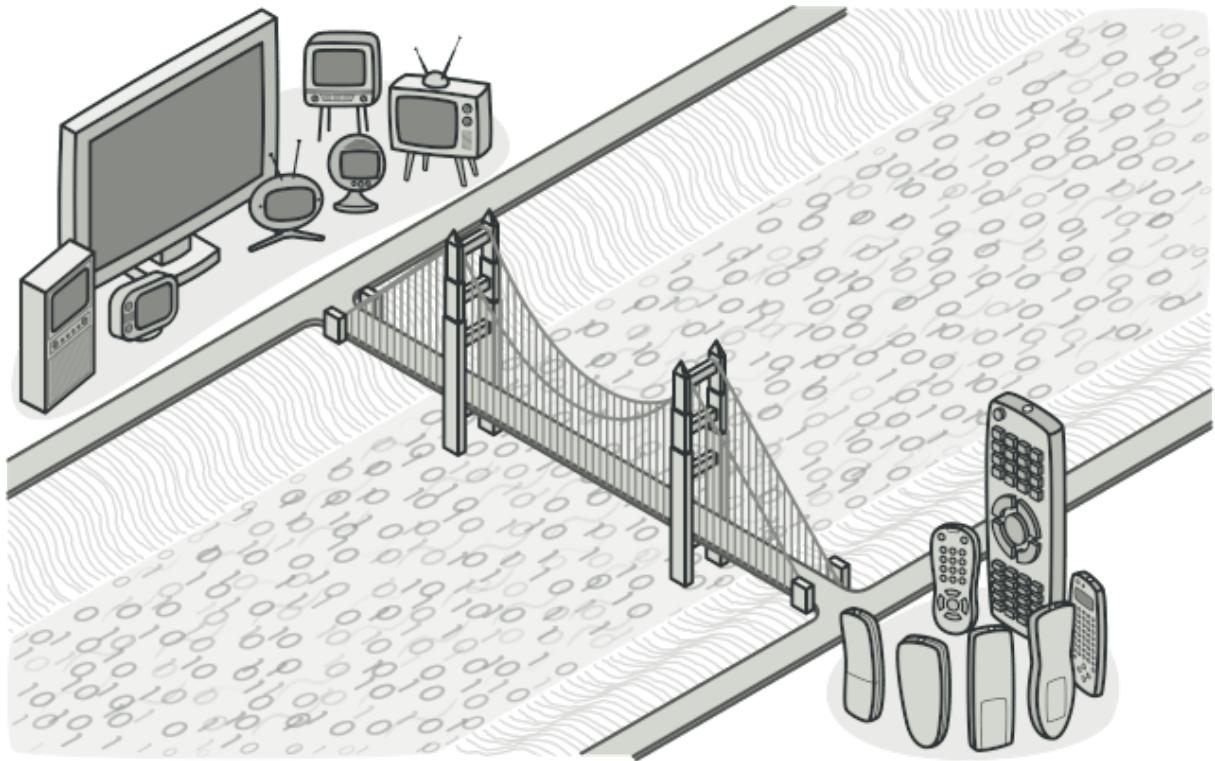
Hạn chế

- Phức tạp thêm hệ thống: Adapter thêm một lớp trung gian giữa Client và Adaptee, có thể làm tăng độ phức tạp của hệ thống, đặc biệt khi có nhiều Adapter được sử dụng.
- Hiệu suất: Việc thêm lớp trung gian có thể ảnh hưởng đến hiệu suất của hệ thống, mặc dù ảnh hưởng này thường không đáng kể.
- Khó khăn trong việc duy trì: Nếu hệ thống sử dụng quá nhiều Adapter, việc quản lý và duy trì mã có thể trở nên phức tạp và khó khăn.
- Không phù hợp cho tất cả các tình huống: Adapter không phải lúc nào cũng là giải pháp tốt nhất. Trong một số trường hợp, việc sửa đổi trực tiếp mã gốc có thể hiệu quả hơn và ít phức tạp hơn.

21. Structural pattern – Bridge

Vấn đề

Khi chúng ta muốn tách rời một hay nhiều abstraction khỏi các implementation của chúng, hoặc có thể hiểu là chúng ta muốn tách rời một số hành vi của đối tượng đó đến khi thực thi chương trình thì mới quyết định cài đặt hành vi đó như thế nào mà không ảnh hưởng đến cấu trúc của các đối tượng đó. Giống như việc chúng ta có một máy Win, máy Mac và kết nối chúng với hai máy in, và khi muốn in ra chúng ta sẽ chọn một trong hai máy in đó để thực hiện việc in.



Hướng giải quyết

Khi đó, chúng ta sẽ dùng design pattern *Bridge* để giải quyết nhu cầu trên. *Bridge* cung cấp một cách tiếp cận linh hoạt cho việc thiết kế phần mềm bằng cách tách biệt các thành phần giao diện (abstraction) và các thành phần triển khai (implementation), giúp chúng có thể thay đổi độc lập với nhau.

Các thành phần cơ bản của design pattern này trong C++ như sau:

Class abstraction thuần ảo cơ sở cho các đối tượng: class Computer.

Các class refined abstraction hiện thực class trên: class Win, class Mac.

Class implementation thuần ảo: class Printer.

Các class concrete implementation để hiện thực hành vi trên: class Epson, class HP.

Hiện thực code

```

class Win :public Computer{
private:
    Printer *p;
public:
    void print(){
        cout<<"Win ";
        p->printaction();
    }
    void setprint(Printer *p){
        this->p=p;
    }
};

class Mac :public Computer{
private:
    Printer *p;
public:
    void print(){
        cout<<"Mac ";
        p->printaction();
    }
    void setprint(Printer *p){
        this->p=p;
    }
};

class Printer{
public:
    virtual void printaction() = 0;
};

class Epson: public Printer{
public:
    void printaction(){
        cout<<"Print by Epson"<<endl;
    }
};

class HP: public Printer{
public:
    void printaction(){
        cout<<"Print by HP"<<endl;
    }
};

class Computer{
public:
    virtual void print() = 0;
    virtual void setprint(Printer *p) = 0;
};

```

- *Printer:*

Giao diện định nghĩa hành động cho việc in ấn.

- *Epson và HP:*

Các lớp cụ thể triển khai giao diện Printer, mỗi lớp cung cấp một triển khai cụ thể cho phương thức printaction().

- *Computer:*

Giao diện định nghĩa hành động in ấn và cài đặt máy in.

- *Win và Mac:*

Các lớp cụ thể của Computer, mỗi lớp triển khai phương thức print() để thực hiện hành động in ấn và setprint() để thiết lập máy in.

- *Trong hàm main:*

Chúng ta tạo một đối tượng Mac và thiết lập máy in của nó thành HP, sau đó gọi phương thức print() để in ấn thông qua máy in HP. Điều này sẽ dẫn đến việc in ra dòng "Mac Print by HP".

Lợi ích

- Bridge cho phép thay đổi cả abstraction và implementation mà không ảnh hưởng đến nhau nên dễ mở rộng.
- Khả năng phân tách abstraction và implementation giúp tái sử dụng code, dễ dàng hơn trong việc bảo trì.
- Thúc đẩy nguyên tắc thiết kế SOLID: Bridge thúc đẩy nguyên tắc Interface Segregation và Dependency Inversion trong SOLID. Nó cho phép các lớp chia thành các thành phần nhỏ và tạo ra các giao diện riêng biệt cho mỗi chức năng, giảm sự phụ thuộc giữa các lớp.

Hạn chế

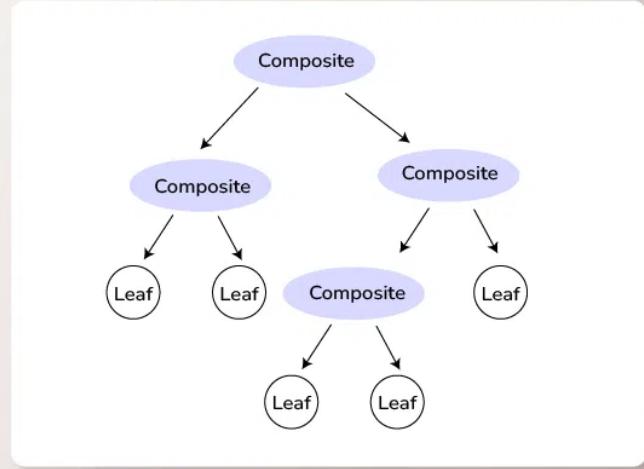
- Khó khăn trong việc triển khai và tăng độ phức tạp của code.

22. Structural pattern – Composite

Vấn đề

Khi chúng ta muốn thực hiện một hành vi trên một nhóm đối tượng dựa trên việc chỉ xử lý trên một đối tượng duy nhất. Ví dụ, chúng ta muốn tìm kiếm một từ khóa trong một folder nhưng thay vì tìm kiếm riêng lẻ từng file của folder đó, ta có thể thực hiện duy nhất trên folder đó mà không cần đến từng file.

Composite Design Pattern in Java



Hướng giải quyết

Khi đó chúng ta có thể dùng design pattern Composite để giải quyết vấn đề trên. Composite cho phép xây dựng các cấu trúc cây phức tạp từ các đối tượng đơn giản và các nhóm của chúng, và sau đó làm việc với chúng giống như với một đối tượng đơn lẻ.

Các thành phần của design pattern này trong c++ như sau:

Class component thuần ảo chứa các hàm chung của hai class dưới.

Class composite chỉ một đối tượng lớn: class Folder.

Class leaf chỉ nhóm đối tượng cần được xử lý: class File.

Hiện thực code

```

class Component {
public:
    virtual void search(string key) = 0;
};

class File : public Component {
private:
    string name;
public:
    File(string st ):name(st){}
    void search(string key){
        cout<<"Search "<<key<<" in "<<name<<endl;
    }
};

class Folder: public Component{
private:
    string name;
    vector<File*> f;
public:
    Folder(string st): name(st){}
    void add(File *f){
        this->f.push_back(f);
    }
    void search(string key){
        for(File* ite:f){
            ite->search(key);
        }
    }
};

int main(){
    Folder* fol=new Folder("Tree");
    fol->add(new File("leaf 1"));
    fol->add(new File("leaf 2"));
    fol->add(new File("leaf 3"));
    fol->search("Money");
}

```

- *Component:*

Là một lớp trừu tượng hoặc giao diện định nghĩa các phương thức chung cho tất cả các thành phần, bao gồm cả thành phần cụ thể và nhóm thành phần. Trong trường hợp này, Component được đại diện bởi lớp Component với phương thức search().

- *File:*

Là một lớp cụ thể kế thừa từ Component, đại diện cho các tệp trong hệ thống tệp. Mỗi File có một tên định danh.

- *Folder:*

Là một lớp cụ thể khác kế thừa từ Component, đại diện cho các thư mục trong hệ thống tệp. Nó có một tên và một vector các con trỏ đến các File hoặc Folder khác.

- *Trong hàm main:*

Một thư mục "Tree" được tạo và ba tệp ("leaf 1", "leaf 2", "leaf 3") được thêm vào thư mục này. Sau đó, hàm search() được gọi trên thư mục với một từ khóa "Money", và nó sẽ tìm kiếm từ khóa này trong tất cả các tệp và thư mục con bên trong thư mục "Tree".

Lợi ích

- Cho phép bạn xây dựng cấu trúc phức tạp từ các đối tượng đơn giản và nhóm của chúng, giúp quản lý cấu trúc dễ dàng hơn.
- Xử lý đồng nhất các cá thể trong nhóm nên việc thêm thành phần, phương thức, hành vi mới trở nên dễ dàng.

Hạn chế

- Hiệu suất không cao.
- Không thể xử lý riêng lẻ một cá thể đặc biệt nào đó trong nhóm. Trong một số trường hợp, việc xác định loại đối tượng cụ thể trong nhóm có thể trở nên phức tạp, đặc biệt khi một số phép toán phụ thuộc vào loại đối tượng cụ thể.

23. Structural pattern – Facade

Vấn đề

Khi chúng ta cần làm việc với một nhóm các đối tượng mà các đối tượng đó có các cách khởi tạo khác nhau và các hành vi làm việc giữa chúng lại rất phức tạp, yêu cầu chúng ta phải tạo ra một giao diện khác bên ngoài để đảm bảo người dùng có thể hiểu và làm việc dễ dàng. Ví dụ, để cho vấn đề phức tạp ta sẽ thực hiện một hệ thống quản lý tài khoản ngân hàng của người dùng để có thể bảo vệ người dùng khi rút tiền.

QUY TRÌNH



Đặt CCCD lên thiết bị đọc, thiết bị này thu nhận ảnh chân dung, vân tay của khách hàng

Thiết bị đọc phân tích dữ liệu và đối sánh dữ liệu sinh trắc đã lưu trên chip thẻ CCCD với thông tin đã lưu trữ tại hệ thống ngân hàng

Khi 2 dữ liệu trùng khớp, các bước tiếp theo được thực hiện để rút tiền



Phương thức bảo mật rất chặt chẽ, khả năng bảo mật cao hơn thẻ ATM (ngoài tuân thủ quy trình bảo mật ngân hàng sẽ có thêm lớp bảo mật sử dụng qua thẻ CCCD gắn chip)



Tiết kiệm thời gian và không cần mang theo thẻ ATM

Hướng giải quyết

Chúng ta sẽ sử dụng design pattern *Facade* để hiện thực vấn đề trên. Facade cung cấp một giao diện đơn giản hóa cho một hệ thống phức tạp, một thư viện, hoặc một framework. Nó đóng vai trò như một mặt tiền hoặc mặt ngoài cho một bộ mã rộng lớn hơn, giúp cho việc tương tác với hệ thống đó dễ dàng hơn cho các nhà phát triển sử dụng nó.

Các thành phần cơ bản của design pattern này trong c++ như sau:

Các class subsystem: class Account, class Code, Class UserWallet, class Notify.

Class facade là giao diện để người dùng thấy được phần nổi của hệ thống: class Wallet.

Hiện thực code

```

class Account{
private:
    string name;
public:
    Account(string st):name(st){}
    void checkaccount(string st){
        if(name==st) cout<<"Hi "<<name<<endl;
        else throw "Error";
    }
};

class Code{
private:
    int code;
public:
    Code(int x):code(x){}
    void checkcode(int x){
        if(x==code) cout<<"Pass succesful"<<endl;
        else throw "Error";
    }
};

class UserWallet{
private:
    int credit;
public:
    UserWallet(int x): credit(x){}
    void incrcredit(int x){
        credit+=x;
    }
    void decrcredit(int x){
        if(x>credit) throw "Error";
        credit-=x;
    }
};

class Notify{
public:
    void notifyincr(int x){
        cout<<"Wallet increase "<<x<<endl;
    }
    void notifydecr(int x){
        cout<<"Wallet decrease "<<x<<endl;
    }
};

class Wallet {
private:
    Account* a;
    Code* c;
    UserWallet* w;
    Notify* n;
public:
    Wallet(string st, int x, int credit){
        a=new Account(st);
        c=new Code(x);
        w=new UserWallet(credit);
        n=new Notify();
    }
    void SendMoney(string st, int x, int money){
        a->checkaccount(st);
        c->checkcode(x);
        w->incrcredit(money);
        n->notifyincr(money);
    }
    void GetMoney(string st, int x, int money){
        a->checkaccount(st);
        c->checkcode(x);
        w->decrcredit(money);
        n->notifydecr(money);
    }
};

int main(){
    Wallet *w=new Wallet("No name",1111,100000);
    w->GetMoney("No name",1111,50000);
    w->GetMoney("No name",1111,60000);
}

```

- *Account:*

Lớp này biểu diễn một tài khoản người dùng với một tên duy nhất. Phương thức checkaccount kiểm tra xem tên được truyền vào có khớp với tên của tài khoản hay không.

- *Code:*

Lớp này biểu diễn một mã xác thực, thường được sử dụng để đảm bảo an toàn trong các giao dịch. Phương thức checkcode kiểm tra xem mã được truyền vào có khớp với mã xác thực hay không.

- *UserWallet:*

Lớp này đại diện cho ví của người dùng, nơi lưu trữ số dư. Phương thức incrcredit tăng số dư trong ví, trong khi decrcredit giảm số dư. Nếu số tiền muốn giảm lớn hơn số dư hiện có, ném ra một ngoại lệ.

- *Notify:*

Lớp này đại diện cho việc thông báo khi có giao dịch thành công hoặc không thành công.

- *Wallet:*

Lớp này là giao diện chính cho việc thao tác với tài khoản, mã xác thực và ví của người dùng. Phương thức SendMoney được sử dụng để gửi tiền và GetMoney được sử dụng để nhận tiền. Trước khi thực hiện giao dịch, nó kiểm tra tài khoản và mã xác thực. Nếu thông qua, nó thực hiện giao dịch và thông báo kết quả.

- *Trong hàm main():*

Một ví mới được tạo với tên "No name", mã xác thực là 1111 và số dư ban đầu là 100.000. Sau đó, một số tiền được rút ra từ ví, với một số tiền vượt quá số dư hiện có, dẫn đến việc ném ra ngoại lệ.

Lợi ích

- Facade cung cấp một giao diện đơn giản để tương tác với một hệ thống phức tạp, giúp người dùng không cần phải hiểu chi tiết về các lớp và phương thức bên trong hệ thống.
- Client code chỉ giao tiếp qua facade nên ít phụ thuộc vào hệ thống phức tạp sâu bên trong cũng như ít bị ảnh hưởng khi hệ thống bên trong có sự thay đổi.
- Tính tái sử dụng.

Hạn chế

- Facade có thể trở thành một đối tượng "god object" nếu quá nhiều chức năng được gom vào một Facade duy nhất, làm cho giao diện này trở nên quá phức tạp và khó quản lý.
- Kém linh hoạt và giảm khả năng mở rộng khi quá nhiều chi tiết bên trong hệ thống bị giấu đi.

24. Structural pattern – Flyweight

Vấn đề

Khi chúng ta có một chương trình mà trong chương trình đó sử dụng một lượng lớn các đối tượng từ các class, và điều đó sẽ gây ra hai vấn đề, một là thời gian tạo lại các đối tượng đó, hai là bộ nhớ để lưu trữ lượng lớn các đối tượng đó. Ví dụ, chúng ta cần tạo ra một hệ thống quản lý các chiến sĩ với một số lượng rất lớn các chiến sĩ.



Hướng giải quyết

Khi đó chúng ta sẽ sử dụng design pattern Flyweight để tái sử dụng lại các đối tượng có hành vi và thuộc tính giống nhau, thông qua việc đó ta có thể giảm thiểu thời gian tạo ra các đối tượng cũng như không gian dùng để lưu trữ chúng. Lỡ như, giữa các đối tượng có sự khác nhau nhỏ thì Flyweight cũng sẽ cung cấp phương pháp để cập nhật lại sự khác nhau đó tại thời điểm được thăm, còn thuộc tính giống nhau sẽ được tái sử dụng hoàn toàn.

Các thành phần của design pattern này trong c++ như sau:

Flyweight interface: chứa các hàm hành vi để liên kết giữa các flyweight object và các context.

Class Concrete Flyweight: có thể có một hoặc nhiều class này, trong class này sẽ chứa các trạng thái nội tại (trạng thái giống).

Class Flyweight factory: là thành phần quan trọng nhất của design pattern này, class này chứa cấu trúc dữ liệu như map để lưu lại các đối tượng đã được khởi tạo

trước đó, nên nếu như chúng ta cần khởi tạo lại đối tượng này thì sẽ tham chiếu đến cấu trúc dữ liệu này mà không cần tạo lại đối tượng khác.

Class context: chứa các trạng thái bên ngoài (trạng thái khác).

Hiện thực code

```
#include <iostream>
#include <unordered_map>
#include <string>
using namespace std;
class iSoldier {
public:
    virtual void display(string id, int star, int weight, int height) = 0;
};

class Soldier : public iSoldier {
private:
    std::string color;

public:
    Soldier(const std::string& color) : color(color) {}

    void display(string id, int star, int weight, int height) {
        std::cout << "Display this soldier wear " << color << " with id: "
             << id << ", " << star << " star(s), with weight " << weight
             << " and height " << height << endl;
    }
};

class SoldierFactory {
private:
    std::unordered_map<std::string, iSoldier*> soldiers;

public:
    ~SoldierFactory() {
        for (auto& pair : soldiers) {
            delete pair.second;
        }
    }

    iSoldier* getSoldier(const string& color) {
        if (soldiers.find(color) == soldiers.end()) {
            soldiers[color] = new Soldier(color);
        }
        return soldiers[color];
    }
};

int main() {
    SoldierFactory fac;

    iSoldier* brownsoldier = fac.getSoldier("red");
    brownsoldier->display("MTX 65", 20, 30, 40);

    iSoldier* greensoldier = fac.getSoldier("blue");
    greensoldier->display("MMX 70", 60, 70, 80);

    iSoldier* brownsoldier2 = fac.getSoldier("red");
    brownsoldier2->display("MTX 70", 25, 35, 45);

    return 0;
}
```

- *Lớp iSoldier:*

Đây là một interface (giao diện) cho các đối tượng Soldier. Nó định nghĩa phương thức display mà các lớp con phải triển khai.

- *Lớp Soldier:*

Lớp này triển khai giao diện iSoldier và đại diện cho một Soldier cụ thể với màu sắc đặc trưng. Nó có một thuộc tính color và triển khai phương thức display để hiển thị thông tin của Soldier.

- *Lớp SoldierFactory:*

Lớp này đóng vai trò là một factory (nhà máy) để quản lý và chia sẻ các đối tượng Soldier. Nó sử dụng một unordered_map để lưu trữ các đối tượng Soldier theo

màu sắc của chúng. Phương thức getSoldier sẽ trả về một đối tượng Soldier với màu sắc được chỉ định, nếu đối tượng đó chưa tồn tại, nó sẽ tạo mới và lưu trữ vào map.

- *Hàm main:*

Trong hàm main, chúng ta tạo một đối tượng SoldierFactory để quản lý các Soldier. Sau đó, chúng ta lấy các Soldier với màu sắc cụ thể từ factory và hiển thị thông tin của chúng. Lưu ý rằng đối tượng Soldier màu đỏ chỉ được tạo ra một lần và được sử dụng lại khi cần.

Lợi ích

- Tiết kiệm bộ nhớ: Giảm đáng kể lượng bộ nhớ cần thiết khi có nhiều đối tượng tương tự.
- Hiệu suất: Có thể cải thiện hiệu suất do việc giảm số lượng đối tượng được tạo ra.

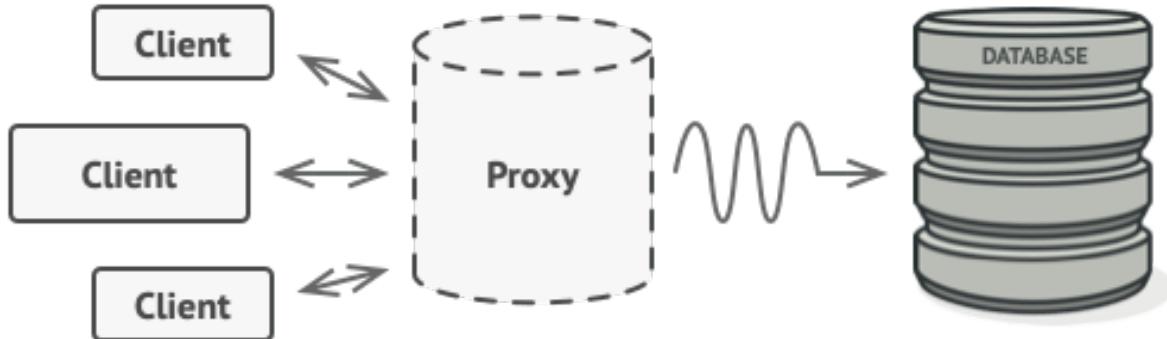
Hạn chế

- Phức tạp hơn trong quản lý: Việc triển khai mẫu Flyweight có thể phức tạp hơn, đặc biệt là khi cần đảm bảo rằng trạng thái chia sẻ và trạng thái không chia sẻ được quản lý đúng cách.
- Không phù hợp với mọi tình huống: Không phải mọi tình huống đều phù hợp để sử dụng mẫu Flyweight, đặc biệt là khi các đối tượng không có nhiều điểm chung.

25. Structural pattern – Proxy

Vấn đề

Khi chúng ta cần một đối tượng để ủy quyền hoặc đại diện cho một đối tượng khác. Có thể là dùng để lưu tạm giá trị cho đối tượng khác, là đối tượng bảo vệ cho đối tượng khác. Trong ví dụ này, chúng ta sẽ tạo một hệ thống để truy cập tài nguyên từ xa (giả sử là một hình ảnh). Hệ thống sẽ sử dụng một Proxy để kiểm soát truy cập vào hình ảnh này. Hình ảnh chỉ được tải xuống khi thật sự cần thiết, giúp tiết kiệm băng thông và tài nguyên.



Hướng giải quyết

Khi đó, chúng ta sẽ dùng design pattern Proxy để giải quyết vấn đề trên. Proxy cho phép tạo ra một đối tượng thay thế hoặc bộ điều khiển truy cập cho một đối tượng khác. Proxy có thể kiểm soát truy cập đến đối tượng thực, quản lý việc tạo hoặc tiêu hủy đối tượng, và thực hiện các thao tác khác như kiểm tra, log, hoặc caching.

Trong một số trường hợp, có thể có các loại Proxy khác nhau như:

- Remote Proxy: Đại diện cho một đối tượng trên một phương tiện truyền thông từ xa.
- Virtual Proxy: Tạo đối tượng thực sự chỉ khi cần thiết.
- Protection Proxy: Kiểm soát quyền truy cập đến đối tượng RealSubject.

Các thành phần của design pattern này trong c++ như sau:

Subject Interface hoặc lớp trừu tượng định nghĩa các phương thức mà Proxy sẽ triển khai. Subject thường là một interface chung cho cả RealSubject và Proxy để RealSubject và Proxy có thể được sử dụng tương đồng.

Proxy là một lớp trung gian hoặc thay thế cho RealSubject. Proxy có cùng interface với RealSubject để client có thể sử dụng Proxy một cách không phân biệt. Proxy giữ một tham chiếu đến RealSubject và kiểm soát quyền truy cập và hành vi của RealSubject. Proxy cung cấp một số tính năng bổ sung như caching, kiểm soát truy cập, lazy initialization, logging, etc.

Real subject là đối tượng thực sự mà Proxy đại diện. RealSubject triển khai các phương thức được định nghĩa trong Subject.

Hiện thực code

```

#include <iostream>
#include <string>
using namespace std;

class Image {
public:
    virtual void display() = 0;
};

class RealImage : public Image {
private:
    string filename;

    void loadFromDisk() {
        cout << "Loading " << filename << endl;
    }

public:
    RealImage(const string& filename) : filename(filename) {
        loadFromDisk();
    }

    void display() override {
        cout << "Displaying " << filename << endl;
    }
};

class ProxyImage : public Image {
private:
    string filename;
    RealImage* realImage;

public:
    ProxyImage(const string& filename) : filename(filename), realImage(nullptr) {}

    ~ProxyImage() {
        delete realImage;
    }

    void display() override {
        if (!realImage) {
            realImage = new RealImage(filename);
        }
        realImage->display();
    }
};

```

```

int main() {
    Image* image1 = new ProxyImage("test_image_1.jpg");
    Image* image2 = new ProxyImage("test_image_2.jpg");

    cout << "Calling display on image1 first time:" << endl;
    image1->display();

    cout << "Calling display on image1 second time:" << endl;
    image1->display();

    cout << "Calling display on image2 first time:" << endl;
    image2->display();

    delete image1;
    delete image2;

    return 0;
}

```

- *Giao diện Image:*

Đây là giao diện chung cho hình ảnh. Nó có một phương thức thuận ảo display mà các lớp con phải triển khai.

- *Lớp RealImage:*

Đây là lớp đại diện cho hình ảnh thực sự. Khi đối tượng RealImage được tạo, nó sẽ tải hình ảnh từ đĩa (giả lập bằng việc in ra dòng chữ).

- *Lớp ProxyImage:*

Đây là lớp Proxy, nó kiểm soát truy cập vào RealImage. Hình ảnh thực sự chỉ được tải xuống khi phương thức display được gọi lần đầu tiên.

- *Hàm main:*

Trong hàm main, chúng ta tạo hai đối tượng ProxyImage. Hình ảnh thực sự chỉ được tải xuống khi phương thức display được gọi lần đầu tiên cho mỗi hình ảnh. Việc này giúp tiết kiệm tài nguyên và băng thông.

Lợi ích

- Tiết kiệm tài nguyên: Chỉ tải tài nguyên khi cần thiết.
- Kiểm soát truy cập: Proxy có thể kiểm soát và quản lý quyền truy cập vào tài nguyên thực.
- Giảm độ phức tạp: Proxy có thể thêm các chức năng bổ sung như caching, logging mà không thay đổi lớp gốc.

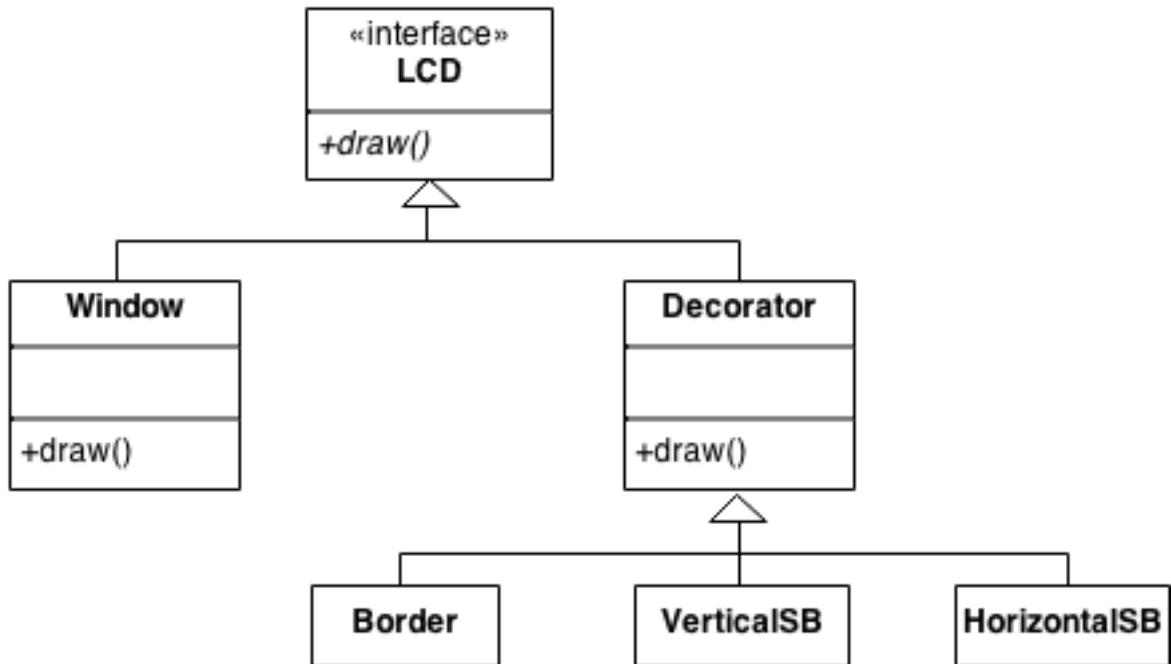
Hạn chế

- Chi phí bổ sung: Có thể thêm chi phí về mặt bộ nhớ và hiệu năng do phải duy trì các đối tượng proxy.
- Phức tạp hơn: Mã nguồn trở nên phức tạp hơn do sự xuất hiện của các lớp Proxy.

26. Structural pattern – Decorator

Vấn đề

Khi chúng ta muốn thêm tính năng mới vào đối tượng mà không muốn ảnh hưởng đến thành phần bên trong của thuộc tính này. Trong ví dụ này, chúng ta sẽ thực hiện một bộ lọc cho một luồng dữ liệu



Hướng giải quyết

Để thực hiện vấn đề này, chúng ta sẽ sử dụng design pattern Decorator.

Decorator cho phép mở rộng chức năng của các đối tượng bằng cách đóng gói chúng bên trong các lớp bao bọc (decorators).

Các thành phần của design pattern này trong c++ như sau:

Component Interface hoặc lớp trừu tượng định nghĩa các phương thức cơ bản mà cả ConcreteComponent và Decorator sẽ triển khai. Thành phần có thể là một lớp hoặc một interface.

Các Concrete component là lớp cơ bản hoặc thành phần cụ thể mà Decorator mở rộng. Nó triển khai các phương thức được định nghĩa trong Component.

Decorator Interface hoặc lớp trừu tượng, cũng triển khai Component, nhưng giữ một tham chiếu đến một đối tượng Component. Decorator có thể thêm các chức năng mới thông qua việc thêm các thành phần khác hoặc thay đổi hành vi của thành phần gốc mà nó đóng gói.

Các Concrete decorator chứa các component properties là các lớp cụ thể của Decorator, mở rộng hoặc thêm chức năng cho ConcreteComponent. Chúng triển khai phương thức của Decorator và thêm các chức năng mới hoặc thay đổi hành vi của Component gốc.

Hiện thực code

```
#include <iostream>
#include <string>
using namespace std;

class Stream {
public:
    virtual void write(string data) = 0;
    virtual ~Stream() = default;
};

class FileStream : public Stream {
public:
    void write(string data) override {
        cout << "Writing data to file: " << data << endl;
    }
};

class StreamDecorator : public Stream {
protected:
    Stream* wrappee;
public:
    StreamDecorator(Stream* stream) : wrappee(stream) {}
    virtual ~StreamDecorator() {
        delete wrappee;
    }
};

class CompressedStream : public StreamDecorator {
public:
    CompressedStream(Stream* stream) : StreamDecorator(stream) {}

    void write(string data) override {
        data = compress(data);
        wrappee->write(data);
    }

    string compress(string data) {
        return "compressed(" + data + ")";
    }
};
```

```

class EncryptedStream : public StreamDecorator {
public:
    EncryptedStream(Stream* stream) : StreamDecorator(stream) {}

    void write(string data) override {
        data = encrypt(data);
        wrappee->write(data);
    }

    string encrypt(string data) {
        return "encrypted(" + data + ")";
    }
};

int main() {
    Stream* fileStream = new FileStream();
    fileStream->write("Hello, World!");

    Stream* compressedStream = new CompressedStream(new FileStream());
    compressedStream->write("Hello, World!");

    Stream* encryptedStream = new EncryptedStream(new FileStream());
    encryptedStream->write("Hello, World!");

    Stream* compressedEncryptedStream = new CompressedStream(new EncryptedStream(new FileStream()));
    compressedEncryptedStream->write("Hello, World!");
}

```

- *Giao diện Stream:*

Đây là giao diện chung cho các luồng dữ liệu, với một phương thức thuận tiện để ghi dữ liệu.

- *Lớp FileStream:*

Lớp này triển khai giao diện Stream và thực hiện việc ghi dữ liệu vào file (giả lập bằng cách in ra màn hình).

- *Lớp cơ sở StreamDecorator:*

Đây là lớp cơ sở cho các decorator, nó giữ một tham chiếu đến đối tượng Stream thực sự và chuyển tiếp các lời gọi phương thức đến đối tượng này.

- *Lớp CompressedStream:*

Lớp này thêm tính năng nén dữ liệu trước khi ghi dữ liệu.

- *Lớp EncryptedStream:*

Lớp này thêm tính năng mã hóa dữ liệu trước khi ghi dữ liệu.

- *Hàm main:*

Trong hàm main, chúng ta tạo ra các đối tượng FileStream cơ bản và các đối tượng FileStream được trang trí với các tính năng nén và mã hóa. Chúng ta thấy rằng các lớp decorator có thể được kết hợp với nhau một cách linh hoạt để tạo ra các luồng dữ liệu phức tạp hơn.

Lợi ích

- Mở rộng chức năng: Cho phép thêm các tính năng mới vào đối tượng mà không cần thay đổi mã nguồn của đối tượng gốc.
- Linh hoạt: Các decorator có thể được kết hợp một cách linh hoạt và được thêm hoặc bớt một cách dễ dàng.
- Nguyên tắc mở-đóng: Mẫu thiết kế này tuân theo nguyên tắc mở-đóng, nghĩa là các lớp có thể mở rộng mà không cần sửa đổi mã nguồn của chúng.

Hạn chế

- Phức tạp: Số lượng các lớp tăng lên đáng kể, làm cho hệ thống trở nên phức tạp hơn.
- Khó bảo trì: Việc quản lý và bảo trì nhiều lớp decorator có thể trở nên khó khăn.

Tài liệu tham khảo

<https://realpython.com/python-functional-programming/>

<https://www.youtube.com/@yuhlaptrinhvien2223>