ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC BÁCH KHOA

BÁO CÁO BÀI TẬP LỚN
MÔN LẬP TRÌNH NÂNG CAO

GVHD: TS Trương Tuấn Anh

Lớp: TN01

Họ và tên: Võ Phương Minh Nhật

MSSV: 2212413

TPHCM, 04-2024

Mục lục

I Lập trình hàm trong ngôn ngữ lập trình Python 3

1 Giới thiệu 3
1.1 Chứng minh hình thức . . . 4
1.2 Tính mô đun . . . 5
1.3 Dễ gỡ lỗi và kiểm tra . . . 5
1.4 Khả năng kết hợp . . . 5

2 Iterators 6
2.1 Các kiểu dữ liệu hỗ trợ Iterator . . . 7

3 Generator expressions và list comprehensions 8

4 Generators 10
4.1 Truyền giá trị vào generator . . . 11

5 Built-in functions 13

6 The itertools module 15
6.1 Tạo các iterator mới . . . 15
6.2 Gọi hàm trên các phần tử . . . 16
6.3 Lựa chọn phần tử . . . 16
6.4 Hàm tổ hợp . . . 17
6.5 Nhóm các phần tử . . . 18

7 Mô-đun functools 19
7.1 Mô-đun operator . . . 20

8 Hàm nhỏ và biểu thức lambda 21

II Các Design pattern trong OOP 23

1 Giới thiệu về Design pattern 23
1.1 Design pattern là gì? . . . 23
1.2 Lịch sử của các pattern . . . 23
1.3 Vì sao cần học design pattern . . . 24
1.4 Các phê bình về design pattern . . . 24
1.5 Phân loại các pattern . . . 25

2 Creational Design Patterns 26
2.1 Factory Method . . . 26
2.2 Abstract Factory . . . 29
2.3 Builder . . . 32
2.4 Prototype . . . 34
2.5 Singleton . . . 37

3 Structural Design Patterns 40
3.1 Adapter . . . 40

3.2 Bridge 42
3.3 Composite 45
3.4 Decorator 47
3.5 Facade 50
3.6 Flyweight 53
3.7 Proxy 55

4 Behavioral Design Patterns **58**
4.1 Chain of Responsibility 58
4.2 Command 61
4.3 Iterator 64
4.4 Mediator 66
4.5 Memento 69
4.6 Observer 72
4.7 State 75
4.8 Strategy 78
4.9 Template method 81
4.10 Visitor 83

Phần I

Lập trình hàm trong ngôn ngữ lập trình Python

1 Giới thiệu

Các ngôn ngữ lập trình hỗ trợ việc giải quyết vấn đề theo các cách khác nhau như sau:

- Hầu hết các ngôn ngữ lập trình đều là lập trình thủ tục: mỗi chương trình là danh sách các lệnh yêu cầu máy tính làm gì đó với input. C, Pascal và cả Unix shells đều là ngôn ngữ thủ tục.
- Với ngôn ngữ khai báo, khi ta viết các chi tiết mô tả vấn đề cần được giải quyết, ngôn ngữ sẽ tự tìm cách để thực hiện một cách hiệu quả. SQL là ngôn ngữ khai báo quen thuộc nhất; mỗi truy vấn SQL mô tả dữ liệu mà ta muốn truy xuất, và SQL sẽ tự quyết định nên scan bảng hay tra chỉ số, nên thực hiện subclause nào trước, ...
- Các chương trình hướng đối tượng quản lý tập hợp các đối tượng. Mỗi đối tượng có đặc tính và phương thức để truy vấn và điều chỉnh các đặc tính. Smalltalk và Java là các ngôn ngữ lập trình hướng đối tượng. C++ và Python có hỗ trợ việc lập trình hướng đối tượng, nhưng không bắt buộc phải dùng.
- Lập trình hàm chia nhỏ vấn đề thành tập hợp các hàm để giải quyết. Lý tưởng nhất là các hàm chỉ nhận input và sinh ra output, và không có phương thức nào ảnh hưởng đến output sinh ra từ một input đã cho. Các ngôn ngữ lập trình hàm phổ biến bao gồm họ ML(Standard ML, OCaml và các biến thể) và Haskell.

Các nhà thiết kế của một số ngôn ngữ máy tính muốn tập trung vào một cách tiếp cận cụ thể để lập trình. Điều này thường gây khó khăn cho việc viết chương trình sử dụng các cách tiếp cận khác. Các ngôn ngữ khác là những ngôn ngữ đa mô hình hỗ trợ một số cách tiếp cận khác nhau. Lisp, C++ và Python là các ngôn ngữ đa mô hình; ta có thể viết các chương trình hoặc thư viện phần lớn mang tính thủ tục, hướng đối tượng hoặc chức năng bằng tất cả các ngôn ngữ này. Trong một chương trình lớn, các phần khác nhau có thể được viết bằng các cách tiếp cận khác nhau; GUI có thể hướng đối tượng trong khi logic xử lý là thủ tục hoặc chức năng chẳng hạn.

Trong một chương trình lập trình hàm, đầu vào đi qua một tập hợp các hàm. Mỗi hàm hoạt động trên đầu vào của nó và tạo ra một số output. Kiểu lập trình hàm không khuyến khích các hàm có tác dụng phụ sửa đổi thuộc tính bên trong hoặc thực hiện các thay đổi khác không hiển thị trong giá trị trả về của hàm. Các hàm hoàn toàn không có tác dụng phụ được gọi là hàm thuần túy. Tránh tác dụng phụ có nghĩa là không sử dụng cấu trúc dữ liệu được cập nhật khi chương trình chạy; output của mọi hàm chỉ phải phụ thuộc vào đầu vào của nó.

Một số ngôn ngữ rất nghiêm ngặt về tính thuần khiết và thậm chí không có câu lệnh gán như $a = 3$ hoặc $c = a + b$, nhưng khó tránh khỏi mọi tác dụng phụ. Ví dụ: in ra màn hình hoặc

ghi vào tập tin đĩa là những tác dụng phụ. Ví dụ: trong Python, lệnh gọi hàm *print*() hoặc *time.sleep*() đều không trả về giá trị hữu ích; tác dụng phụ của chúng là gửi một số văn bản tới màn hình hoặc tạm dừng thực thi trong một giây.

Các chương trình Python được viết theo phong cách lập trình hàm thường sẽ không tránh được tất cả các thao tác I/O hoặc tất cả các phép gán; thay vào đó, chúng sẽ cung cấp giao diện có chức năng nhưng sẽ sử dụng các tính năng không sử dụng hàm trong nội bộ. Ví dụ, việc triển khai một hàm vẫn sẽ sử dụng các phép gán cho các biến cục bộ nhưng sẽ không sửa đổi các biến toàn cục hoặc có tác dụng phụ khác.

Lập trình hàm có thể được coi là đối lập với lập trình hướng đối tượng. Đối tượng chứa một số thuộc tính nội bộ cùng với một tập hợp các lệnh gọi phương thức cho phép ta sửa đổi các thuộc tính này và các chương trình bao gồm việc thực hiện đúng các thay đổi. Lập trình hàm muốn tránh những thay đổi thuộc tính nhiều nhất có thể và hoạt động với luồng dữ liệu giữa các chức năng. Trong Python ta có thể kết hợp hai cách tiếp cận này bằng cách viết các hàm lấy và trả về các thực thể đại diện cho các đối tượng trong ứng dụng của ta (tin nhắn e-mail, giao dịch, v.v.).

Lập trình hàm hình như có một hạn chế. Tại sao ta nên tránh các đối tượng và tác dụng phụ? Phong cách lập trình hàm có những ưu điểm về mặt lý thuyết và thực tiễn:

- Chứng minh hình thức
- Tính mô đun
- Khả năng kết hợp
- Dễ gỡ lỗi và kiểm tra

1.1 Chứng minh hình thức

Lợi ích về mặt lý thuyết là ta có thể xây dựng chứng minh toán học cho một chương trình lập trình hàm.

Trong một thời gian dài, các nhà nghiên cứu đã quan tâm đến việc tìm cách chứng minh các chương trình là đúng về mặt toán học. Điều này khác với việc kiểm tra một chương trình trên nhiều đầu vào và kết luận rằng output của nó thường đúng hoặc đọc mã nguồn của chương trình và kết luận rằng mã có vẻ đúng; thay vào đó, mục tiêu là một bằng chứng nghiêm ngặt cho thấy một chương generator ra kết quả đúng cho tất cả các đầu vào có thể có.

Kỹ thuật được sử dụng để chứng minh chương trình là ghi lại các bất biến, các thuộc tính của dữ liệu đầu vào và các biến của chương trình luôn đúng. Sau đó, đối với mỗi dòng mã, ta chỉ ra rằng nếu các bất biến X và Y là đúng trước khi dòng được thực thi thì các bất biến X' và Y' là đúng sau khi dòng được thực thi. Điều này tiếp tục cho đến khi ta kết thúc chương trình, tại thời điểm đó các bất biến phải phù hợp với các điều kiện mong muốn ở output của chương trình.

Việc tránh các nhiệm vụ trong lập trình hàm nảy sinh vì các nhiệm vụ khó xử lý bằng kỹ thuật này; các phép gán có thể phá vỡ các bất biến đúng trước phép gán mà không tạo ra bất kỳ bất

biến mới nào có thể được nhân rộng về sau.

Thật không may, việc chứng minh tính chính xác của chương trình phần lớn là không thực tế và không liên quan đến phần mềm Python. Ngay cả những chương trình tầm thường cũng yêu cầu những bản chứng minh dài vài trang; bằng chứng về tính đúng đắn đối với một chương trình phức tạp vừa phải sẽ rất lớn và rất ít hoặc không có chương trình nào ta sử dụng hàng ngày (trình thông dịch Python, trình phân tích cú pháp XML, trình duyệt web của ta) có thể được chứng minh là đúng. Ngay cả khi ta viết ra hoặc tạo ra một chứng minh thì vẫn sẽ có vấn đề về việc xác minh bằng chứng đó; có thể có lỗi trong đó và ta tin nhầm rằng mình đã chứng minh chương trình là đúng.

1.2 Tính mô đun

Một lợi ích thiết thực hơn của lập trình hàm là nó buộc ta phải chia nhỏ vấn đề của mình thành từng phần nhỏ. Kết quả là các chương trình có nhiều mô-đun hơn. Việc chỉ định và viết một hàm nhỏ thực hiện một việc sẽ dễ dàng hơn so với hàm lớn thực hiện một phép biến đổi phức tạp. Các hàm nhỏ cũng dễ đọc và dễ kiểm tra lỗi hơn.

1.3 Dễ gỡ lỗi và kiểm tra

Việc kiểm tra và gỡ lỗi một chương trình lập trình hàm sẽ dễ dàng hơn.

Việc gỡ lỗi được đơn giản hóa vì các hàm thường nhỏ và được chỉ định rõ ràng. Khi một chương trình không hoạt động, mỗi hàm là một điểm nơi ta có thể kiểm tra xem dữ liệu có chính xác hay không. ta có thể xem xét đầu vào và output trung gian để nhanh chóng xác định hàm gây ra lỗi.

Việc kiểm tra dễ dàng hơn vì các hàm đã được chia nhỏ. Các hàm không phụ thuộc vào trạng thái hệ thống cần được sao chép trước khi chạy thử nghiệm; thay vào đó ta chỉ cần tổng hợp đầu vào phù hợp và sau đó kiểm tra xem output có khớp với mong đợi hay không.

1.4 Khả năng kết hợp

Khi làm việc trên một chương trình lập trình hàm, ta sẽ viết một số hàm với đầu vào và output khác nhau. Một số hàm này chắc chắn sẽ được chuyên biệt hóa cho một ứng dụng cụ thể, nhưng những hàm khác sẽ hữu ích trong nhiều chương trình khác nhau. Ví dụ: một hàm lấy đường dẫn thư mục và trả về tất cả các tệp XML trong thư mục hoặc hàm lấy tên tệp và trả về nội dung của nó, có thể được áp dụng cho nhiều tình huống khác nhau.

Theo thời gian, ta sẽ hình thành một thư viện tiện ích cá nhân. Thông thường, ta sẽ tập hợp các chương trình mới bằng cách sắp xếp các hàm hiện có trong một cấu hình mới và viết một số hàm chuyên dụng cho tác vụ hiện tại.

2 Iterators

Ta sẽ bắt đầu bằng cách xem xét một tính năng của ngôn ngữ Python, một tính năng quan trọng để viết các chương trình kiểu lập trình hàm: iterators.

Iterators là một đối tượng đại diện cho một luồng dữ liệu; đối tượng này trả về dữ liệu một phần tử tại một thời điểm. Iterator trong Python phải hỗ trợ một phương thức có tên *_next_*() không có đối số và luôn trả về phần tử tiếp theo của luồng. Nếu không còn phần tử nào trong luồng, *_next_*() phải đưa ra ngoại lệ *StopIteration*. Tuy nhiên, các iterator không nhất thiết phải hữu hạn; hoàn toàn hợp lý khi viết một iterator tạo ra luồng dữ liệu vô hạn.

Hàm *iter*() tích hợp lấy một đối tượng tùy ý và cố gắng trả về một iterator sẽ trả về nội dung hoặc phần tử của đối tượng, gây ra *TypeError* nếu đối tượng không hỗ trợ phép lặp. Một số kiểu dữ liệu tích hợp của Python hỗ trợ phép lặp, phổ biến nhất là list và dictionary. Một đối tượng được gọi là có thể lặp nếu ta có thể lấy một iterator cho nó.

Ta có thể thử nghiệm iterator theo cách thủ công:

```
>>> L = [1,2,3]
>>> it = iter(L)
>>> it
<...iterator object at ...>
>>> it.__next__() # same as next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
StopIteration
>>>
```

Python thường có các đối tượng có thể lặp lại trong một số ngữ cảnh khác nhau, quan trọng nhất là câu lệnh *for*. Trong câu lệnh cho X trong Y, Y phải là một iterator hoặc một đối tượng nào đó mà *iter*() có thể tạo một iterator. Hai câu lệnh này là tương đương:

```
for i in iter(obj):
    print(i)
for i in obj:
    print(i)
```

Các iterator có thể được sử dụng như *list*() hay *tuple*():

```
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

Việc giải nén trình tự cũng hỗ trợ các iterator: nếu ta biết một iterator sẽ trả về N phần tử, ta có thể giải nén chúng thành N-tuple:

```
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> a,b,c = iterator
>>> a,b,c
(1, 2, 3)
```

Các hàm có sẵn như *max*() và *min*() có thể nhận một đối số lặp duy nhất và sẽ trả về phần tử lớn nhất hoặc nhỏ nhất. Các toán tử "*in*" và "*not in*" cũng hỗ trợ các iterator: *X in iterator* là đúng nếu X được tìm thấy trong luồng được iterator trả về. ta sẽ gặp phải vấn đề rõ ràng nếu vòng lặp là vô hạn; *max*(), *min*() sẽ không bao giờ trả về và nếu phần tử X không bao giờ xuất hiện trong luồng thì các toán tử "*in*" và "*not in*" cũng sẽ không trả về.

Lưu ý rằng ta chỉ có thể tiếp tục trong một iterator; không có cách nào để lấy phần tử trước đó, đặt lại iterator hoặc tạo bản sao của phần tử đó. Các đối tượng Iterator có thể tùy ý cung cấp các khả năng bổ sung này, nhưng giao thức iterator chỉ chỉ định phương thức *_next_*(). Do đó, các hàm có thể sử dụng tất cả output của iterator và nếu ta cần thực hiện điều gì đó khác biệt với cùng một luồng, ta sẽ phải tạo một iterator mới.

2.1 Các kiểu dữ liệu hỗ trợ Iterator

Chúng ta đã thấy list và tuple hỗ trợ các iterator như thế nào. Trên thực tế, bất kỳ dữ liệu dạng chuỗi nào, chẳng hạn như string, sẽ tự động hỗ trợ việc tạo một iterator.

Gọi *iter*() trên dictionary sẽ trả về một iterator sẽ lặp qua các khóa của dictionary:

```
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6,
...      'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}
>>> for key in m:
...     print(key, m[key])
Mar 3
Feb 2
Aug 8
Sep 9
Apr 4
Jun 6
Jul 7
Jan 1
May 5
Nov 11
Dec 12
Oct 10
```

Lưu ý rằng thứ tự về cơ bản là ngẫu nhiên vì nó dựa trên thứ tự băm của các đối tượng trong dictionary.

Việc áp dụng *iter*() cho một dictionary luôn lặp lại các khóa, nhưng dictionary có các phương thức trả về các iterator khác. Nếu ta muốn lặp lại các giá trị hoặc cặp khóa/giá trị, ta có thể gọi các phương thức *value*() hoặc *items*() một cách rõ ràng để có được một iterator thích hợp.

Hàm tạo *dict*() có thể chấp nhận một iterator trả về một luồng hữu hạn gồm các tuple (khóa, giá trị):

```
>>> L = [('Italy', 'Rome'), ('France', 'Paris'), ('US', 'Washington DC')]
>>> dict(iter(L))
{'Italy': 'Rome', 'US': 'Washington DC', 'France': 'Paris'}
```

Các file cũng hỗ trợ vòng lặp bằng cách gọi phương thức *readline*() cho đến khi không còn dòng nào trong file. Điều này có nghĩa là ta có thể đọc từng dòng của file như thế này:

```
for line in file:
    # do something for each line
    ...
```

Các set có thể lấy nội dung của chúng từ một iterator và cho phép ta lặp trên các phần tử của set:

```
S = {2, 3, 5, 7, 11, 13}
for i in S:
    print(i)
```

3 Generator expressions và list comprehensions

Hai thao tác phổ biến trên output của iterator là 1) thực hiện một số thao tác cho mọi phần tử, 2) chọn một tập hợp con các phần tử đáp ứng một số điều kiện. Ví dụ: với một danh sách các chuỗi, ta có thể muốn loại bỏ khoảng trắng ở cuối mỗi dòng hoặc trích xuất tất cả các chuỗi chứa một chuỗi con nhất định.

List comprehensions và generator expressions (viết ngắn gọn: "listcomps" và "genexps") là cách gọi ngắn gọn cho các thao tác như vậy, được mượn từ ngôn ngữ lập trình hàm Haskell. ta có thể loại bỏ tất cả khoảng trắng khỏi luồng chuỗi bằng mã sau:

```
line_list = [' line 1\n', 'line 2 \n', ...]
# Generator expression -- returns iterator
stripped_iter = (line.strip() for line in line_list)
# List comprehension -- returns list
stripped_list = [line.strip() for line in line_list]
```

ta chỉ có thể chọn một số phần tử nhất định bằng cách thêm điều kiện "*if*":

```
stripped_list = [line.strip() for line in line_list if line != ""]
```

Với một list comprehension, ta nhận được một Python list; *stripped*$_l$*ist* là list chứa các dòng kết quả, không phải là một iterator. Generator expressions trả về một iterator tính toán các giá trị khi cần thiết, không cần tính tất cả các giá trị cùng một lúc. Điều này có nghĩa là list comprehension không hữu ích nếu ta đang làm việc với các iterator trả về luồng vô hạn hoặc lượng dữ liệu rất lớn. Generator expressions thích hợp hơn trong những tình huống này.

Generator expressions được bao quanh bởi dấu ngoặc đơn ("()") và list comprehension được bao quanh bởi dấu ngoặc vuông ("[]"). Generator expressions có dạng:

```
( expression for expr in sequence1
```

```
            if condition1
            for expr2 in sequence2
            if condition2
            for expr3 in sequence3 ...
            if condition3
            for exprN in sequenceN
            if conditionN )
```

Một lần nữa, list comprehension chỉ có các dấu ngoặc bên ngoài là khác nhau (dấu ngoặc vuông thay vì dấu ngoặc đơn).

Các phần tử của output được tạo sẽ là các giá trị liên tiếp của *expression*. Các mệnh đề *if* đều là tùy chọn; nếu có, biểu thức chỉ được đánh giá và thêm vào kết quả khi điều kiện đúng.

Generator expressions luôn phải được viết bên trong dấu ngoặc đơn, nhưng các dấu ngoặc đơn báo hiệu lệnh gọi hàm cũng được tính. Nếu ta muốn tạo một iterator sẽ được chuyển ngay đến một hàm, ta có thể viết:

```
obj_total = sum(obj.count for obj in list_all_objects())
```

Mệnh đề *for...in* chứa các chuỗi được lặp lại. Các chuỗi không nhất thiết phải có cùng độ dài vì chúng được lặp từ trái sang phải, không song song. Đối với mỗi phần tử trong dãy 1, dãy 2 được lặp lại từ đầu. Sau đó, chuỗi 3 được lặp lại cho từng cặp phần tử thu được từ chuỗi 1 và chuỗi 2.

Nói cách khác, một list comprehension hoặc generator expression tương đương với mã Python sau:

```
for expr1 in sequence1:
    if not (condition1):
        continue # Skip this element
    for expr2 in sequence2:
        if not (condition2):
            continue # Skip this element
        ...
        for exprN in sequenceN:
            if not (conditionN):
                continue # Skip this element
            # Output the value of
            # the expression.
```

Điều này có nghĩa là khi có nhiều mệnh đề *for...in* nhưng không có mệnh đề *if*, độ dài của output sẽ bằng tích độ dài của tất cả các chuỗi. Nếu ta có hai danh sách có độ dài 3 thì danh sách output dài 9 phần tử:

```
>>> seq1 = 'abc'
>>> seq2 = (1,2,3)
>>> [(x, y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3),
('b', 1), ('b', 2), ('b', 3),
('c', 1), ('c', 2), ('c', 3)]
```

Để tránh gây ra sự mơ hồ trong ngữ pháp của Python, nếu *expression* đang tạo một tuple thì nó phải được bao quanh bằng dấu ngoặc đơn. Việc list comprehension đầu tiên dưới đây là một

lỗi cú pháp, trong khi list comprehension thứ hai là đúng:

```
# Syntax error
[x, y for x in seq1 for y in seq2]
# Correct
[(x, y) for x in seq1 for y in seq2]
```

4 Generators

Generators là một lớp hàm đặc biệt giúp đơn giản hóa nhiệm vụ viết các iterator. Các hàm thông thường tính toán một giá trị và trả về giá trị đó, nhưng các generator trả về một iterator trả về một luồng giá trị.

Cách hoạt động của các lệnh gọi hàm thông thường trong Python hoặc C đã trở nên rất quen thuộc. Khi ta gọi một hàm, nó sẽ có một vùng tên riêng nơi các biến cục bộ của nó được tạo. Khi hàm đạt đến câu lệnh *return*, các biến cục bộ sẽ bị hủy và giá trị được trả về cho hàm gọi. Lệnh gọi sau đó tới cùng một hàm sẽ tạo ra một namespace riêng tư mới và một tập hợp các biến cục bộ mới. Tuy nhiên, điều gì sẽ xảy ra nếu các biến cục bộ không bị loại bỏ khi thoát khỏi hàm?

Đây là ví dụ đơn giản nhất về hàm tạo:

```
>>> def generate_ints(N):
... for i in range(N):
... yield i
```

Bất kỳ hàm nào chứa từ khóa *yield* đều là hàm tạo; điều này được phát hiện bởi trình biên dịch của Python để biên dịch hàm một cách đặc biệt.

Khi ta gọi một hàm tạo, nó không trả về một giá trị nào; thay vào đó nó trả về một đối tượng generator hỗ trợ iterator protocol. Khi thực thi biểu thức *yield*, generator sẽ xuất ra giá trị của i, tương tự như câu lệnh trả về. Sự khác biệt lớn giữa câu lệnh *yield* và câu lệnh *return* là khi đạt được *yield*, trạng thái thực thi của generator bị tạm dừng và các biến cục bộ được giữ nguyên. Trong lệnh gọi tiếp theo đến phương thức *_next_*() của generator, hàm sẽ tiếp tục thực thi.

Đây là cách sử dụng pattern của generator *generate_ints*():

```
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
Traceback (most recent call last):
    File "stdin", line 1, in ?
    File "stdin", line 2, in generate_ints
StopIteration
```

ta cũng có thể viết $for\ i\ in\ generate\_ints(5)$ hoặc $a, b, c = generate\_ints(3)$.

Bên trong hàm tạo, $return\ value$ làm cho $StopIteration(value)$ được nâng lên từ phương thức $\_next\_()$. Khi điều này xảy ra hoặc đạt đến cuối hàm, quá trình xử lý các giá trị kết thúc và generator không thể mang lại bất kỳ giá trị nào nữa.

ta có thể đạt được hiệu quả của generator theo cách thủ công bằng cách viết lớp của riêng ta và lưu trữ tất cả các biến cục bộ của generator dưới dạng biến thể hiện. Ví dụ: việc trả về một danh sách các số nguyên có thể được thực hiện bằng cách đặt $self.count$ thành 0 và có phương thức $\_next\_()$ tăng $self.count$ và trả về nó. Tuy nhiên, đối với một generator có độ phức tạp vừa phải, việc viết một lớp tương ứng có thể phức tạp hơn nhiều.

Bộ thử nghiệm đi kèm với thư viện Python, $Lib/test/test\_generators.py$, chứa một số ví dụ thú vị hơn. Dây là một generator thực hiện việc duyệt cây theo thứ tự bằng cách sử dụng generator đệ quy.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x
        yield t.label

        for x in inorder(t.right):
            yield x
```

Hai ví dụ khác trong $test\_generators.py$ đưa ra lời giải cho bài toán N-Queens (đặt N quân hậu trên bàn cờ NxN để không có quân hậu nào ăn quân khác) và Knight's Tour (tìm đường đưa quân mã đến mọi ô của bàn cờ NxN mà không đi qua bất kỳ ô vuông nào hai lần).

4.1 Truyền giá trị vào generator

Trong Python 2.4 trở về trước, generator chỉ tạo ra output. Sau khi mã của generator được gọi để tạo iterator, không có cách nào chuyển bất kỳ thông tin mới nào vào hàm khi quá trình thực thi của nó được tiếp tục. ta có thể kết hợp khả năng này bằng cách làm cho generator xem xét một biến toàn cục hoặc bằng cách chuyển vào một số đối tượng có thể thay đổi, nhưng những cách tiếp cận này rất lộn xộn.

Trong Python 2.5 có một cách đơn giản để chuyển các giá trị vào generator. $yield$ đã trở thành một biểu thức, trả về một giá trị có thể được gán cho một biến:

```
val = (yield i)
```

Nên đặt dấu ngoặc đơn xung quanh biểu thức $yield$ khi ta đang làm điều gì đó với giá trị được trả về, như trong ví dụ trên. Dấu ngoặc đơn không phải lúc nào cũng cần thiết nhưng việc thêm chúng vào luôn sẽ dễ dàng hơn thay vì phải nhớ khi cần.

PEP 342 giải thích các quy tắc chính xác, đó là biểu thức $yield$ phải luôn được đặt trong ngoặc đơn trừ khi nó xuất hiện ở biểu thức cấp cao nhất ở phía bên phải của phép gán. Điều này có nghĩa là ta có thể viết $val = yield\ i$ nhưng phải sử dụng dấu ngoặc đơn khi có phép toán, như

trong $val = (yield\ i) + 12$.

Giá trị được truyền vào hàm tạo bởi cách gọi phương thức $send(value)$. Phương thức này tiếp tục thực thi hàm tạo và biểu thức *yield* trả về giá trị cụ thể. Nếu phương thức $\_next()\_$ được gọi, *yield* trả về *None*.

Đây là một biến đếm đơn giản tăng thêm 1 và cho phép thay đổi giá trị của biến đếm bên trong.

```
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

Và đây là một ví dụ về việc thay đổi biến đếm:

```
>>> it = counter(10)
>>> next(it)
0
>>> next(it)
1
>>> it.send(8)
8
>>> next(it)
9
>>> next(it)
Traceback (most recent call last):
    File "t.py", line 15, in ?
        it.next()
StopIteration
```

Vì *yield* thường trả về *None*, ta nên kiểm tra trường hợp này. Đừng chỉ dùng giá trị trong biểu thức trừ khi chắc chắn rằng phương thức *send*() là phương thức duy nhất tiếp tục thực thi hàm tạo.

Bên cạnh *send*() còn có 2 phương thức khác với hàm tạo:

- $throw(type, value = None, traceback = None)$ được sử dụng để đưa ra một ngoại lệ bên trong generator; các ngoại lệ được đưa ra bởi biểu thức *yield* trong đó quá trình thực thi của generator bị tạm dừng.
- *close*() tạo ra ngoại lệ *GeneratorExit* bên trong generator để kết thúc vòng lặp. Khi nhận được ngoại lệ này, mã của generator phải tăng *GeneratorExit* hoặc *StopIteration*; xử lý ngoại lệ và làm bất cứ điều gì khác là không được phép và sẽ gây ra *RuntimeError*. *close*() cũng sẽ được trình thu gom rác của Python gọi khi generator được thu gom rác.

  Nếu ta cần chạy mã dọn dẹp khi *GeneratorExit* xảy ra, tôi khuyên ta nên sử dụng $try : ...finally :$ thay vì xử lý *GeneratorExit*.

Tác động tích lũy của những thay đổi này là biến người tạo ra thông tin một chiều thành cả người sản xuất và người tiêu dùng.

generator cũng trở thành coroutine, một dạng chương trình con tổng quát hơn. Các chương trình con được nhập tại một điểm và thoát tại một điểm khác (đầu hàm và câu lệnh *return*), nhưng các coroutine có thể được nhập, thoát và tiếp tục ở nhiều điểm khác nhau (câu lệnh *yield*).

5 Built-in functions

Chúng ta hãy xem xét chi tiết hơn các hàm dựng sẵn thường được sử dụng với các vòng lặp.

Hai trong số các hàm dựng sẵn của Python, map() và filter() sao chép các tính năng của biểu thức generator:

$map(f, iterA, iterB, ...)$ trả về một iterator trên chuỗi $f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2]$

```
>>> def upper(s):
    ... return s.upper()
>>> list(map(upper, ['sentence', 'fragment']))
['SENTENCE', 'FRAGMENT']
>>> [upper(s) for s in ['sentence', 'fragment']]
['SENTENCE', 'FRAGMENT']
```

Tất nhiên ta cũng có thể làm vậy với list comprehension.

$filter(predicate, iter)$ trả về một iterator trên tất cả các thành phần chuỗi đáp ứng một điều kiện nhất định và được sao chép bằng list comprehension. Predicate (vị từ) là hàm trả về giá trị đúng của một số điều kiện; để sử dụng với $filter()$, predicate phải nhận một giá trị duy nhất.

```
>>> def is_even(x):
... return (x % 2) == 0
>>> list(filter(is_even, range(10)))
[0, 2, 4, 6, 8]
```

Có thể viết dưới dạng list comprehension:

```
>>> list(x for x in range(10) if is_even(x))
[0, 2, 4, 6, 8]
```

$enumerate(iter)$ đếm số phần tử trong iterable, trả về 2-tuple chứa số đếm và từng phần tử.

```
>>> for item in enumerate(['subject', 'verb', 'object']):
... print(item)
(0, 'subject')
(1, 'verb')
(2, 'object')
```

enumerate() thường được sử dụng khi lặp qua danh sách và ghi lại các chỉ mục đáp ứng các điều kiện nhất định:

```
f = open('data.txt', 'r')
for i, line in enumerate(f):
    if line.strip() == '':
        print('Blank line at line #%i' % i)
```

$sorted(iterable, key = None, reverse = False)$ thu thập tất cả các phần tử của iterable vào một danh sách, sắp xếp danh sách và trả về kết quả được sắp xếp. Các đối số *key* và *reverse* được chuyển qua phương thức *sort*() của danh sách được xây dựng.

```
>>> import random
>>> # Generate 8 random numbers between [0, 10000)
>>> rand_list = random.sample(range(10000), 8)
>>> rand_list
[769, 7953, 9828, 6431, 8442, 9878, 6213, 2207]
>>> sorted(rand_list)
[769, 2207, 6213, 6431, 7953, 8442, 9828, 9878]
>>> sorted(rand_list, reverse=True)
[9878, 9828, 8442, 7953, 6431, 6213, 2207, 769]
```

Hàm $any(iter)$ và $all(iter)$ nhìn vào bảng chân trị của một iterable. *any*() trả về True nếu tồn tại một phần tử trong iterable mang giá trị đúng, và *all*() trả về True nếu tất cả phần tử đều đúng:

```
>>> any([0,1,0])
True
>>> any([0,0,0])
False
>>> any([1,1,1])
True
>>> all([0,1,0])
False
>>> all([0,0,0])
False
>>> all([1,1,1])
True
```

$zip(iterA, iterB, ...)$ lấy mỗi phần tử từ iterable và trả về chúng trong một tuple:

```
zip(['a', 'b', 'c'], (1, 2, 3)) =>
    ('a', 1), ('b', 2), ('c', 3)
```

Nó không xây dựng danh sách trong bộ nhớ và loại bỏ tất cả các vòng lặp đầu vào trước khi quay lại; thay vào đó, các tuple được xây dựng và chỉ trả về nếu chúng được yêu cầu. (Thuật ngữ kỹ thuật cho hành vi này là lazy evaluation)

iterator này được thiết kế để sử dụng với các vòng lặp có cùng độ dài. Nếu các lần lặp có độ dài khác nhau, luồng kết quả sẽ có cùng độ dài với lần lặp ngắn nhất.

```
zip(['a', 'b'], (1, 2, 3)) =>
    ('a', 1), ('b', 2)
```

Tuy nhiên, ta nên tránh làm điều này vì một phần tử có thể được lấy từ các vòng lặp dài hơn và bị loại bỏ. Điều này có nghĩa là ta không thể tiếp tục sử dụng các iterator vì ta có nguy cơ bỏ qua phần tử bị loại bỏ.

6 The itertools module

Mô-đun itertools chứa một số trình lặp thường được sử dụng cũng như các hàm để kết hợp một số iterator. Phần này sẽ giới thiệu nội dung của mô-đun qua các ví dụ nhỏ.

Các chức năng của mô-đun:

- Các hàm tạo một iterator mới dựa trên một iterator hiện có.
- Các hàm xử lý các phần tử của iterator như các đối số của hàm.
- Các hàm chọn các phần output của iterator.
- Một hàm để nhóm output của một iterator.

6.1 Tạo các iterator mới

itertools.count(n) trả về một dòng số nguyên vô hạn, mỗi lần tăng thêm 1. ta có thể tùy ý cung cấp số bắt đầu, mặc định là 0:

```
itertools.count() =>
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

itertools.count(10) =>
    10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
```

itertools.cycle($iter$) lưu một bản sao nội dung của một iterable được cung cấp và trả về một iterator mới trả về các phần tử của nó từ đầu đến cuối. iterator mới sẽ lặp lại các phần tử này vô tận.

```
itertools.cycle([1,2,3,4,5]) =>
    1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...
```

itertools.repeat($elem$, $[n]$) trả về phần tử được cung cấp n lần hoặc trả về phần tử vô tận nếu n không được cung cấp.

```
itertools.repeat('abc') =>
    abc, abc, abc, abc, abc, abc, abc, abc, abc, abc, ...
itertools.repeat('abc', 5) =>
    abc, abc, abc, abc, abc
```

itertools.chain($iterA, iterB, ...$) lấy một số lần lặp tùy ý làm đầu vào và trả về tất cả các phần tử của lần lặp đầu tiên, sau đó là tất cả các phần tử của lần lặp thứ hai, v.v., cho đến khi tất cả các lần lặp đã được thực hiện.

```
itertools.chain(['a', 'b', 'c'], (1, 2, 3)) =>
    a, b, c, 1, 2, 3
```

itertools.islice($iter$, $[start]$, $stop$, $[step]$) trả về một luồng là một phần của iterator. Với một đối số *stop* duy nhất, nó sẽ trả về các phần tử *stop* đầu tiên. Nếu ta cung cấp chỉ mục bắt đầu, ta

sẽ nhận được các phần tử $stop - start$ và nếu ta cung cấp giá trị $step$, các phần tử sẽ bị bỏ qua tương ứng. Không giống như việc cắt chuỗi và danh sách của Python, ta không thể sử dụng các giá trị âm cho $stop, start, step$.

```
itertools.islice(range(10), 8) =>
    0, 1, 2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8) =>
    2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8, 2) =>
    2, 4, 6
```

$itertools.tee(iter, [n])$ sao chép một iterator; nó trả về n iterator độc lập, tất cả sẽ trả về nội dung của iterator nguồn. Nếu ta không cung cấp giá trị cho n thì giá trị mặc định là 2. Việc sao chép các iterator yêu cầu lưu một số nội dung của iterator nguồn, do đó, điều này có thể tiêu tốn bộ nhớ đáng kể nếu iterator lớn và một trong các iterator mới được tiêu thụ nhiều hơn những iterator khác.

```
itertools.tee( itertools.count() ) =>
    iterA, iterB
where iterA ->
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
and iterB ->
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

6.2 Gọi hàm trên các phần tử

Mô-đun *operator* chứa một tập hợp các hàm tương ứng với các toán tử của Python. Một số ví dụ là $operator.add(a, b)$ (cộng hai giá trị), $operator.ne(a, b)$ (giống như $a! = b$) và $operator.attrgetter('id')$ (trả về một giá trị có thể gọi được tìm nạp thuộc tính $.id$).

$itertools.starmap(func, iter)$ giả định rằng iterable sẽ trả về một luồng các tuple và gọi $func$ bằng cách sử dụng các tuple này làm đối số:

```
itertools.starmap(os.path.join, [('/bin', 'python'), ('/usr', 'bin', 'java')
                                    , ('/usr', 'bin', 'perl'), ('/usr', '
                                    bin', 'ruby')]) =>
    /bin/python, /usr/bin/java, /usr/bin/perl, /usr/bin/ruby
```

6.3 Lựa chọn phần tử

Một nhóm hàm khác chọn một tập hợp con các phần tử của iterator dựa trên một vị từ.

$itertools.filterfalse(predicate, iter)$ ngược lại với $filter()$, trả về tất cả các phần tử mà vị từ trả về false:

```
itertools.filterfalse(is_even, itertools.count()) =>
    1, 3, 5, 7, 9, 11, 13, 15, ...
```

$itertools.takewhile(predicate, iter)$ trả về các phần tử miễn là vị từ trả về true. Khi vị từ trả về sai, iterator sẽ báo hiệu kết thúc của nó.

```
def less_than_10(x):
    return x < 10

itertools.takewhile(less_than_10, itertools.count()) =>
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9

itertools.takewhile(is_even, itertools.count()) =>
    0
```

itertools.dropwhile(*predicate*, *iter*) loại bỏ các phần tử trong khi vị từ trả về true và sau đó trả về phần còn lại của kết quả lặp lại.

```
itertools.dropwhile(less_than_10, itertools.count()) =>
    10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...

itertools.dropwhile(is_even, itertools.count()) =>
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

itertools.compress(*data*, *selectors*) lấy hai iterator và chỉ trả về những phần tử của *data* mà phần tử tương ứng của *selector* là đúng, dừng bất cứ khi nào một trong hai không còn phần tử nào:

```
itertools.compress([1,2,3,4,5], [True, True, False, False, True]) =>
    1, 2, 5
```

6.4 Hàm tổ hợp

itertools.combinations(*iterable*, *r*) trả về một iterator đưa ra tất cả các tổ hợp r-tuple có thể có của các phần tử có trong *iterable*.

```
itertools.combinations([1, 2, 3, 4, 5], 2) =>
    (1, 2), (1, 3), (1, 4), (1, 5), (2, 3),
    (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)

itertools.combinations([1, 2, 3, 4, 5], 3) =>
    (1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5),
    (1, 4, 5), (2, 3, 4), (2, 3, 5), (2, 4, 5), (3, 4, 5)
```

Các phần tử trong mỗi tuple vẫn giữ nguyên thứ tự khi *iterable* trả về chúng. Ví dụ: số 1 luôn đứng trước 2, 3, 4 hoặc 5 trong các ví dụ trên. Một hàm tương tự, *itertools.permutations*(*iterable*, $r = None$), loại bỏ ràng buộc này trên thứ tự, trả về tất cả các cách sắp xếp có thể có của độ dài r:

```
itertools.permutations([1, 2, 3, 4, 5], 2) =>
    (1, 2), (1, 3), (1, 4), (1, 5),
    (2, 1), (2, 3), (2, 4), (2, 5),
    (3, 1), (3, 2), (3, 4), (3, 5),
    (4, 1), (4, 2), (4, 3), (4, 5),
    (5, 1), (5, 2), (5, 3), (5, 4)

itertools.permutations([1, 2, 3, 4, 5]) =>
    (1, 2, 3, 4, 5), (1, 2, 3, 5, 4), (1, 2, 4, 3, 5),
    ...
    (5, 4, 3, 2, 1)
```

Nếu ta không cung cấp giá trị cho r thì độ dài của iterable sẽ được sử dụng, nghĩa là tất cả các phần tử đều được hoán vị.

Lưu ý rằng các hàm này tạo ra tất cả các kết hợp có thể có theo vị trí và không yêu cầu nội dung của iterable là duy nhất:

```
itertools.permutations('aba', 3) =>
    ('a', 'b', 'a'), ('a', 'a', 'b'), ('b', 'a', 'a'),
    ('b', 'a', 'a'), ('a', 'a', 'b'), ('a', 'b', 'a')
```

tuple giống hệt nhau $('a','a','b')$ xuất hiện hai lần, nhưng hai chuỗi 'a' đến từ các vị trí khác nhau.

Hàm $itertools.combinations\_with\_replacement(iterable, r)$ loại bỏ một ràng buộc khác: các phần tử có thể được lặp lại trong một tuple. Về mặt khái niệm, một phần tử được chọn cho vị trí đầu tiên của mỗi tuple và sau đó được thay thế trước phần tử thứ hai được chọn.

```
itertools.combinations_with_replacement([1, 2, 3, 4, 5], 2) =>
    (1, 1), (1, 2), (1, 3), (1, 4), (1, 5),
    (2, 2), (2, 3), (2, 4), (2, 5),
    (3, 3), (3, 4), (3, 5),
    (4, 4), (4, 5),
    (5, 5)
```

6.5 Nhóm các phần tử

Hàm cuối cùng tôi sẽ thảo luận, $itertools.groupby(iter, key\_func = None)$, là hàm phức tạp nhất. $key\_func(elem)$ là một hàm có thể tính toán giá trị khóa cho mỗi phần tử được trả về bởi iterable. Nếu ta không cung cấp hàm key, thì key chỉ đơn giản là từng phần tử.

groupby() thu thập tất cả các phần tử liên tiếp từ iterable cơ bản có cùng giá trị khóa và trả về một luồng gồm 2 bộ chứa giá trị khóa và một iterator cho các phần tử có khóa đó.

```
city_list = [('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL'), ('
                                    Anchorage', 'AK'), ('Nome', 'AK'),
             ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ'), ... ]

def get_state(city_state):
    return city_state[1]

itertools.groupby(city_list, get_state) =>
    ('AL', iterator-1),
    ('AK', iterator-2),
    ('AZ', iterator-3), ...

where
iterator-1 =>
    ('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL')
iterator-2 =>
    ('Anchorage', 'AK'), ('Nome', 'AK')
iterator-3 =>
    ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ')
```

groupby() giả định rằng nội dung của iterable cơ bản sẽ được sắp xếp dựa trên khóa. Lưu ý rằng các iterator được trả về cũng sử dụng iterator cơ bản, vì vậy ta phải sử dụng các kết quả của iterator-1 trước khi yêu cầu iterator-2 và khóa tương ứng của nó.

7 Mô-đun functools

Mô-đun *functools* trong Python 2.5 chứa một số hàm bậc cao hơn. Hàm bậc cao hơn lấy một hoặc nhiều hàm làm đầu vào và trả về một hàm mới. Công cụ hữu ích nhất trong mô-đun này là hàm *functools.partial*()

Đối với các chương trình được viết theo kiểu hàm, đôi khi ta sẽ muốn xây dựng các biến thể của hàm hiện có có điền một số tham số. Hãy xem xét hàm Python $f(a, b, c)$; ta có thể muốn tạo một hàm mới $g(b, c)$ tương đương với $f(1, b, c)$; ta đang điền giá trị cho một trong các tham số của $f()$. Đây được gọi là "partial function application".

Hàm tạo của *partial*() lấy các đối số $(function, arg1, arg2, ..., kwarg1 = value1, kwarg2 = value2)$. Đối tượng kết quả có thể gọi được, vì vậy ta chỉ cần gọi nó để gọi hàm với các đối số được điền vào.

Đây là một ví dụ nhỏ nhưng thực tế:

```
def log(message, subsystem):
    """Write the contents of 'message' to the specified subsystem."""
    print('%s: %s' % (subsystem, message))
    ...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

$functools.reduce(func, iter, [initial\_value])$ thực hiện tích lũy một thao tác trên tất cả các phần tử của iterable và do đó, không thể áp dụng cho các lần lặp vô hạn. $func$ phải là hàm nhận vào hai phần tử và trả về một giá trị duy nhất. *functools.reduce*() lấy hai phần tử đầu tiên A và B được iterator trả về và tính toán $func(A, B)$. Sau đó, nó yêu cầu phần tử thứ ba, C, tính toán $func(func(A, B), C)$, kết hợp kết quả này với phần tử thứ tư được trả về và tiếp tục cho đến khi hết phần tử lặp. Nếu iterable không trả về giá trị nào cả, ngoại lệ *TypeError* sẽ xuất hiện. Nếu giá trị ban đầu được cung cấp, giá trị đó sẽ được dùng làm điểm bắt đầu và $func(initial\_value, A)$ là phép tính đầu tiên.

```
>>> import operator, functools
>>> functools.reduce(operator.concat, ['A', 'BB', 'C'])
'ABBC'
>>> functools.reduce(operator.concat, [])
Traceback (most recent call last):
    ...
TypeError: reduce() of empty sequence with no initial value
>>> functools.reduce(operator.mul, [1,2,3], 1)
6
>>> functools.reduce(operator.mul, [], 1)
1
```

Nếu ta sử dụng *operator.add*() với *functools.reduce*(), ta sẽ cộng tất cả các phần tử của iterable.

Trường hợp này phổ biến đến mức có một hàm tích hợp đặc biệt gọi là *sum*() để tính toán nó:

```
>>> import functools, operator
>>> functools.reduce(operator.add, [1,2,3,4], 0)
10
>>> sum([1,2,3,4])
10
>>> sum([])
0
```

Tuy nhiên, đối với nhiều cách sử dụng *functools.reduce*(), có thể rõ ràng hơn nếu chỉ viết vòng lặp for rõ ràng:

```
import functools
# Instead of:
product = functools.reduce(operator.mul, [1,2,3], 1)

# You can write:
product = 1
for i in [1,2,3]:
    product *= i
```

Một hàm liên quan là *itertools.accumulate*(*iterable*, *func* = *operator.add*) < *itertools.accumulate*. Nó thực hiện phép tính tương tự, nhưng thay vì chỉ trả về kết quả cuối cùng, *accumulate*() trả về một iterator cũng mang lại từng kết quả một phần:

```
itertools.accumulate([1,2,3,4,5]) =>
    1, 3, 6, 10, 15

itertools.accumulate([1,2,3,4,5], operator.mul) =>
    1, 2, 6, 24, 120
```

7.1 Mô-đun operator

Mô-đun *operator* đã được đề cập trước đó. Nó chứa một tập hợp các hàm tương ứng với các toán tử của Python. Các hàm này thường hữu ích trong mã lập trình hàm vì chúng giúp ta không phải viết các hàm tầm thường thực hiện một thao tác đơn lẻ.

Một số hàm trong mô-đun này là:

- Các phép toán: $add(), sub(), mul(), Floordiv(), abs(), ...$
- Các phép toán logic: $not\_(), Truth()$
- Các phép toán theo bit: $and\_(), or\_(), invert()$
- So sánh: $eq(), ne(), lt(), le(), gt(), ge()$
- Nhận dạng đối tượng: $is\_(), is\_not()$.

Tham khảo tài liệu của mô-đun operator để có danh sách đầy đủ.

8 Hàm nhỏ và biểu thức lambda

Khi viết các chương trình kiểu lập trình hàm, ta thường sẽ cần các hàm nhỏ đóng vai trò như vị từ hoặc kết hợp các phần tử theo một cách nào đó.

Nếu có sẵn một hàm Python hoặc một hàm mô-đun phù hợp, ta không cần phải xác định hàm mới:

```
stripped_lines = [line.strip() for line in lines]
existing_files = filter(os.path.exists, file_list)
```

Nếu hàm ta cần không tồn tại, ta cần phải viết nó. Một cách để viết các hàm nhỏ là sử dụng câu lệnh *lambda*. *lambda* nhận một số tham số và một biểu thức kết hợp các tham số này rồi tạo một hàm ẩn danh trả về giá trị của biểu thức:

```
adder = lambda x, y: x+y
print_assign = lambda name, value: name + '=' + str(value)
```

Một cách khác là chỉ sử dụng câu lệnh def và định nghĩa hàm theo cách thông thường:

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

def print_assign(name, value):
    return name + '=' + str(value)
```

Lựa chọn thay thế nào là thích hợp hơn? Đó là một câu hỏi về văn phong; lựa chọn thông thường của tôi là tránh sử dụng *lambda*.

Một lý do khiến tôi ưa thích là *lambda* khá hạn chế về các chức năng mà nó có thể xác định. Kết quả phải có thể tính toán được dưới dạng một biểu thức duy nhất, có nghĩa là ta không thể có các so sánh đa chiều *if...elif...else* hoặc các câu lệnh *try...except*. Nếu ta cố gắng làm quá nhiều thứ trong câu lệnh *lambda*, ta sẽ nhận được một biểu thức quá phức tạp và khó đọc. Ví dụ, đoạn mã sau đang làm gì?

```
import functools
total = functools.reduce(lambda a, b: (0, a[1] + b[1]), items)[1]
```

ta có thể hiểu nó, nhưng phải mất thời gian để giải mã biểu thức để tìm hiểu chuyện gì đang xảy ra. Việc sử dụng các câu lệnh *def* lồng nhau ngắn sẽ giúp mọi việc tốt hơn một chút:

```
import functools
def combine(a, b):
    return 0, a[1] + b[1]

total = functools.reduce(combine, items)[1]
```

Nhưng tốt nhất là ta chỉ cần sử dụng vòng lặp for:

```
total = 0
for a, b in items:
    total += b
```

Hoặc hàm sum() có sẵn và một generator expression:

```
total = sum(b for a,b in items)
```

Nhiều cách sử dụng *functools.reduce*() sẽ rõ ràng hơn khi được viết dưới dạng vòng lặp for.

Fredrik Lundh đã từng đề xuất bộ quy tắc sau để tái cấu trúc việc sử dụng lambda:

1. Viết hàm lambda.
2. Viết bình luận giải thích lambda đó làm gì.
3. Nghiên cứu nhận xét và đặt một cái tên có thể nắm bắt được nội dung chính của nhận xét.
4. Chuyển đổi lambda thành câu lệnh def, sử dụng tên đó.
5. Xóa bình luận.

Phần II

Các Design pattern trong OOP

1 Giới thiệu về Design pattern

1.1 Design pattern là gì?

Các design pattern là giải pháp điển hình cho các vấn đề thường xảy ra trong thiết kế phần mềm. Chúng giống như các bản thiết kế được tạo sẵn mà bạn có thể tùy chỉnh để giải quyết vấn đề thiết kế định kỳ trong đoạn code của mình.

Bạn không thể chỉ tìm một pattern và sao chép nó vào chương trình của mình theo cách bạn có thể làm với các hàm hoặc thư viện có sẵn. pattern không phải là một đoạn code cụ thể mà là một khái niệm chung để giải quyết một vấn đề cụ thể. Bạn có thể làm theo các pattern và triển khai giải pháp phù hợp với thực tế chương trình của riêng bạn.

Các pattern thường bị nhầm lẫn với các thuật toán vì cả hai khái niệm đều mô tả các giải pháp điển hình cho một số vấn đề đã biết. Mặc dù thuật toán luôn xác định một tập hợp lệnh rõ ràng có thể đạt được một số mục tiêu, nhưng pattern là mô tả giải pháp ở cấp độ cao hơn. Mã của cùng một pattern áp dụng cho hai chương trình khác nhau có thể khác nhau.

Tương tự với thuật toán là công thức nấu ăn: cả hai đều có các bước rõ ràng để đạt được mục tiêu. Mặt khác, một pattern giống một bản thiết kế hơn: bạn có thể xem kết quả và các tính năng của nó là gì, nhưng thứ tự thực hiện chính xác là tùy thuộc vào bạn.

Pattern gồm những gì?

Hầu hết các pattern được mô tả rất rõ ràng để mọi người có thể sử dụng chúng trong nhiều ngữ cảnh. Dưới đây là các phần thường có trong mô tả pattern:

- Ý định của pattern mô tả ngắn gọn cả vấn đề và giải pháp.
- Động lực giải thích thêm vấn đề và giải pháp mà mô hình có thể tạo ra.
- Cấu trúc của các lớp hiển thị từng phần của pattern và chúng có liên quan như thế nào.
- Ví dụ về mã bằng một trong những ngôn ngữ lập trình phổ biến giúp bạn dễ dàng nắm bắt được ý tưởng đằng sau pattern hơn.

1.2 Lịch sử của các pattern

Ai đã phát minh ra các pattern? Đó là một câu hỏi hay, nhưng không chính xác lắm. Các design pattern không phải là những khái niệm phức tạp, khó hiểu mà hoàn toàn ngược lại. Các

pattern là giải pháp điển hình cho các vấn đề thường gặp trong thiết kế hướng đối tượng. Khi một giải pháp được lặp đi lặp lại trong nhiều dự án khác nhau, cuối cùng sẽ có người đặt tên cho nó và mô tả chi tiết giải pháp đó. Về cơ bản đó là cách một pattern được phát hiện.

Khái niệm về các pattern được mô tả lần đầu tiên bởi Christopher Alexander trong sách "A Pattern Language: Towns, Buildings, Construction". Cuốn sách mô tả một "ngôn ngữ" để thiết kế môi trường đô thị. Đơn vị của ngôn ngữ này là các pattern. Họ có thể mô tả cửa sổ phải cao bao nhiêu, tòa nhà nên có bao nhiêu tầng, diện tích cây xanh trong khu phố phải rộng bao nhiêu, v.v.

Ý tưởng này được bốn tác giả: Erich Gamma, John Vlissides, Ralph Johnson và Richard Helm lựa chọn. Năm 1994, họ xuất bản sách "Design Patterns: Elements of Reusable Object-Oriented Software", trong đó họ áp dụng khái niệm pattern thiết kế vào lập trình. Cuốn sách giới thiệu 23 pattern giải quyết các vấn đề khác nhau của thiết kế hướng đối tượng và nhanh chóng trở thành sách bán chạy nhất. Do cái tên dài dòng của nó, mọi người bắt đầu gọi nó là "cuốn sách của nhóm bốn người (Gang of Four)" và nhanh chóng được rút ngắn thành "cuốn sách GoF".

Kể từ đó, hàng chục mô hình hướng đối tượng khác đã được phát hiện. "Cách tiếp cận theo pattern" đã trở nên rất phổ biến trong các lĩnh vực lập trình khác, vì vậy hiện nay có rất nhiều pattern khác tồn tại ngoài thiết kế hướng đối tượng.

1.3 Vì sao cần học design pattern

Sự thật là bạn có thể làm lập trình viên trong nhiều năm mà không biết về một design pattern nào. Rất nhiều người làm điều đó. Tuy nhiên, ngay cả trong trường hợp đó, bạn có thể đang triển khai một số pattern mà không hề biết. Vậy tại sao bạn lại dành thời gian để học chúng?

Các design pattern là một bộ công cụ gồm các giải pháp đã được thử nghiệm và kiểm tra cho các vấn đề phổ biến trong thiết kế phần mềm. Ngay cả khi bạn chưa bao giờ gặp phải những vấn đề này, việc biết các mẫu vẫn hữu ích vì nó dạy bạn cách giải quyết mọi loại vấn đề bằng cách sử dụng các nguyên tắc thiết kế hướng đối tượng.

Các design pattern xác định một ngôn ngữ chung mà bạn và đồng đội của mình có thể sử dụng để giao tiếp hiệu quả hơn. Bạn có thể nói: "Ồ, chỉ cần sử dụng Singleton cho việc đó" và mọi người sẽ hiểu ý tưởng đằng sau đề xuất của bạn. Không cần phải giải thích singleton là gì nếu bạn biết pattern và tên của nó.

1.4 Các phê bình về design pattern

Có vẻ như chỉ những người lười biếng mới chưa chê design pattern. Chúng ta hãy xem những lập luận điển hình nhất chống lại việc sử dụng các pattern.

Chỉ là giải pháp tạm thời cho các ngôn ngữ lập trình yếu

Thông thường, nhu cầu về các pattern phát sinh khi mọi người chọn ngôn ngữ lập trình hoặc công nghệ thiếu mức độ trừu tượng cần thiết. Trong trường hợp này, các pattern trở thành

một giải pháp tạm thời mang lại cho ngôn ngữ những khả năng siêu phàm rất cần thiết.

Ví dụ pattern Strategy có thể được hiện thực bằng hàm ẩn danh (lambda) đơn giản trong hầu hết các ngôn ngữ lập trình hiện đại.

Giải pháp kém hiệu quả

Các mô hình cố gắng hệ thống hóa các phương pháp tiếp cận đã được sử dụng rộng rãi. Sự thống nhất này được nhiều người coi là giáo điều và họ thực hiện các pattern theo "từng chữ" mà không điều chỉnh chúng cho phù hợp với bối cảnh dự án của họ.

Sử dụng không hợp lý

> If all you have is a hammer, everything looks like a nail.

Đây là vấn đề ám ảnh nhiều người mới làm quen với các pattern. Sau khi tìm hiểu về các pattern, họ cố gắng áp dụng chúng ở mọi nơi, ngay cả trong những tình huống mà mã đơn giản hơn sẽ hoạt động tốt.

1.5 Phân loại các pattern

Các design pattern khác nhau bởi độ phức tạp, mức độ chi tiết và quy mô áp dụng cho toàn bộ hệ thống được thiết kế. Tôi thích sự tương tự với việc xây dựng đường: bạn có thể làm cho giao lộ an toàn hơn bằng cách lắp đặt một số đèn giao thông hoặc xây dựng toàn bộ nút giao thông nhiều tầng với lối đi ngầm dành cho người đi bộ.

Các pattern cơ bản và cấp thấp nhất thường được gọi là *idioms*. Chúng thường chỉ áp dụng cho một ngôn ngữ lập trình duy nhất.

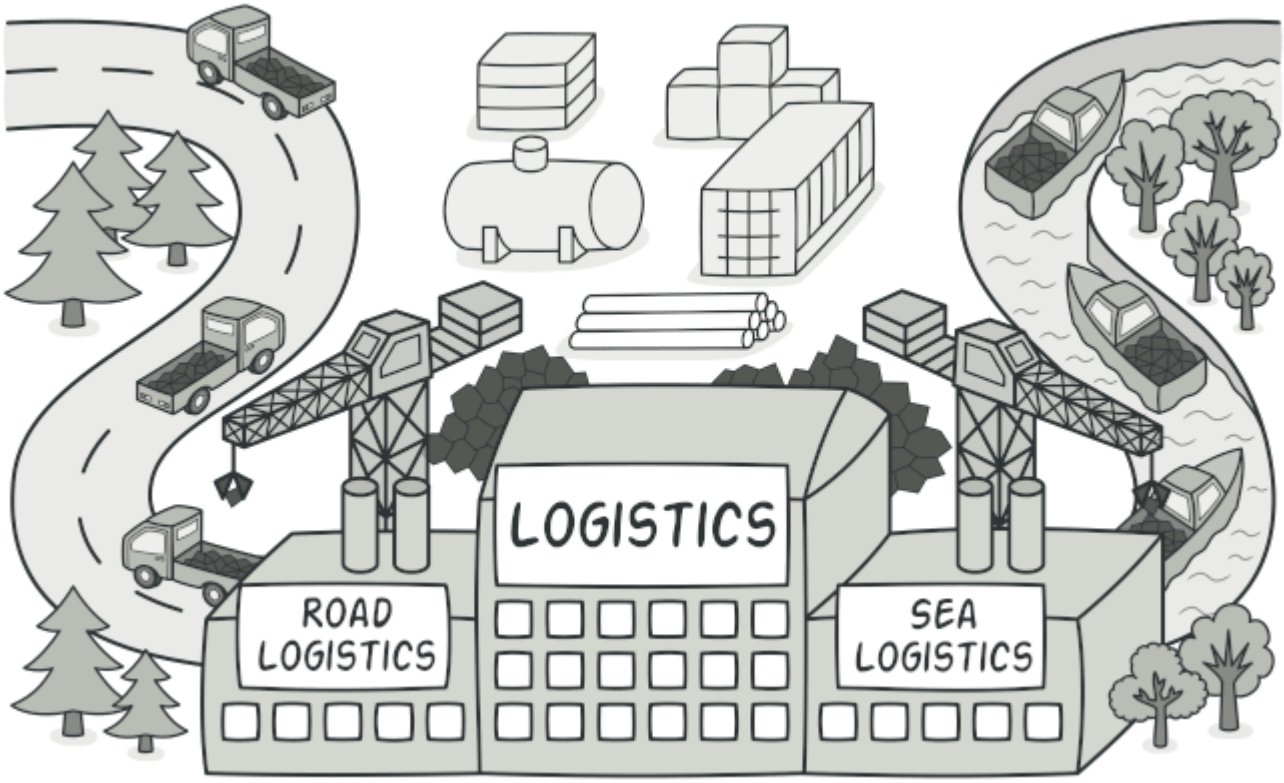
Các pattern phổ quát và cấp cao nhất là các *architectural patterns*. Các nhà phát triển có thể triển khai các pattern này bằng hầu hết mọi ngôn ngữ. Không giống như các pattern khác, chúng có thể được sử dụng để thiết kế kiến trúc của toàn bộ ứng dụng.

Ngoài ra, tất cả các mẫu có thể được phân loại theo ý định hoặc mục đích của chúng. Cuốn sách này bao gồm ba nhóm design pattern chính:

- **Creational patterns** cung cấp các cơ chế tạo đối tượng giúp tăng tính linh hoạt và tái sử dụng mã hiện có.
- **Structural patterns** giải thích cách tập hợp các đối tượng và lớp thành các cấu trúc lớn hơn, đồng thời giữ cho các cấu trúc này linh hoạt và hiệu quả.
- **Behavioral patterns** đảm bảo việc giao tiếp hiệu quả và phân công trách nhiệm giữa các đối tượng.

2 Creational Design Patterns

2.1 Factory Method



2.1.1 Định nghĩa

Factory Method là một creational design pattern cung cấp giao diện để tạo đối tượng trong lớp cha, nhưng cho phép các lớp con thay đổi loại đối tượng sẽ được tạo.

2.1.2 Ứng dụng

Sử dụng Factory Method khi bạn không biết trước các loại chính xác và các phụ thuộc của các đối tượng mà mã của bạn cần làm việc.

Factory Method tách mã xây dựng sản phẩm ra khỏi mã sử dụng sản phẩm thực sự. Do đó, việc mở rộng mã xây dựng sản phẩm dễ dàng hơn mà không phụ thuộc vào phần còn lại của mã.

Ví dụ, để thêm một loại sản phẩm mới vào ứng dụng, bạn chỉ cần tạo một lớp con của creator mới và ghi đè phương thức factory trong đó.

Sử dụng Factory Method khi bạn muốn cung cấp cho người dùng của thư viện hoặc framework của bạn một cách mở rộng các thành phần nội bộ của nó.

Kế thừa có lẽ là cách dễ nhất để mở rộng hành vi mặc định của một thư viện hoặc framework. Nhưng làm thế nào framework có thể nhận ra rằng lớp con của bạn nên được sử dụng thay vì một thành phần tiêu chuẩn?

Giải pháp là giảm mã xây dựng các thành phần trên toàn bộ framework thành một phương thức factory duy nhất và để bất kỳ ai cũng có thể ghi đè phương thức này ngoài việc mở rộng chính thành phần đó.

Hãy xem cách thực hiện điều này. Hãy tưởng tượng rằng bạn viết một ứng dụng bằng cách sử dụng một framework UI mã nguồn mở. Ứng dụng của bạn nên có các nút tròn, nhưng framework chỉ cung cấp các nút vuông. Bạn mở rộng lớp Button tiêu chuẩn với một lớp con RoundButton tuyệt vời. Nhưng bây giờ bạn cần cho biết lớp chính UIFramework phải sử dụng lớp con nút mới thay vì một lớp mặc định.

Để đạt được điều này, bạn tạo một lớp con UIWithRoundButtons từ một lớp cơ sở của framework và ghi đè phương thức createButton của nó. Trong khi phương thức này trả về các đối tượng Button trong lớp cơ sở, bạn làm cho lớp con của bạn trả về các đối tượng RoundButton. Bây giờ hãy sử dụng lớp UIWithRoundButtons thay vì UIFramework. Và đó là tất cả!

Sử dụng Factory Method khi bạn muốn tiết kiệm tài nguyên hệ thống bằng cách tái sử dụng các đối tượng hiện có thay vì xây dựng lại chúng mỗi lần.

Bạn thường gặp nhu cầu này khi làm việc với các đối tượng lớn, tốn tài nguyên như kết nối cơ sở dữ liệu, hệ thống tệp, và tài nguyên mạng.

Hãy suy nghĩ về những gì phải làm để tái sử dụng một đối tượng hiện có:

Trước hết, bạn cần tạo ra một lưu trữ để theo dõi tất cả các đối tượng đã được tạo ra. Khi ai đó yêu cầu một đối tượng, chương trình sẽ tìm kiếm một đối tượng không sử dụng trong hồ bơi đó. ... và sau đó trả về nó cho mã client. Nếu không có đối tượng không sử dụng, chương trình sẽ tạo ra một đối tượng mới (và thêm nó vào hồ bơi). Đó là một lượng mã lớn! Và tất cả phải được đặt vào một nơi duy nhất để bạn không làm ô nhiễm chương trình bằng mã trùng lặp.

Có lẽ nơi rõ ràng và thuận tiện nhất để đặt mã này là trong constructor của lớp mà các đối tượng của chúng ta đang cố gắng tái sử dụng. Tuy nhiên, theo định nghĩa, một constructor phải luôn luôn trả về các đối tượng mới. Nó không thể trả về các thể hiện hiện có.

Do đó, bạn cần có một phương thức thông thường có khả năng tạo ra các đối tượng mới cũng như tái sử dụng các đối tượng hiện có. Điều này nghe có vẻ rất giống như một phương thức factory.

2.1.3 Cách hiện thực

1. Đảm bảo tất cả các sản phẩm tuân theo cùng một giao diện. Giao diện này nên khai báo các phương thức có ý nghĩa trong mọi sản phẩm.

2. Thêm một phương thức factory trống vào bên trong lớp creator. Kiểu trả về của phương thức này nên phù hợp với giao diện sản phẩm chung.

3. Trong mã của creator, tìm tất cả các tham chiếu đến các constructor sản phẩm. Từng cái một, thay thế chúng bằng cuộc gọi đến phương thức factory, trong khi trích xuất mã tạo sản phẩm vào phương thức factory.

Bạn có thể cần thêm một tham số tạm thời vào phương thức factory để kiểm soát loại sản phẩm trả về.

Ở điểm này, mã của phương thức factory có thể trông khá xấu. Nó có thể có một câu lệnh switch lớn để chọn lớp sản phẩm nào để khởi tạo. Nhưng đừng lo lắng, chúng ta sẽ sửa nó sớm thôi.

4. Bây giờ, tạo một tập hợp các lớp con của creator cho mỗi loại sản phẩm được liệt kê trong phương thức factory. Ghi đè phương thức factory trong các lớp con và trích xuất các phần phù hợp của mã xây dựng từ phương thức cơ sở.

5. Nếu có quá nhiều loại sản phẩm và không hợp lý để tạo ra lớp con cho tất cả chúng, bạn có thể tái sử dụng tham số kiểm soát từ lớp cơ sở trong các lớp con.

Ví dụ, hãy tưởng tượng rằng bạn có cấu trúc lớp như sau: lớp cơ sở Mail với một số lớp con: AirMail và GroundMail; các lớp Transport là Plane, Truck và Train. Trong khi lớp AirMail chỉ sử dụng các đối tượng Plane, GroundMail có thể làm việc với cả đối tượng Truck và Train. Bạn có thể tạo ra một lớp con mới (ví dụ TrainMail) để xử lý cả hai trường hợp, nhưng cũng có một lựa chọn khác. Mã client có thể truyền một đối số vào phương thức factory của lớp GroundMail để kiểm soát loại sản phẩm mà nó muốn nhận.

6. Nếu, sau tất cả các trích xuất, phương thức factory cơ sở trở thành trống, bạn có thể làm cho nó trừu tượng. Nếu còn gì đó, bạn có thể làm cho nó trở thành hành vi mặc định của phương thức.

2.1.4 Ưu và nhược điểm

Ưu điểm:

- Bằng cách sử dụng Factory Method, bạn tránh được sự kết hợp chặt chẽ giữa creator và các sản phẩm cụ thể.
- Nguyên lý Đơn trách nhiệm Đơn: Bạn có thể di chuyển mã tạo sản phẩm vào một nơi trong chương trình, làm cho mã dễ hỗ trợ hơn.
- Nguyên lý Mở/Rộng: Bạn có thể giới thiệu các loại sản phẩm mới vào chương trình mà không làm hỏng mã client hiện tại.

Nhược điểm: Mã có thể trở nên phức tạp hơn vì bạn cần giới thiệu nhiều lớp con mới để triển khai mẫu thiết kế. Tình huống tốt nhất là khi bạn đang giới thiệu mẫu thiết kế vào một thứ bậc hiện tại của các lớp creator.

2.1.5 Mối liên hệ với các pattern khác

Nhiều thiết kế bắt đầu bằng cách sử dụng Factory Method (ít phức tạp hơn và dễ tùy chỉnh hơn thông qua các lớp con) và phát triển về Abstract Factory, Prototype, hoặc Builder (linh hoạt hơn, nhưng phức tạp hơn).

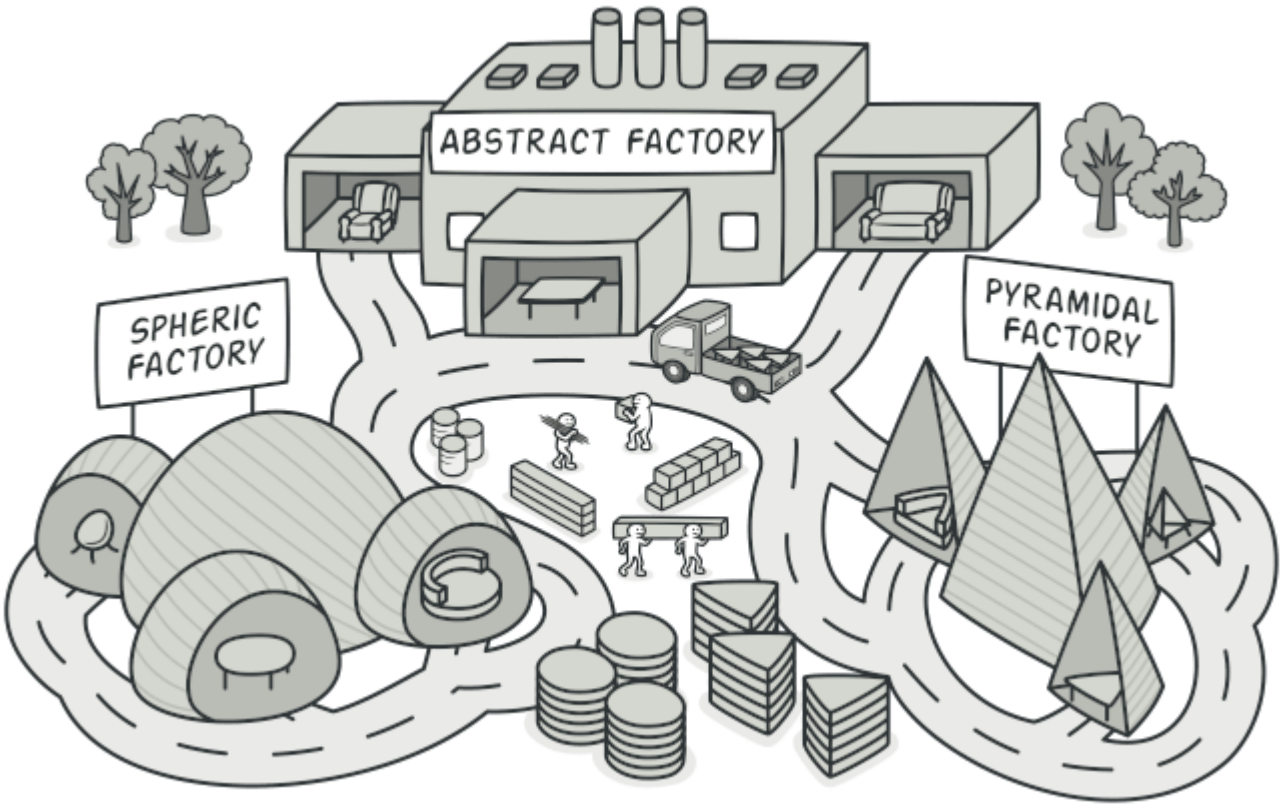
Các lớp Abstract Factory thường được dựa trên một tập hợp các Factory Methods, nhưng bạn cũng có thể sử dụng Prototype để hợp thành các phương thức trên các lớp này.

Bạn có thể sử dụng Factory Method cùng với Iterator để cho phép các lớp sưu tập trả về các loại iterators khác nhau tương thích với các sưu tập.

Prototype không dựa trên việc kế thừa, nên nó không có các hạn chế của nó. Tuy nhiên, Prototype yêu cầu một quá trình khởi tạo phức tạp của đối tượng đã nhân bản. Factory Method dựa trên việc kế thừa nhưng không yêu cầu bước khởi tạo.

Factory Method là một loại đặc biệt của Template Method. Đồng thời, một Factory Method có thể phục vụ như một bước trong một Template Method lớn.

2.2 Abstract Factory



2.2.1 Định nghĩa

Abstract Factory là một mẫu thiết kế sáng tạo cho phép bạn tạo ra các họ đối tượng liên quan mà không cần chỉ định các lớp cụ thể của chúng.

2.2.2 Ứng dụng

Sử dụng Abstract Factory khi mã của bạn cần làm việc với các họ đối tượng liên quan, nhưng bạn không muốn nó phụ thuộc vào các lớp cụ thể của các đối tượng đó - chúng có thể không biết trước hoặc bạn đơn giản chỉ muốn cho phép tính mở rộng trong tương lai.

Abstract Factory cung cấp cho bạn một giao diện để tạo ra các đối tượng từ mỗi lớp trong họ đối tượng. Miễn là mã của bạn tạo ra các đối tượng thông qua giao diện này, bạn không cần lo lắng về việc tạo ra biến thể sai của một sản phẩm không phù hợp với các sản phẩm đã được tạo ra bởi ứng dụng của bạn.

Xem xét triển khai Abstract Factory khi bạn có một lớp với một tập hợp các Factory Methods mà làm mờ trách nhiệm chính của nó.

Trong một chương trình được thiết kế tốt, mỗi lớp chỉ chịu trách nhiệm về một điều. Khi một lớp xử lý nhiều loại sản phẩm, có thể đáng giá để trích xuất các phương thức factory của nó thành một lớp factory độc lập hoặc một triển khai Abstract Factory đầy đủ.

2.2.3 Cách hiện thực

1. Tạo ma trận loại sản phẩm đặc biệt so với các biến thể của các sản phẩm đó.

2. Khai báo giao diện sản phẩm trừu tượng cho tất cả các loại sản phẩm. Sau đó, làm cho tất cả các lớp sản phẩm cụ thể triển khai các giao diện này.

3. Khai báo giao diện Factory trừu tượng với một tập hợp các phương thức tạo ra cho tất cả các sản phẩm trừu tượng.

4. Triển khai một tập hợp các lớp Factory cụ thể, mỗi lớp cho một biến thể sản phẩm.

5. Tạo mã khởi tạo cho Factory ở đâu đó trong ứng dụng. Nó nên khởi tạo một trong các lớp Factory cụ thể, phụ thuộc vào cấu hình ứng dụng hoặc môi trường hiện tại. Truyền đối tượng Factory này cho tất cả các lớp xây dựng sản phẩm.

6. Quét qua mã và tìm tất cả các cuộc gọi trực tiếp đến các constructor sản phẩm. Thay thế chúng bằng các cuộc gọi đến phương thức tạo ra thích hợp trên đối tượng Factory.

2.2.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể chắc chắn rằng các sản phẩm bạn nhận từ một nhà máy là tương thích với nhau.
- Bạn tránh sự kết hợp chặt chẽ giữa các sản phẩm cụ thể và mã client.
- Nguyên lý Đơn trách nhiệm. Bạn có thể trích xuất mã tạo sản phẩm vào một nơi, làm cho mã dễ hỗ trợ hơn.
- Nguyên lý Mở/Rộng. Bạn có thể giới thiệu các biến thể mới của sản phẩm mà không làm hỏng mã client hiện tại.

Nhược điểm: Mã có thể trở nên phức tạp hơn so với cần thiết, vì nhiều giao diện và lớp mới được giới thiệu cùng với mẫu thiết kế này.

2.2.5 Mối liên hệ với các pattern khác

Nhiều thiết kế thường bắt đầu bằng cách sử dụng Factory Method (ít phức tạp hơn và có thể tùy chỉnh hơn thông qua các lớp con) và tiến triển về phía Abstract Factory, Prototype hoặc Builder (linh hoạt hơn, nhưng phức tạp hơn).

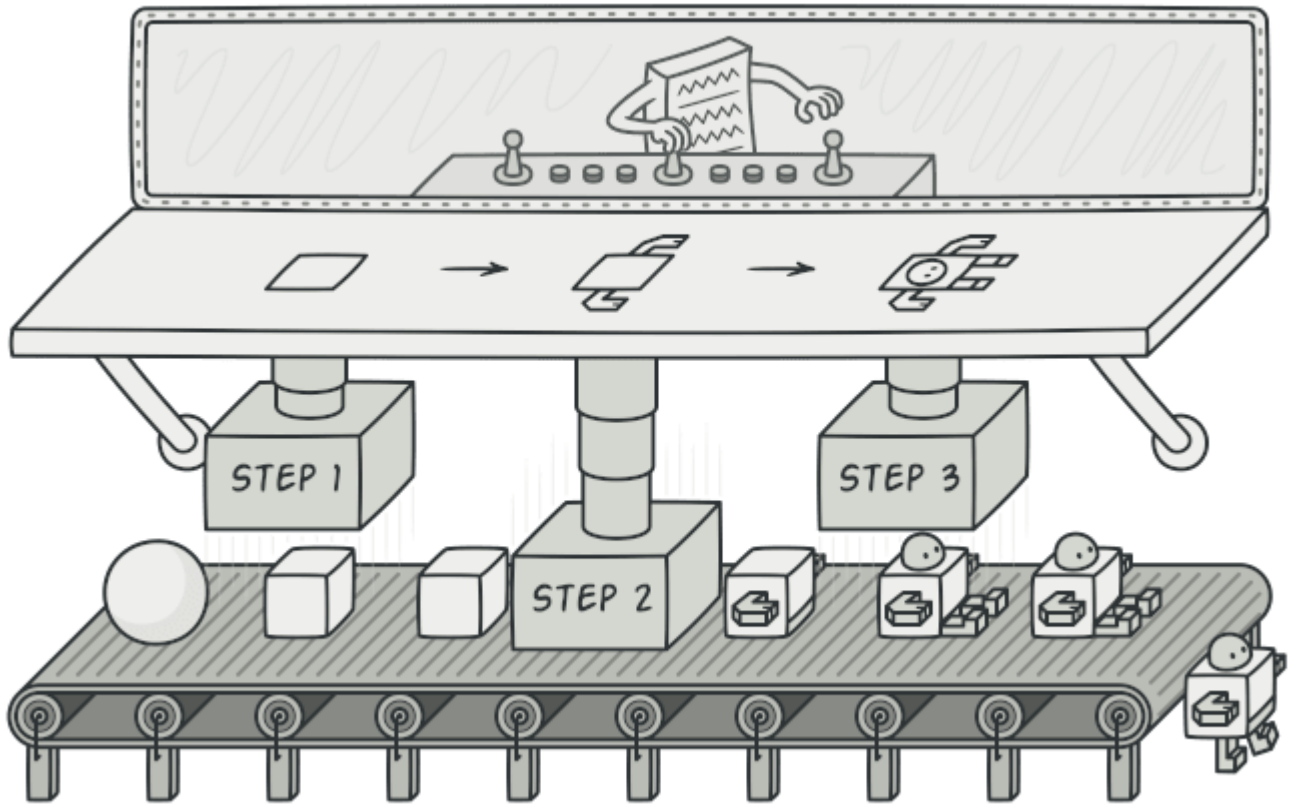
Builder tập trung vào việc xây dựng các đối tượng phức tạp bước by bước. Abstract Factory chuyên tạo ra các họ đối tượng liên quan. Abstract Factory trả về sản phẩm ngay lập tức, trong khi Builder cho phép bạn thực hiện một số bước xây dựng bổ sung trước khi lấy sản phẩm.

Các lớp Abstract Factory thường dựa trên một tập hợp các Factory Methods, nhưng bạn cũng có thể sử dụng Prototype để hợp thành các phương thức trên các lớp này.

Abstract Factory có thể phục vụ như một phương pháp thay thế cho Facade khi bạn chỉ muốn che giấu cách các đối tượng con hệ thống được tạo ra từ mã nguồn của khách hàng.

Bạn có thể sử dụng Abstract Factory cùng với Bridge. Kết hợp này hữu ích khi một số trừu tượng được định nghĩa bởi Bridge chỉ có thể làm việc với các triển khai cụ thể. Trong trường hợp này, Abstract Factory có thể bao gồm các mối quan hệ này và ẩn đi sự phức tạp khỏi mã nguồn của khách hàng.

Abstract Factory, Builder và Prototype đều có thể được triển khai như Singleton.



2.3 Builder

2.3.1 Định nghĩa

Builder là một mẫu thiết kế sáng tạo cho phép bạn xây dựng các đối tượng phức tạp bước by bước. Mẫu thiết kế này cho phép bạn tạo ra các loại và biểu diễn khác nhau của một đối tượng bằng cùng mã xây dựng.

2.3.2 Ứng dụng

Sử dụng mẫu thiết kế Builder để loại bỏ việc sử dụng "constructor telescoping".

Giả sử bạn có một constructor với mười tham số tùy chọn. Gọi một cái như thế rất bất tiện; do đó, bạn tạo ra một số phiên bản ngắn hơn với ít tham số hơn. Những constructor này vẫn tham chiếu đến constructor chính, truyền một số giá trị mặc định vào bất kỳ tham số nào được bỏ qua.

Mẫu thiết kế Builder cho phép bạn xây dựng các đối tượng bước by bước, chỉ sử dụng những bước mà bạn thực sự cần. Sau khi triển khai mẫu thiết kế này, bạn không cần phải chèn hàng chục tham số vào các constructor của mình nữa.

Sử dụng mẫu thiết kế Builder khi bạn muốn mã của mình có khả năng tạo ra các biểu diễn khác nhau của một sản phẩm nào đó (ví dụ: nhà đá và nhà gỗ).

Mẫu thiết kế Builder có thể được áp dụng khi việc xây dựng các biểu diễn khác nhau của sản phẩm liên quan đến các bước tương tự chỉ khác biệt trong các chi tiết.

Giao diện builder cơ sở định nghĩa tất cả các bước xây dựng có thể có, và các builder cụ thể triển khai các bước này để xây dựng các biểu diễn cụ thể của sản phẩm. Trong khi đó, lớp director hướng dẫn thứ tự xây dựng.

Sử dụng Builder để xây dựng cây Composite hoặc các đối tượng phức tạp khác.

Mẫu thiết kế Builder cho phép bạn xây dựng các sản phẩm bước by bước. Bạn có thể trì hoãn việc thực thi một số bước mà không làm hỏng sản phẩm cuối cùng. Bạn còn có thể gọi các bước một cách đệ quy, điều này rất tiện lợi khi bạn cần xây dựng một cây đối tượng.

Một builder không tiết lộ sản phẩm chưa hoàn thành trong quá trình thực hiện các bước xây dựng. Điều này ngăn mã của khách hàng từ việc truy xuất kết quả không hoàn chỉnh.

2.3.3 Cách hiện thực

1. Đảm bảo bạn có thể rõ ràng xác định các bước xây dựng chung cho việc xây dựng tất cả các biểu diễn sản phẩm có sẵn. Nếu không, bạn sẽ không thể tiến hành triển khai mẫu thiết kế này.

2. Khai báo các bước này trong giao diện builder cơ sở.

3. Tạo một lớp builder cụ thể cho mỗi biểu diễn sản phẩm và triển khai các bước xây dựng của chúng.

Đừng quên triển khai một phương thức để lấy kết quả của quá trình xây dựng. Lý do tại sao phương thức này không thể được khai báo trong giao diện builder là vì các builder khác nhau có thể xây dựng các sản phẩm không có một giao diện chung. Do đó, bạn không biết đối tượng trả về của phương thức đó sẽ là gì. Tuy nhiên, nếu bạn đang làm việc với các sản phẩm từ một cấu trúc duy nhất, phương thức lấy có thể được thêm vào giao diện cơ sở một cách an toàn.

4. Suy nghĩ về việc tạo một lớp director. Nó có thể đóng gói các cách khác nhau để xây dựng một sản phẩm bằng cách sử dụng cùng một đối tượng builder.

5. Mã nguồn khách hàng tạo cả đối tượng builder và director. Trước khi bắt đầu quá trình xây dựng, khách hàng phải chuyển một đối tượng builder cho director. Thông thường, khách hàng chỉ làm điều này một lần, thông qua tham số của hàm tạo của lớp director. Director sử dụng đối tượng builder trong tất cả các quá trình xây dựng tiếp theo. Có một cách tiếp cận thay thế, trong đó builder được chuyển đến một phương thức xây dựng sản phẩm cụ thể của director.

6. Kết quả xây dựng chỉ có thể được lấy trực tiếp từ director nếu tất cả các sản phẩm tuân thủ cùng một giao diện. Nếu không, khách hàng nên lấy kết quả từ builder.

2.3.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể xây dựng đối tượng từng bước một, hoãn các bước xây dựng hoặc chạy các bước đệ quy.
- Bạn có thể tái sử dụng cùng mã xây dựng khi xây dựng các biểu diễn khác nhau của sản phẩm.
- Nguyên lý Đơn trách nhiệm. Bạn có thể cách ly mã xây dựng phức tạp khỏi logic kinh doanh của sản phẩm.

Nhược điểm: Tổng quan về sự phức tạp của mã tăng lên vì mẫu thiết kế yêu cầu tạo ra nhiều lớp mới.

2.3.5 Mối liên hệ với các pattern khác

Nhiều thiết kế bắt đầu bằng cách sử dụng Factory Method (ít phức tạp hơn và có thể tùy chỉnh hơn thông qua các lớp con) và phát triển thành Abstract Factory, Prototype hoặc Builder (linh hoạt hơn, nhưng phức tạp hơn).

Builder tập trung vào việc xây dựng các đối tượng phức tạp bước by bước. Abstract Factory chuyên tạo ra các họ đối tượng liên quan. Abstract Factory trả về sản phẩm ngay lập tức, trong khi Builder cho phép bạn chạy một số bước xây dựng bổ sung trước khi lấy sản phẩm.

Bạn có thể sử dụng Builder khi tạo ra các cây Composite phức tạp vì bạn có thể lập trình các bước xây dựng của nó để hoạt động đệ quy.

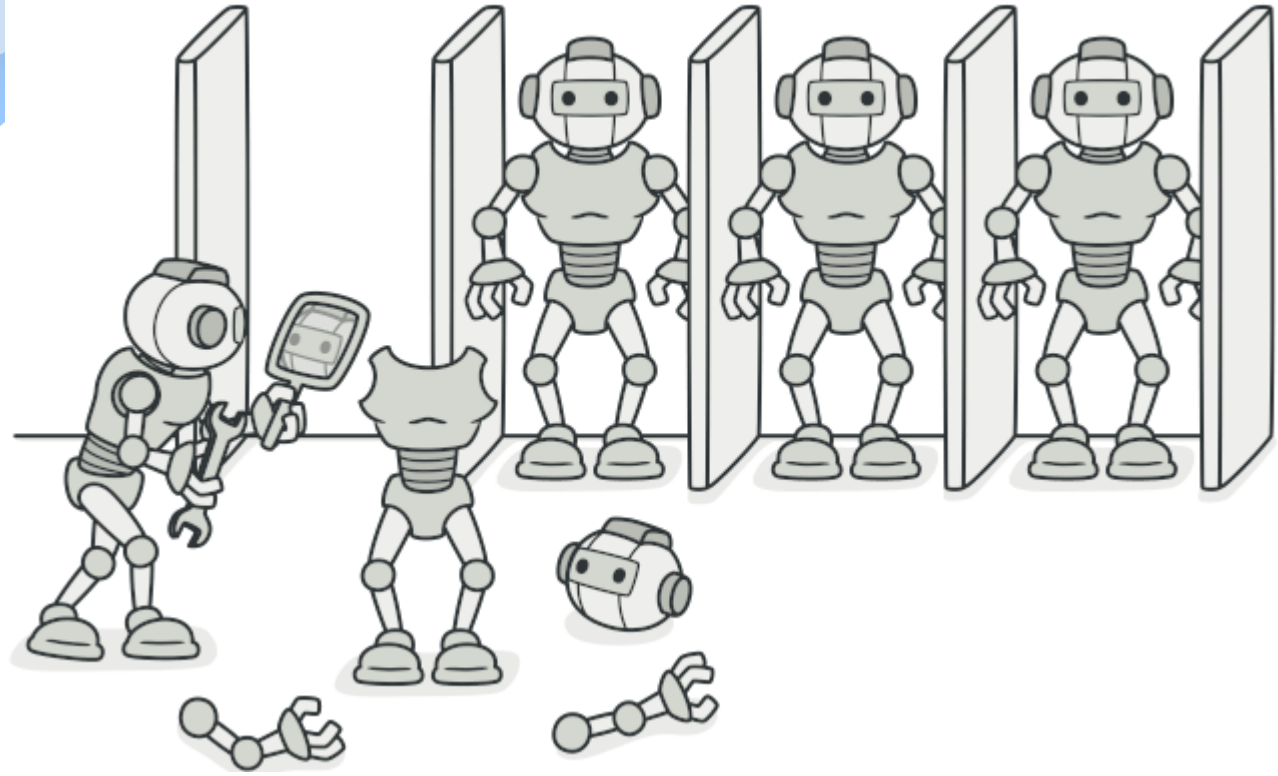
Bạn có thể kết hợp Builder với Bridge: lớp director chơi vai trò của trừu tượng, trong khi các builder khác nhau đóng vai trò của các triển khai.

Abstract Factories, Builders và Prototypes đều có thể được triển khai như Singleton.

2.4 Prototype

2.4.1 Định nghĩa

Prototype là một mẫu thiết kế sáng tạo cho phép bạn sao chép các đối tượng hiện có mà không làm cho mã của bạn phụ thuộc vào các lớp của chúng.



2.4.2 Ứng dụng

Sử dụng mẫu thiết kế Prototype khi mã của bạn không nên phụ thuộc vào các lớp cụ thể của các đối tượng mà bạn cần sao chép.

Điều này thường xảy ra khi mã của bạn làm việc với các đối tượng được chuyển đến từ mã của bên thứ ba thông qua một giao diện nào đó. Các lớp cụ thể của các đối tượng này là không xác định, và bạn không thể phụ thuộc vào chúng ngay cả khi bạn muốn.

Mẫu thiết kế Prototype cung cấp cho mã của khách hàng một giao diện chung để làm việc với tất cả các đối tượng hỗ trợ sao chép. Giao diện này làm cho mã của khách hàng độc lập với các lớp cụ thể của các đối tượng mà nó sao chép.

Sử dụng mẫu khi bạn muốn giảm số lượng các lớp con chỉ khác nhau trong cách họ khởi tạo các đối tượng tương ứng của họ.

Giả sử bạn có một lớp phức tạp yêu cầu một cấu hình khó khăn trước khi có thể sử dụng. Có một số cách phổ biến để cấu hình lớp này, và mã này phân tán trong ứng dụng của bạn. Để giảm sự trùng lặp, bạn tạo ra một số lớp con và đặt mọi mã cấu hình chung vào các hàm tạo của chúng. Bạn đã giải quyết vấn đề trùng lặp, nhưng bây giờ bạn có rất nhiều lớp con giả.

Mẫu thiết kế Prototype cho phép bạn sử dụng một tập hợp các đối tượng được xây dựng trước được cấu hình theo các cách khác nhau như các bản prototype. Thay vì tạo một lớp con phù hợp với một cấu hình nào đó, mã của khách hàng có thể đơn giản tìm kiếm một prototype phù hợp và sao chép nó.

2.4.3 Cách hiện thực

1. Tạo giao diện prototype và khai báo phương thức clone trong đó. Hoặc chỉ cần thêm phương thức này vào tất cả các lớp của một cấu trúc lớp hiện có, nếu có.

2. Một lớp prototype phải xác định hàm tạo thay thế chấp nhận một đối tượng của lớp đó như một đối số. Hàm tạo phải sao chép các giá trị của tất cả các trường được xác định trong lớp từ đối tượng được chuyển vào đối tượng mới được tạo ra. Nếu bạn đang thay đổi một lớp con, bạn phải gọi hàm tạo của lớp cha để cho phép lớp cha xử lý việc sao chép các trường riêng của nó.

Nếu ngôn ngữ lập trình của bạn không hỗ trợ nạp chồng phương thức, bạn sẽ không thể tạo ra một hàm tạo "prototype" riêng biệt. Do đó, việc sao chép dữ liệu của đối tượng vào bản sao mới sẽ phải được thực hiện trong phương thức clone. Tuy nhiên, việc có mã này trong một hàm tạo thông thường an toàn hơn vì đối tượng kết quả được trả về đã được cấu hình đầy đủ ngay sau khi bạn gọi toán tử new.

3. Phương thức sao chép thường chỉ bao gồm một dòng: chạy một toán tử new với phiên bản prototype của hàm tạo. Lưu ý rằng mỗi lớp phải một cách rõ ràng ghi đè phương thức sao chép và sử dụng tên lớp của chính nó cùng với toán tử new. Nếu không, phương thức sao chép có thể tạo ra một đối tượng của một lớp cha.

4. Tuỳ chọn, tạo một bảng đăng ký prototype tập trung để lưu trữ một danh mục các bản prototype được sử dụng thường xuyên.

Bạn có thể triển khai bảng đăng ký như một lớp factory mới hoặc đặt nó trong lớp prototype cơ bản với một phương thức tĩnh để lấy bản prototype. Phương thức này nên tìm kiếm một bản prototype dựa trên tiêu chí tìm kiếm mà mã khách hàng chuyển cho phương thức. Tiêu chí có thể là một thẻ chuỗi đơn giản hoặc một tập hợp phức tạp các tham số tìm kiếm. Sau khi tìm thấy bản prototype phù hợp, bảng đăng ký nên sao chép nó và trả về bản sao cho mã khách hàng.

Cuối cùng, thay thế các cuộc gọi trực tiếp đến hàm tạo của các lớp con bằng các cuộc gọi đến phương thức factory của bảng đăng ký prototype.

2.4.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể sao chép các đối tượng mà không phụ thuộc vào các lớp cụ thể của chúng.
- Bạn có thể loại bỏ mã khởi tạo lặp đi lặp lại để sao chép các bản prototype được xây dựng trước.
- Bạn có thể sản xuất các đối tượng phức tạp một cách thuận tiện hơn.
- Bạn có một phương thức thay thế cho việc kế thừa khi làm việc với cài đặt cấu hình cho các đối tượng phức tạp.

Nhược điểm: Việc sao chép các đối tượng phức tạp có tham chiếu vòng có thể rất phức tạp.

2.4.5 Mối liên hệ với các pattern khác

Nhiều thiết kế bắt đầu bằng cách sử dụng Factory Method (ít phức tạp hơn và có thể tùy chỉnh hơn thông qua các lớp con) và tiến triển về phía Abstract Factory, Prototype, hoặc Builder (linh hoạt hơn, nhưng phức tạp hơn).

Các lớp Abstract Factory thường dựa trên một tập hợp các Factory Methods, nhưng bạn cũng có thể sử dụng Prototype để kết hợp các phương thức trên các lớp này.

Prototype có thể hữu ích khi bạn cần lưu bản sao của các Commands vào lịch sử.

Thiết kế sử dụng nhiều Composite và Decorator thường có thể hưởng lợi từ việc sử dụng Prototype. Áp dụng mẫu thiết kế cho phép bạn sao chép cấu trúc phức tạp thay vì xây dựng lại chúng từ đầu.

Prototype không dựa trên kế thừa, vì vậy nó không có nhược điểm của kế thừa. Tuy nhiên, Prototype yêu cầu một quá trình khởi tạo phức tạp cho đối tượng được sao chép. Factory Method dựa trên kế thừa nhưng không yêu cầu một bước khởi tạo.

Đôi khi Prototype có thể là một lựa chọn đơn giản hơn so với Memento. Điều này hoạt động nếu đối tượng, trạng thái của nó mà bạn muốn lưu vào lịch sử, khá đơn giản và không có liên kết với các tài nguyên bên ngoài, hoặc các liên kết này dễ dàng được thiết lập lại.

Các Abstract Factories, Builders và Prototypes đều có thể được triển khai như Singletons.

2.5 Singleton

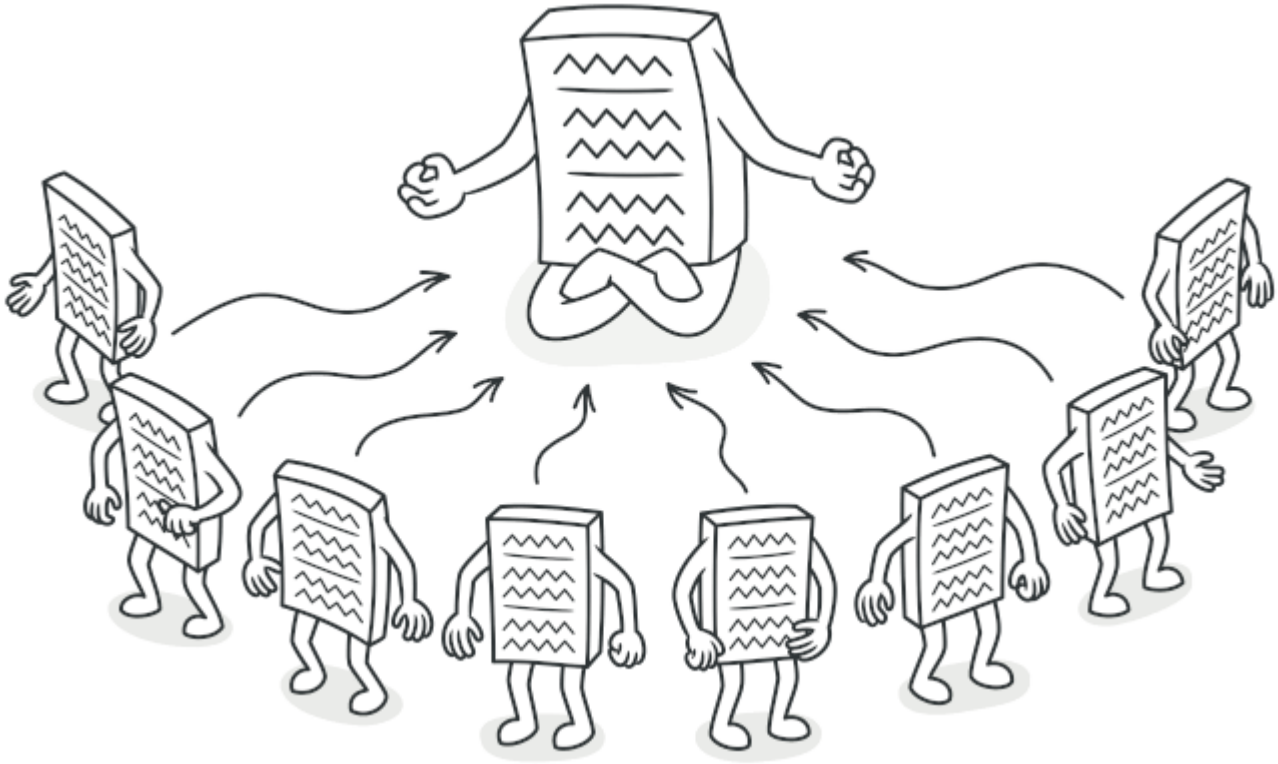
2.5.1 Định nghĩa

Singleton là một mẫu thiết kế sáng tạo cho phép bạn đảm bảo rằng một lớp chỉ có một thể hiện, đồng thời cung cấp một điểm truy cập toàn cầu đến thể hiện này.

2.5.2 Ứng dụng

Sử dụng mẫu Singleton khi một lớp trong chương trình của bạn chỉ nên có một thể hiện duy nhất có sẵn cho tất cả các client; ví dụ, một đối tượng cơ sở dữ liệu duy nhất được chia sẻ bởi các phần khác nhau của chương trình.

Mẫu Singleton vô hiệu hóa tất cả các phương tiện khác để tạo đối tượng của một lớp ngoại trừ phương thức tạo đặc biệt. Phương thức này entweder tạo một đối tượng mới hoặc trả về một đối tượng đã tồn tại nếu nó đã được tạo ra trước đó.



Sử dụng mẫu Singleton khi bạn cần kiểm soát chặt chẽ hơn đối với các biến toàn cầu.

Khác với các biến toàn cầu, mẫu Singleton đảm bảo rằng chỉ có một thể hiện của một lớp. Không có gì, ngoại trừ chính lớp Singleton, có thể thay thế thể hiện được lưu trữ.

Lưu ý rằng bạn luôn có thể điều chỉnh hạn chế này và cho phép tạo bất kỳ số lượng thể hiện Singleton nào. Điều chỉnh duy nhất cần thay đổi là phần thân của phương thức getInstance.

2.5.3 Cách hiện thực

1. Thêm một trường tĩnh private static vào lớp để lưu trữ thể hiện singleton.

2. Khai báo một phương thức tạo public static để lấy thể hiện singleton.

3. Thực hiện "khởi tạo lười biếng" bên trong phương thức tĩnh. Nó sẽ tạo một đối tượng mới trong lần gọi đầu tiên và đặt nó vào trường tĩnh. Phương thức sẽ luôn luôn trả về thể hiện đó trong tất cả các lần gọi sau này.

4. Làm cho hàm tạo của lớp là private. Phương thức tĩnh của lớp vẫn có thể gọi hàm tạo, nhưng không thể được gọi từ các đối tượng khác.

5. Kiểm tra mã khách hàng và thay thế tất cả các cuộc gọi trực tiếp đến hàm tạo của singleton bằng các cuộc gọi đến phương thức tạo tĩnh của nó.

2.5.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể chắc chắn rằng một lớp chỉ có một thể hiện duy nhất.
- Bạn có một điểm truy cập toàn cầu đến thể hiện đó.
- Đối tượng singleton chỉ được khởi tạo khi nó được yêu cầu lần đầu tiên.

Nhược điểm:

- Vi phạm nguyên tắc Single Responsibility. Mẫu thiết kế giải quyết hai vấn đề cùng một lúc.
- Mẫu Singleton có thể ẩn giấu thiết kế tồi, ví dụ, khi các thành phần của chương trình biết quá nhiều về nhau.
- Mẫu thiết kế yêu cầu xử lý đặc biệt trong môi trường đa luồng để đảm bảo rằng nhiều luồng không tạo nhiều thể hiện singleton nhiều lần.
- Có thể khó để kiểm thử đơn vị mã khách hàng của Singleton vì nhiều framework kiểm thử phụ thuộc vào kế thừa khi tạo ra đối tượng giả mạo. Vì hàm tạo của lớp singleton là private và việc ghi đè các phương thức tĩnh là không thể trong hầu hết các ngôn ngữ, bạn sẽ cần nghĩ ra một cách sáng tạo để giả mạo singleton. Hoặc đơn giản là không viết các bài kiểm thử. Hoặc không sử dụng mẫu Singleton.

2.5.5 Mối liên hệ với các pattern khác

Một lớp Facade thường có thể được chuyển đổi thành một Singleton vì một đối tượng facade duy nhất là đủ trong hầu hết các trường hợp.

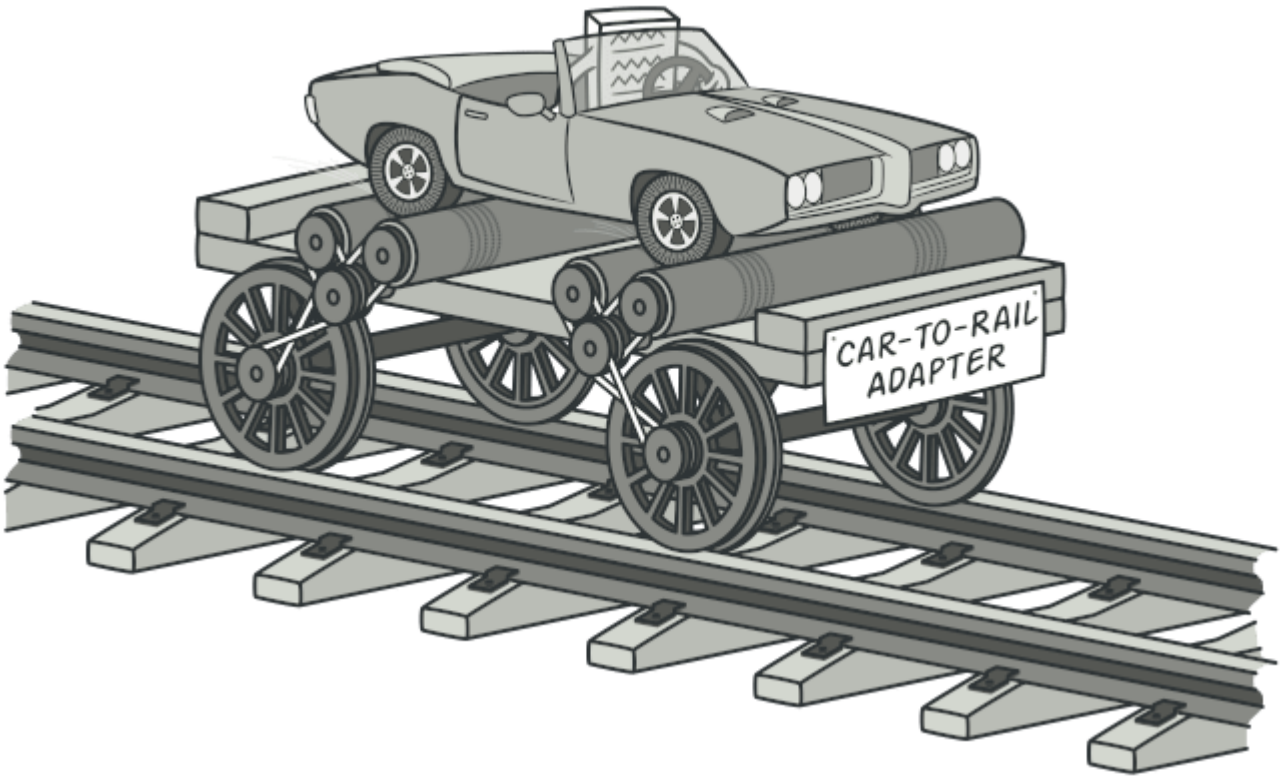
Flyweight sẽ giống như Singleton nếu bạn có cách nào đó giảm tất cả các trạng thái chia sẻ của các đối tượng xuống chỉ còn một đối tượng flyweight. Nhưng có hai sự khác biệt cơ bản giữa các mẫu thiết kế này:

- Chỉ nên có một thể hiện Singleton, trong khi một lớp Flyweight có thể có nhiều thể hiện với các trạng thái nội tại khác nhau.
- Đối tượng Singleton có thể thay đổi. Các đối tượng Flyweight là không thể thay đổi.

Các mẫu thiết kế Abstract Factories, Builders và Prototypes đều có thể được triển khai như Singleton.

3 Structural Design Patterns

3.1 Adapter



3.1.1 Định nghĩa

Adapter là một mẫu thiết kế cấu trúc cho phép các đối tượng có giao diện không tương thích hợp tác.

3.1.2 Ứng dụng

Sử dụng lớp Adapter khi bạn muốn sử dụng một số lớp hiện có, nhưng giao diện của chúng không tương thích với phần còn lại của mã của bạn.

Mẫu thiết kế Adapter cho phép bạn tạo ra một lớp trung gian giúp dịch giữa mã của bạn và một lớp cũ, một lớp từ bên thứ ba hoặc bất kỳ lớp nào có giao diện kỳ lạ.

Sử dụng mẫu này khi bạn muốn tái sử dụng một số lớp con hiện có mà thiếu một số chức năng chung không thể được thêm vào lớp cha.

Bạn có thể mở rộng mỗi lớp con và đưa chức năng bị thiếu vào các lớp con mới. Tuy nhiên, bạn sẽ cần phải nhân bản mã trong tất cả các lớp mới này, điều này rất không tốt.

Một giải pháp tinh tế hơn nhiều sẽ là đưa chức năng bị thiếu vào một lớp adapter. Sau đó, bạn có thể bọc các đối tượng với các tính năng bị thiếu bên trong adapter, đạt được các tính năng cần thiết một cách linh hoạt. Để thực hiện điều này, các lớp mục tiêu phải có một giao diện chung, và trường của adapter nên tuân theo giao diện đó. Phương pháp này rất giống với mẫu thiết kế Decorator.

3.1.3 Cách hiện thực

1. Đảm bảo rằng bạn có ít nhất hai lớp với giao diện không tương thích:

- Một lớp dịch vụ hữu ích, mà bạn không thể thay đổi (thường là bên thứ 3, kế thừa hoặc có nhiều phụ thuộc hiện có).
- Một hoặc nhiều lớp khách hàng có thể hưởng lợi từ việc sử dụng lớp dịch vụ này.

2. Khai báo giao diện khách hàng và mô tả cách các khách hàng giao tiếp với dịch vụ.

3. Tạo lớp adapter và làm cho nó tuân theo giao diện khách hàng. Để tất cả các phương thức trống rỗng cho bây giờ.

4. Thêm một trường vào lớp adapter để lưu trữ tham chiếu đến đối tượng dịch vụ. Thực tiễn thông thường là khởi tạo trường này thông qua constructor, nhưng đôi khi tiện lợi hơn khi truyền nó cho adapter khi gọi các phương thức của nó.

5. Từng bước một, triển khai tất cả các phương thức của giao diện khách hàng trong lớp adapter. Adapter nên ủy quyền hầu hết công việc thực sự cho đối tượng dịch vụ, chỉ xử lý chuyển đổi giao diện hoặc định dạng dữ liệu.

6. Các khách hàng nên sử dụng adapter thông qua giao diện khách hàng. Điều này sẽ cho phép bạn thay đổi hoặc mở rộng các adapter mà không ảnh hưởng đến mã của khách hàng.

3.1.4 Ưu và nhược điểm

Ưu điểm:

- Nguyên tắc Trách nhiệm Đơn lẻ (Single Responsibility Principle). Bạn có thể tách riêng mã chuyển đổi giao diện hoặc dữ liệu khỏi logic nghiệp vụ chính của chương trình.
- Nguyên tắc Đóng/Mở (Open/Closed Principle). Bạn có thể giới thiệu các loại adapter mới vào chương trình mà không làm hỏng mã khách hàng hiện có, miễn là chúng làm việc với các adapter thông qua giao diện khách hàng.

Nhược điểm: Tổng thể độ phức tạp của mã tăng lên vì bạn cần giới thiệu một tập hợp các giao diện và lớp mới. Đôi khi đơn giản hơn là chỉ thay đổi lớp dịch vụ sao cho phù hợp với phần còn lại của mã của bạn.

3.1.5 Mối liên hệ với các pattern khác

Cầu thường được thiết kế từ trước, cho phép bạn phát triển các phần của ứng dụng một cách độc lập với nhau. Ngược lại, Bộ điều hợp thường được sử dụng với một ứng dụng hiện có để làm cho một số lớp không tương thích có thể hoạt động cùng nhau một cách tốt đẹp.

Bộ điều hợp cung cấp một giao diện hoàn toàn khác để truy cập vào một đối tượng hiện có. Ngược lại, với mẫu Trang trí, giao diện hoặc vẫn giữ nguyên hoặc được mở rộng. Ngoài ra, Trang trí hỗ trợ cấu trúc đệ quy, điều không thể thực hiện khi bạn sử dụng Bộ điều hợp.

Với Bộ điều hợp, bạn truy cập vào một đối tượng hiện có thông qua giao diện khác. Với Ủy quyền, giao diện vẫn giữ nguyên. Với Trang trí, bạn truy cập vào đối tượng thông qua một giao diện được nâng cao.

Mặt trước định nghĩa một giao diện mới cho các đối tượng hiện có, trong khi Bộ điều hợp cố gắng làm cho giao diện hiện có có thể sử dụng được. Bộ điều hợp thường bao bọc chỉ một đối tượng, trong khi Mặt trước làm việc với toàn bộ hệ thống con của các đối tượng.

Cầu, Trạng thái, Chiến lược (và ở một mức độ nào đó Bộ điều hợp) có cấu trúc rất giống nhau. Thật vậy, tất cả các mẫu này đều dựa trên sự hợp thành, nghĩa là ủy thác công việc cho các đối tượng khác. Tuy nhiên, chúng đều giải quyết các vấn đề khác nhau. Một mẫu không chỉ là một công thức để cấu trúc mã của bạn theo một cách cụ thể. Nó cũng có thể truyền đạt cho các nhà phát triển khác vấn đề mà mẫu đó giải quyết.

3.2 Bridge

3.2.1 Định nghĩa

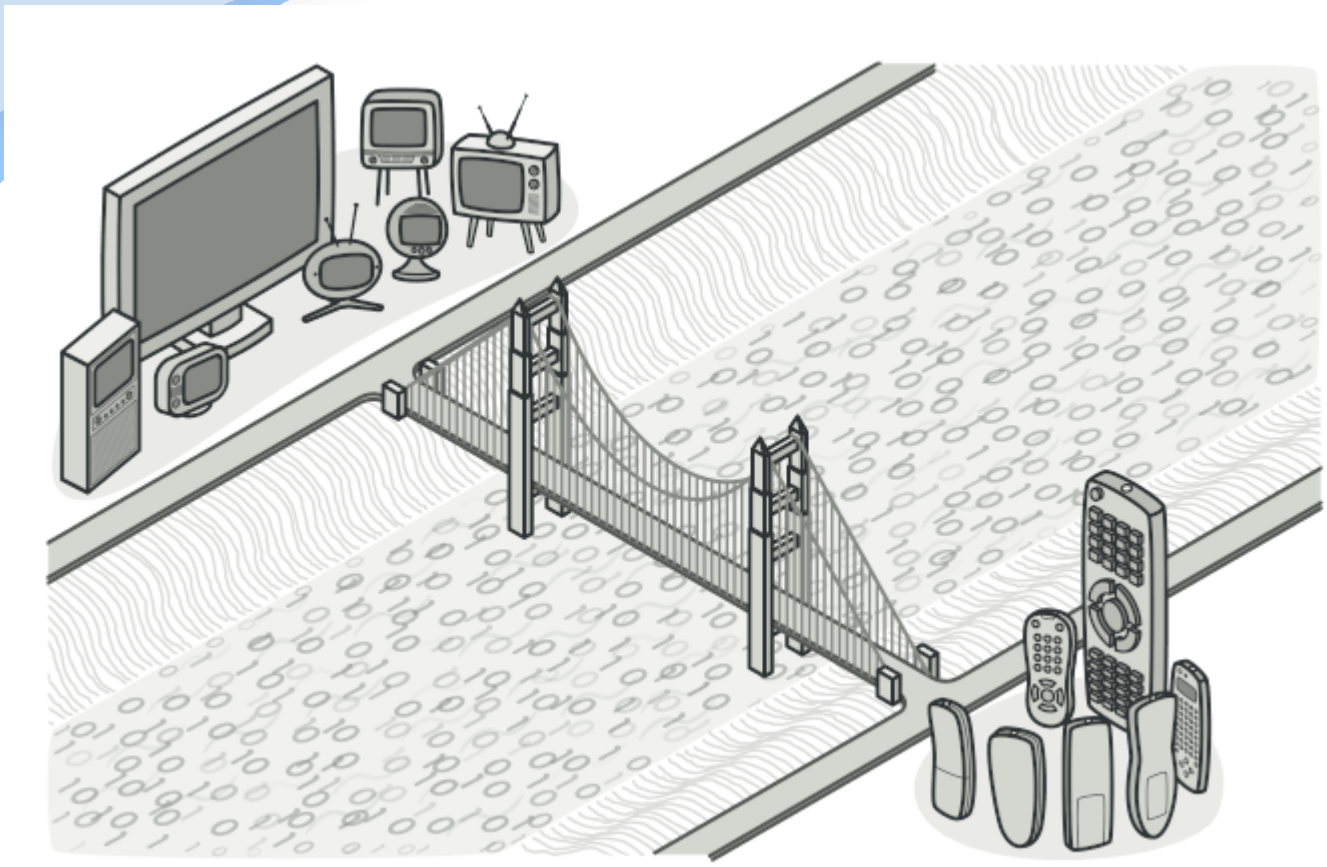
Cầu là một mẫu thiết kế cấu trúc cho phép bạn chia một lớp lớn hoặc một tập hợp các lớp liên quan chặt chẽ thành hai hệ thống phân cấp riêng biệt—trừu tượng và triển khai—có thể được phát triển độc lập với nhau.

3.2.2 Ứng dụng

Sử dụng mẫu Bridge khi bạn muốn chia nhỏ và tổ chức lại một lớp đơn khối có nhiều biến thể của một số chức năng (ví dụ, nếu lớp có thể làm việc với nhiều máy chủ cơ sở dữ liệu khác nhau).

Lớp càng lớn, càng khó hiểu cách nó hoạt động và càng mất nhiều thời gian để thực hiện một thay đổi. Những thay đổi được thực hiện đối với một trong các biến thể của chức năng có thể yêu cầu thực hiện thay đổi trên toàn bộ lớp, điều này thường dẫn đến việc mắc lỗi hoặc không giải quyết được một số tác dụng phụ quan trọng.

Mẫu Bridge cho phép bạn chia lớp đơn khối thành nhiều hệ thống phân cấp lớp. Sau đó, bạn có thể thay đổi các lớp trong mỗi hệ thống phân cấp một cách độc lập với các lớp trong các hệ



thống phân cấp khác. Cách tiếp cận này đơn giản hóa việc bảo trì mã và giảm thiểu rủi ro phá vỡ mã hiện có.

Sử dụng mẫu này khi bạn cần mở rộng một lớp theo nhiều chiều độc lập (không phụ thuộc).

Bridge gợi ý rằng bạn nên tách một hệ thống phân cấp lớp riêng biệt cho mỗi chiều. Lớp ban đầu ủy quyền công việc liên quan cho các đối tượng thuộc về những hệ thống phân cấp đó thay vì tự làm tất cả.

Sử dụng Bridge nếu bạn cần có khả năng chuyển đổi các triển khai tại thời gian chạy.

Mặc dù điều này là tùy chọn, mẫu Bridge cho phép bạn thay thế đối tượng triển khai bên trong sự trừu tượng. Điều này dễ dàng như gán một giá trị mới cho một trường.

Nhân tiện, điều cuối cùng này là lý do chính tại sao nhiều người nhầm lẫn giữa Bridge và mẫu Strategy. Hãy nhớ rằng một mẫu không chỉ là một cách nhất định để cấu trúc các lớp của bạn. Nó cũng có thể truyền đạt ý định và một vấn đề đang được giải quyết.

3.2.3 Cách hiện thực

1. Xác định các chiều trực giao trong các lớp của bạn. Những khái niệm độc lập này có thể là: trừu tượng/nền tảng, miền/cơ sở hạ tầng, giao diện người dùng/phía máy chủ, hoặc giao

diện/triển khai.

2. Xem xét các thao tác mà khách hàng cần và định nghĩa chúng trong lớp trừu tượng cơ bản.

3. Xác định các thao tác có sẵn trên tất cả các nền tảng. Khai báo những thao tác mà lớp trừu tượng cần trong giao diện triển khai chung.

4. Đối với tất cả các nền tảng trong miền của bạn, tạo các lớp triển khai cụ thể, nhưng đảm bảo rằng tất cả đều tuân theo giao diện triển khai.

5. Bên trong lớp trừu tượng, thêm một trường tham chiếu cho kiểu triển khai. Lớp trừu tượng ủy quyền phần lớn công việc cho đối tượng triển khai được tham chiếu trong trường đó.

6. Nếu bạn có nhiều biến thể của logic cấp cao, tạo các lớp trừu tượng tinh chỉnh cho từng biến thể bằng cách mở rộng lớp trừu tượng cơ bản.

7. Mã của khách hàng nên truyền một đối tượng triển khai vào hàm tạo của lớp trừu tượng để liên kết đối tượng triển khai với lớp trừu tượng. Sau đó, khách hàng có thể quên đi việc triển khai và chỉ làm việc với đối tượng trừu tượng.

3.2.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể tạo các lớp và ứng dụng độc lập với nền tảng.
- Mã khách hàng làm việc với các trừu tượng mức cao. Nó không bị phơi bày với các chi tiết của nền tảng.
- Nguyên tắc Mở/Đóng. Bạn có thể giới thiệu các trừu tượng và triển khai mới một cách độc lập với nhau.
- Nguyên tắc Trách nhiệm Đơn lẻ. Bạn có thể tập trung vào logic mức cao trong phần trừu tượng và chi tiết nền tảng trong phần triển khai.

Nhược điểm: Bạn có thể làm cho mã phức tạp hơn bằng cách áp dụng mẫu cho một lớp có độ kết dính cao.

3.2.5 Mối liên hệ với các pattern khác

Cầu nối (Bridge) thường được thiết kế trước, cho phép bạn phát triển các phần của ứng dụng một cách độc lập với nhau. Mặt khác, Bộ chuyển đổi (Adapter) thường được sử dụng với một ứng dụng hiện có để làm cho một số lớp vốn không tương thích có thể hoạt động cùng nhau một cách trơn tru.

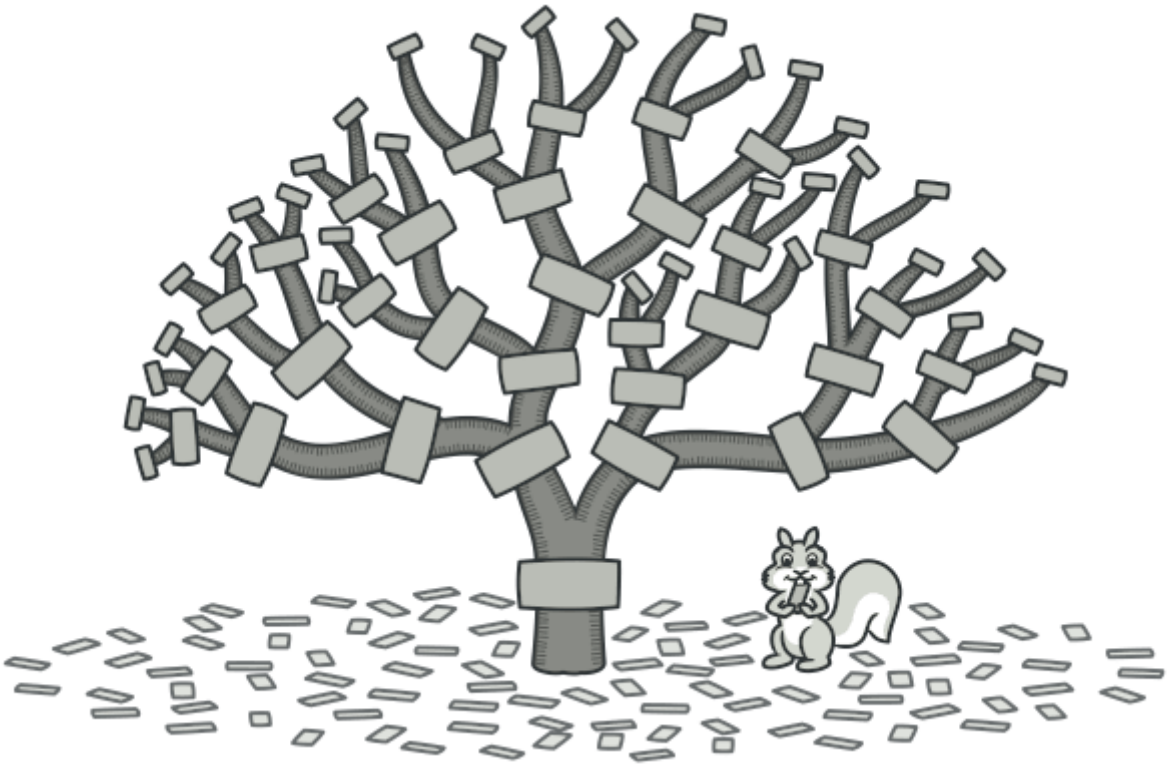
Cầu nối (Bridge), Trạng thái (State), Chiến lược (Strategy) (và đến một mức độ nào đó là Bộ chuyển đổi (Adapter)) có cấu trúc rất giống nhau. Thực tế, tất cả các mẫu thiết kế này đều

dựa trên sự kết hợp, tức là ủy thác công việc cho các đối tượng khác. Tuy nhiên, chúng giải quyết các vấn đề khác nhau. Một mẫu thiết kế không chỉ là một công thức để cấu trúc mã của bạn theo một cách cụ thể. Nó cũng có thể truyền đạt cho các nhà phát triển khác vấn đề mà mẫu thiết kế giải quyết.

Bạn có thể sử dụng Nhà máy trừu tượng (Abstract Factory) cùng với Cầu nối (Bridge). Sự kết hợp này hữu ích khi một số trừu tượng được định nghĩa bởi Cầu nối chỉ có thể hoạt động với các triển khai cụ thể. Trong trường hợp này, Nhà máy trừu tượng có thể bao bọc các mối quan hệ này và giấu đi sự phức tạp khỏi mã khách hàng.

Bạn có thể kết hợp Người xây dựng (Builder) với Cầu nối (Bridge): lớp chỉ đạo đóng vai trò là sự trừu tượng, trong khi các người xây dựng khác nhau đóng vai trò là các triển khai.

3.3 Composite



3.3.1 Định nghĩa

Composite là một mẫu thiết kế cấu trúc cho phép bạn tổ hợp các đối tượng thành cấu trúc cây và sau đó làm việc với các cấu trúc này như là các đối tượng cá nhân.

3.3.2 Ứng dụng

Sử dụng mẫu Composite khi bạn cần triển khai một cấu trúc đối tượng giống như cây.

Mẫu Composite cung cấp cho bạn hai loại phần tử cơ bản chia sẻ một giao diện chung: lá đơn giản và các bộ chứa phức tạp. Một bộ chứa có thể được tạo thành từ cả lá và các bộ chứa khác. Điều này cho phép bạn xây dựng một cấu trúc đối tượng đệ quy lồng nhau giống như một cây.

Sử dụng mẫu khi bạn muốn mã nguồn của client xử lý cả các phần tử đơn giản và phức tạp một cách đồng đều.

Tất cả các phần tử được định nghĩa bởi mẫu Composite đều chia sẻ một giao diện chung. Sử dụng giao diện này, client không cần lo lắng về lớp cụ thể của các đối tượng mà nó làm việc.

3.3.3 Cách hiện thực

1. Đảm bảo rằng mô hình cốt lõi của ứng dụng của bạn có thể được biểu diễn dưới dạng cấu trúc cây. Hãy cố gắng phân tách nó thành các phần tử và bao chứa đơn giản. Nhớ rằng bao chứa phải có khả năng chứa cả các phần tử đơn giản và các bao chứa khác.

2. Khai báo giao diện thành phần với một danh sách các phương thức có ý nghĩa cho cả các thành phần đơn giản và phức tạp.

3. Tạo một lớp lá để biểu diễn các phần tử đơn giản. Một chương trình có thể có nhiều lớp lá khác nhau.

4. Tạo một lớp bao chứa để biểu diễn các phần tử phức tạp. Trong lớp này, cung cấp một trường mảng để lưu trữ các tham chiếu đến các phần tử con. Mảng phải có khả năng lưu trữ cả lá và bao chứa, vì vậy đảm bảo nó được khai báo với kiểu giao diện thành phần.

Trong khi triển khai các phương thức của giao diện thành phần, hãy nhớ rằng một bao chứa được giả định sẽ ủy quyền phần lớn công việc cho các phần tử con.

5. Cuối cùng, xác định các phương thức để thêm và loại bỏ các phần tử con trong bao chứa.

Hãy nhớ rằng các hoạt động này có thể được khai báo trong giao diện thành phần. Điều này sẽ vi phạm Nguyên tắc Chia Interface vì các phương thức sẽ trống trong lớp lá. Tuy nhiên, khách hàng sẽ có thể xử lý tất cả các phần tử một cách bình đẳng, ngay cả khi tạo thành cây.

3.3.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể làm việc với cấu trúc cây phức tạp một cách tiện lợi hơn: sử dụng đa hình và đệ quy để có lợi thế
- Nguyên tắc Mở/Rộng. Bạn có thể giới thiệu các loại phần tử mới vào ứng dụng mà không làm hỏng mã hiện tại, mà hiện tại đã hoạt động với cây đối tượng.

Nhược điểm: Có thể là khó khăn để cung cấp một giao diện chung cho các lớp có chức năng

khác nhau quá nhiều. Trong một số tình huống, bạn sẽ cần phải tổng quát hóa giao diện thành phần, làm cho việc hiểu nó trở nên khó khăn hơn.

3.3.5 Mối liên hệ với các pattern khác

Bạn có thể sử dụng Builder khi tạo cây Composite phức tạp vì bạn có thể lập trình các bước xây dựng của nó để hoạt động đệ quy.

Chain of Responsibility thường được sử dụng kết hợp với Composite. Trong trường hợp này, khi một thành phần lá nhận được một yêu cầu, nó có thể chuyển tiếp qua chuỗi của tất cả các thành phần cha xuống gốc của cây đối tượng.

Bạn có thể sử dụng Iterators để duyệt qua các cây Composite.

Bạn có thể sử dụng Visitor để thực hiện một hoạt động trên toàn bộ cây Composite.

Bạn có thể thực hiện các nút lá chung của cây Composite như Flyweights để tiết kiệm một số RAM.

Composite và Decorator có cấu trúc biểu đồ tương tự vì cả hai đều dựa vào việc hợp thành đệ quy để tổ chức một số lượng đối tượng không giới hạn.

Một Decorator giống như một Composite nhưng chỉ có một thành phần con. Có một sự khác biệt đáng kể khác: Decorator thêm các trách nhiệm bổ sung cho đối tượng được bọc, trong khi Composite chỉ "tích hợp" kết quả của các con của nó.

Tuy nhiên, các mẫu cũng có thể hợp tác: bạn có thể sử dụng Decorator để mở rộng hành vi của một đối tượng cụ thể trong cây Composite.

Thiết kế sử dụng nhiều Composite và Decorator thường có thể được hưởng lợi từ việc sử dụng Prototype. Áp dụng mẫu này cho phép bạn nhân bản cấu trúc phức tạp thay vì xây dựng lại chúng từ đầu.

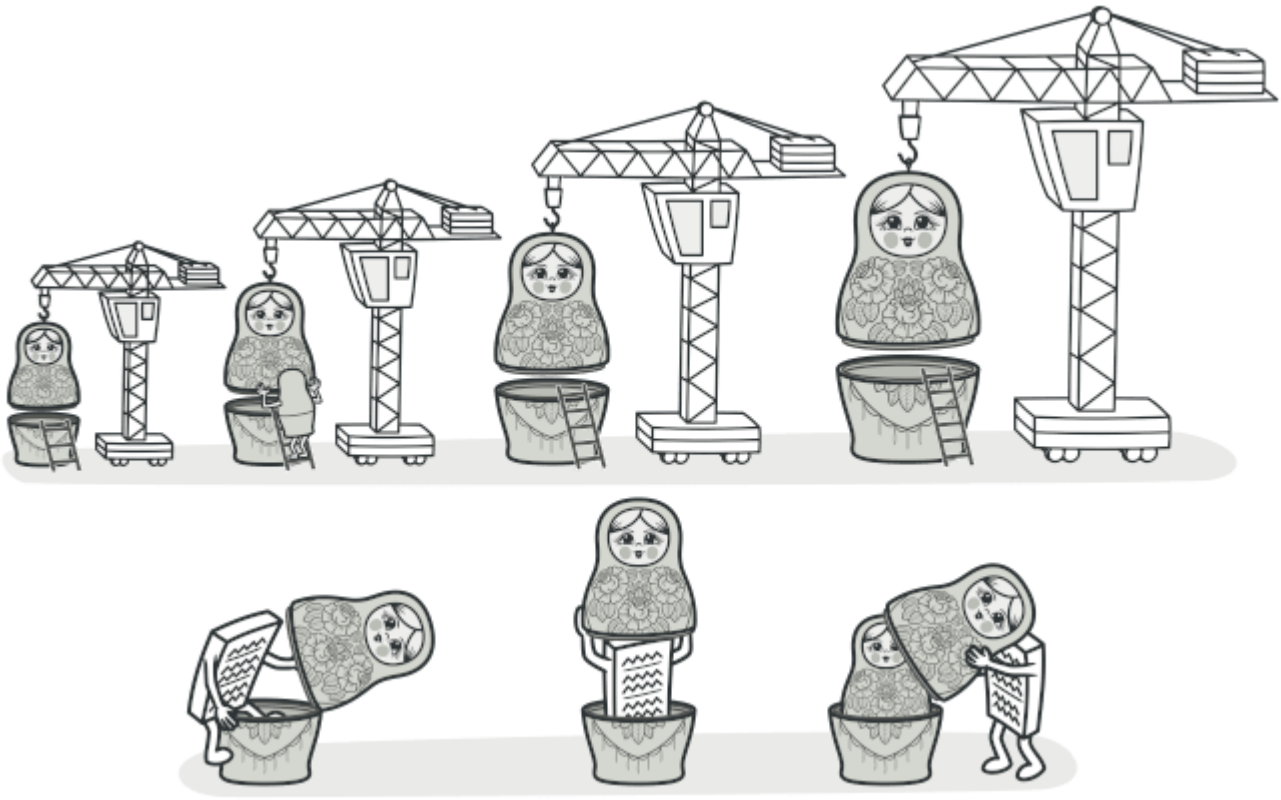
3.4 Decorator

3.4.1 Định nghĩa

Decorator là một mẫu thiết kế cấu trúc cho phép bạn đính kèm các hành vi mới vào các đối tượng bằng cách đặt các đối tượng này bên trong các đối tượng bọc đặc biệt chứa các hành vi.

3.4.2 Ứng dụng

Sử dụng mẫu Decorator khi bạn cần có khả năng gán các hành vi bổ sung cho các đối tượng vào thời gian chạy mà không làm hỏng mã nguồn sử dụng các đối tượng



này.

Decorator cho phép bạn cấu trúc logic kinh doanh của mình thành các lớp, tạo một decorator cho mỗi lớp và tổ hợp các đối tượng với các kết hợp khác nhau của logic này vào thời gian chạy. Mã khách hàng có thể xử lý tất cả các đối tượng này cùng một cách, vì chúng đều tuân theo một giao diện chung.

Sử dụng mẫu khi việc mở rộng hành vi của một đối tượng bằng kế thừa là vụng về hoặc không thể thực hiện được.

Nhiều ngôn ngữ lập trình có từ khóa final có thể được sử dụng để ngăn chặn việc mở rộng thêm của một lớp. Đối với một lớp final, cách duy nhất để tái sử dụng hành vi hiện có sẽ là bọc lớp đó với bọc của riêng bạn, sử dụng mẫu Decorator.

3.4.3 Cách hiện thực

1. Hãy đảm bảo lĩnh vực kinh doanh của bạn có thể được đại diện như một thành phần chính với nhiều lớp tùy chọn trên đó.

2. Tìm ra những phương pháp phổ biến đối với cả thành phần chính và các lớp tùy chọn. Tạo một giao diện thành phần và khai báo những phương pháp đó ở đó.

3. Tạo một lớp thành phần cụ thể và xác định hành vi cơ bản trong đó.

4. Tạo một lớp trang trí cơ bản. Nó nên có một trường để lưu trữ một tham chiếu đến một đối

tượng được bao bọc. Trường nên được khai báo với loại giao diện thành phần để cho phép liên kết với các thành phần cụ thể cũng như trang trí. Trang trí cơ bản phải ủy quyền tất cả công việc cho đối tượng được bao bọc.

5. Hãy đảm bảo tất cả các lớp đều triển khai giao diện thành phần.

6. Tạo trang trí cụ thể bằng cách mở rộng chúng từ trang trí cơ bản. Một trang trí cụ thể phải thực hiện hành vi của nó trước hoặc sau cuộc gọi đến phương thức cha (luôn ủy quyền cho đối tượng được bao bọc).

7. Mã khách hàng phải chịu trách nhiệm tạo ra các trang trí và sắp xếp chúng theo cách mà khách hàng cần.

3.4.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể mở rộng hành vi của một đối tượng mà không cần tạo một lớp con mới.
- Bạn có thể thêm hoặc loại bỏ các trách nhiệm của một đối tượng vào thời gian chạy.
- Bạn có thể kết hợp một số hành vi bằng cách bọc một đối tượng vào nhiều bộ trang trí.
- Nguyên lý Trách nhiệm Đơn lẻ. Bạn có thể chia một lớp monolithic mà thực hiện nhiều biến thể hành vi khả dĩ thành một số lớp nhỏ hơn.

Nhược điểm:

- Rất khó để loại bỏ một bọc cụ thể từ ngăn xếp các bọc.
- Rất khó để triển khai một trang trí một cách sao cho hành vi của nó không phụ thuộc vào thứ tự trong ngăn xếp các trang trí.
- Mã cấu hình ban đầu của các lớp có thể trông khá xấu.

3.4.5 Mối liên hệ với các pattern khác

Adapter cung cấp một giao diện hoàn toàn khác cho việc truy cập vào một đối tượng đã tồn tại. Ngược lại, với mẫu Decorator, giao diện có thể giữ nguyên hoặc được mở rộng. Ngoài ra, Decorator hỗ trợ việc sử dụng đệ quy, điều này không thể xảy ra khi bạn sử dụng Adapter.

Với Adapter, bạn truy cập vào một đối tượng đã tồn tại thông qua giao diện khác. Với Proxy, giao diện vẫn giữ nguyên. Với Decorator, bạn truy cập vào đối tượng thông qua một giao diện được tăng cường.

Chain of Responsibility và Decorator có cấu trúc lớp rất tương tự nhau. Cả hai mẫu đều phụ thuộc vào việc sử dụng đệ quy để truyền thực thi qua một loạt các đối tượng. Tuy nhiên, có một số khác biệt quan trọng.

Các bộ xử lý CoR có thể thực hiện các hoạt động tùy ý độc lập với nhau. Họ cũng có thể ngừng truyền yêu cầu tiếp theo tại bất kỳ điểm nào. Ngược lại, các Decorator khác nhau có thể mở rộng hành vi của đối tượng trong khi vẫn giữ nó phù hợp với giao diện cơ bản. Ngoài ra, các decorator không được phép phá vỡ luồng của yêu cầu.

Composite và Decorator có cấu trúc sơ đồ tương tự vì cả hai đều phụ thuộc vào việc sử dụng đệ quy để tổ chức một số lượng đối tượng không giới hạn.

Một Decorator giống như một Composite nhưng chỉ có một thành phần con. Có một khác biệt quan trọng khác: Decorator thêm các trách nhiệm bổ sung cho đối tượng được bọc, trong khi Composite chỉ "tổng hợp" kết quả của các thành phần con.

Tuy nhiên, các mẫu này cũng có thể hợp tác: bạn có thể sử dụng Decorator để mở rộng hành vi của một đối tượng cụ thể trong cây Composite.

Thiết kế sử dụng nhiều Composite và Decorator có thể thường có lợi từ việc sử dụng Prototype. Áp dụng mẫu cho phép bạn nhân bản các cấu trúc phức tạp thay vì xây dựng lại chúng từ đầu.

Decorator cho phép bạn thay đổi bề ngoài của một đối tượng, trong khi Strategy cho phép bạn thay đổi cách thức hoạt động.

Decorator và Proxy có cấu trúc tương tự, nhưng mục đích rất khác nhau. Cả hai mẫu đều được xây dựng trên nguyên tắc sự phối hợp, trong đó một đối tượng được giao phó một số công việc cho một đối tượng khác. Sự khác biệt là một Proxy thường quản lý vòng đời của đối tượng dịch vụ của nó một cách độc lập, trong khi việc kết hợp của các Decorator luôn luôn được điều khiển bởi khách hàng.

3.5 Facade

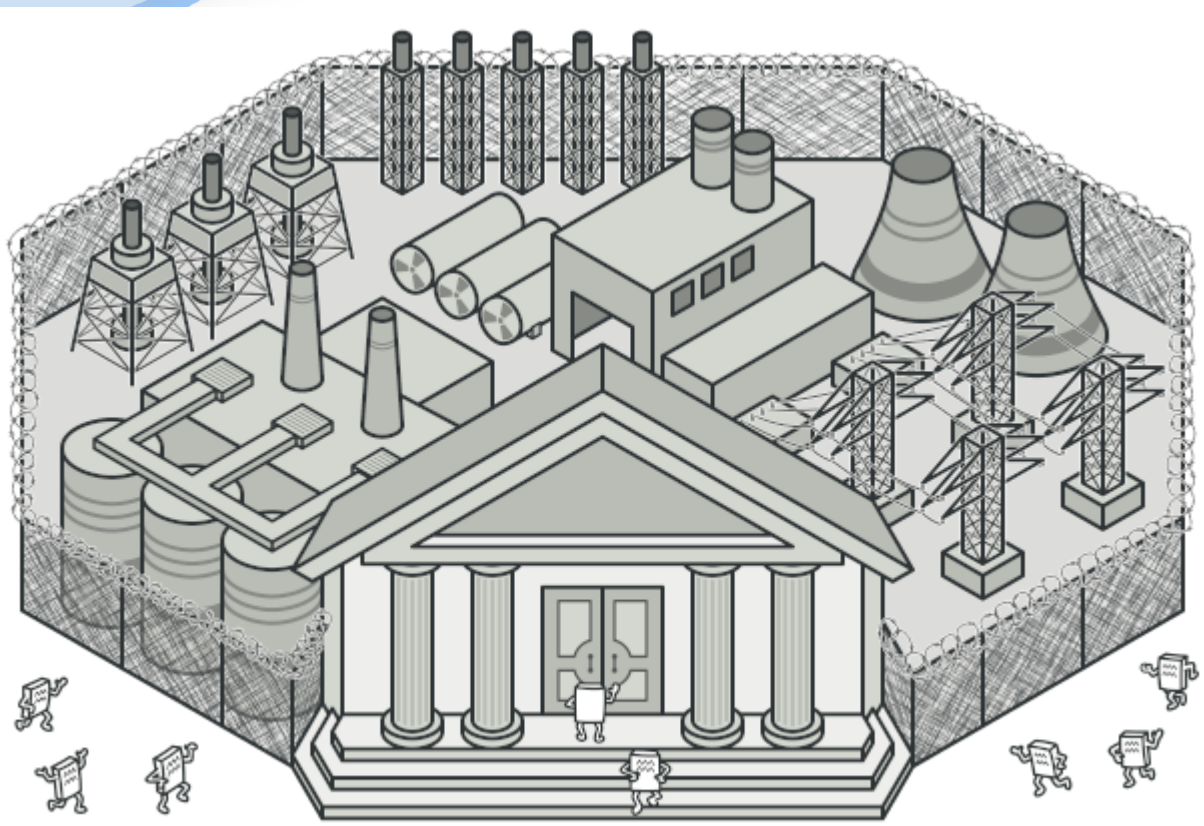
3.5.1 Định nghĩa

Facades là một mẫu thiết kế cấu trúc cung cấp một giao diện đơn giản hóa cho một thư viện, một framework, hoặc bất kỳ bộ lớp phức tạp nào khác.

3.5.2 Ứng dụng

Sử dụng mẫu Facade khi bạn cần có một giao diện hạn chế nhưng đơn giản đến một hệ thống phức tạp.

Thường, các hệ thống con trở nên phức tạp hơn theo thời gian. Ngay cả việc áp dụng các mẫu thiết kế thường dẫn đến việc tạo ra nhiều lớp hơn. Một hệ thống con có thể trở nên linh hoạt hơn và dễ tái sử dụng hơn trong các ngữ cảnh khác nhau, nhưng lượng cấu hình và mã boilerplate mà nó đòi hỏi từ một khách hàng ngày càng lớn hơn. Facade cố gắng khắc phục vấn đề này bằng cách cung cấp một lối tắt đến các tính năng được sử dụng nhiều nhất của hệ thống con phù hợp với hầu hết yêu cầu của khách hàng.



Sử dụng Facade khi bạn muốn cấu trúc một hệ thống con thành các tầng.

Tạo các facade để xác định các điểm nhập cho mỗi tầng của một hệ thống con. Bạn có thể giảm sự liên kết giữa nhiều hệ thống con bằng cách yêu cầu chúng giao tiếp chỉ thông qua các facade.

Ví dụ, hãy quay lại framework chuyển đổi video của chúng ta. Nó có thể được chia thành hai tầng: liên quan đến video và âm thanh. Đối với mỗi tầng, bạn có thể tạo một facade và sau đó làm cho các lớp của mỗi tầng giao tiếp với nhau thông qua các facade đó. Cách tiếp cận này trông rất giống với mẫu Mediator.

3.5.3 Cách hiện thực

1. Kiểm tra xem có thể cung cấp một giao diện đơn giản hơn so với những gì một hệ thống con hiện tại đã cung cấp hay không. Bạn đang đi đúng hướng nếu giao diện này làm cho mã khách độc lập với nhiều lớp của hệ thống con.

2. Khai báo và triển khai giao diện này trong một lớp facade mới. Facade này nên chuyển hướng các cuộc gọi từ mã khách đến các đối tượng phù hợp của hệ thống con. Facade nên chịu trách nhiệm khởi tạo hệ thống con và quản lý vòng đời của nó, trừ khi mã khách đã làm điều này.

3. Để tận dụng tối đa từ mẫu thiết kế này, làm cho tất cả mã khách giao tiếp với hệ thống con chỉ thông qua facade. Bây giờ mã khách được bảo vệ khỏi bất kỳ thay đổi nào trong mã hệ thống con. Ví dụ, khi một hệ thống con được nâng cấp lên phiên bản mới, bạn chỉ cần sửa đổi

mã trong facade.

4. Nếu facade trở nên quá lớn, hãy xem xét việc trích xuất một phần của hành vi của nó thành một lớp facade mới, được tinh chỉnh.

3.5.4 Ưu và nhược điểm

Ưu điểm: Bạn có thể cô lập mã của mình khỏi sự phức tạp của một hệ thống phụ.

Nhược điểm: Một bề ngoài có thể trở thành một đối tượng thần kinh liên kết với tất cả các lớp của ứng dụng.

3.5.5 Mối liên hệ với các pattern khác

Fasade định nghĩa một giao diện mới cho các đối tượng hiện có, trong khi Adapter cố gắng làm cho giao diện hiện có có thể sử dụng được. Adapter thường chỉ bao bọc một đối tượng, trong khi Fasade làm việc với toàn bộ hệ thống các đối tượng.

Abstract Factory có thể phục vụ như một lựa chọn thay thế cho Fasade khi bạn chỉ muốn che giấu cách các đối tượng hệ thống được tạo ra khỏi mã khách hàng.

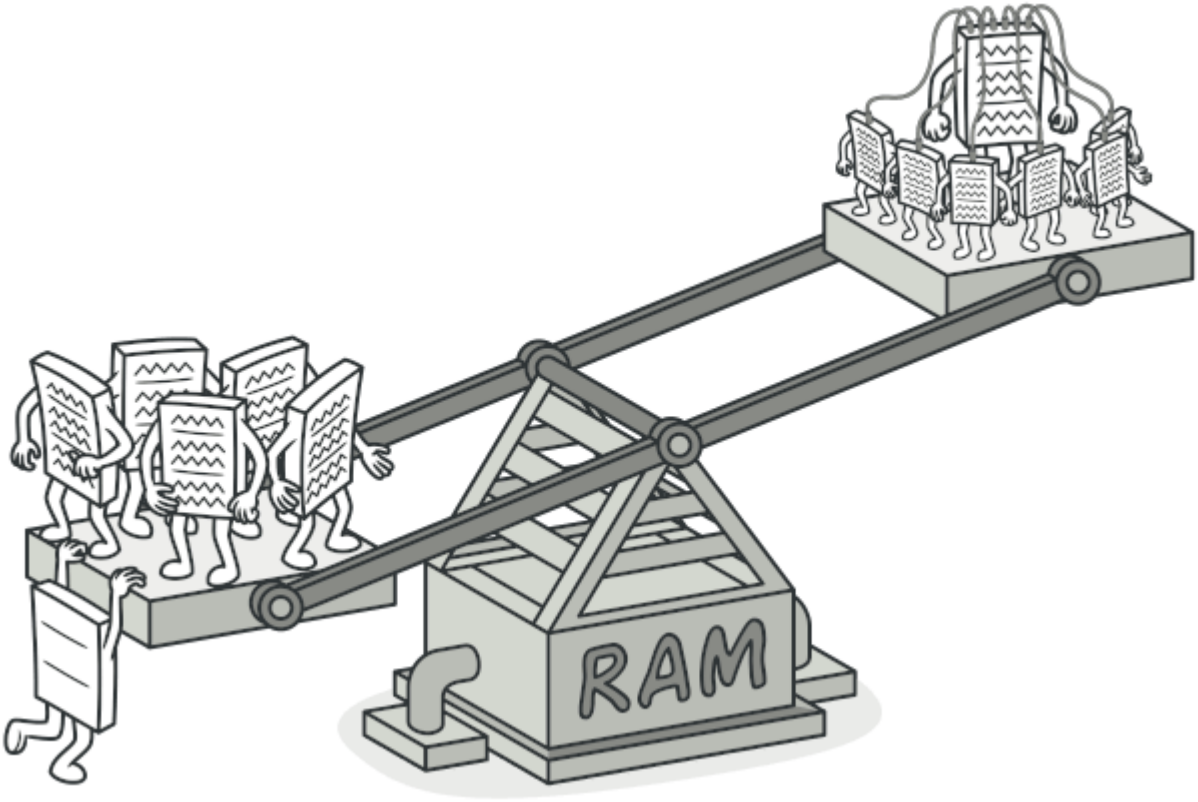
Flyweight chỉ ra cách tạo ra nhiều đối tượng nhỏ, trong khi Fasade chỉ ra cách tạo ra một đối tượng duy nhất đại diện cho toàn bộ hệ thống.

Fasade và Trung gian có công việc tương tự: họ cố gắng tổ chức sự hợp tác giữa nhiều lớp chặt chẽ liên kết.

Fasade định nghĩa một giao diện đơn giản hóa cho một hệ thống các đối tượng, nhưng nó không giới thiệu bất kỳ chức năng mới nào. Hệ thống chính nó không nhận biết Fasade. Các đối tượng trong hệ thống có thể giao tiếp trực tiếp. Trung gian tập trung giao tiếp giữa các thành phần của hệ thống. Các thành phần chỉ biết về đối tượng trung gian và không giao tiếp trực tiếp.

Một lớp Fasade thường có thể được chuyển đổi thành một Singleton vì một đối tượng fasade duy nhất là đủ trong hầu hết các trường hợp.

Fasade tương tự như Proxy ở chỗ cả hai đều đệm một thực thể phức tạp và khởi tạo nó một cách độc lập. Khác với Fasade, Proxy có cùng một giao diện với đối tượng dịch vụ của nó, điều này làm cho chúng có thể thay thế được.



3.6 Flyweight

3.6.1 Định nghĩa

Flyweight là một mẫu thiết kế cấu trúc cho phép bạn sắp xếp nhiều đối tượng hơn vào lượng RAM có sẵn bằng cách chia sẻ các phần chung của trạng thái giữa nhiều đối tượng thay vì giữ tất cả dữ liệu trong mỗi đối tượng.

3.6.2 Ứng dụng

Sử dụng mẫu Flyweight chỉ khi chương trình của bạn phải hỗ trợ một lượng lớn đối tượng mà gần như không vừa vào RAM có sẵn.

Lợi ích của việc áp dụng mẫu phụ thuộc nặng vào cách và nơi mà nó được sử dụng. Nó rất hữu ích khi:

- một ứng dụng cần tạo ra một số lượng lớn đối tượng tương tự
- điều này làm cạn kiệt toàn bộ RAM có sẵn trên thiết bị đích
- các đối tượng chứa trạng thái trùng lặp có thể được trích xuất và chia sẻ giữa nhiều đối tượng

3.6.3 Cách hiện thực

1. Chia các trường của một lớp sẽ trở thành flyweight thành hai phần:

trạng thái nội tại: các trường chứa dữ liệu không thay đổi được sao chép qua nhiều đối tượng

trạng thái ngoại lai: các trường chứa dữ liệu ngữ cảnh duy nhất cho mỗi đối tượng

2. Để lại các trường đại diện cho trạng thái nội tại trong lớp, nhưng đảm bảo rằng chúng không thể thay đổi. Chúng nên nhận các giá trị ban đầu chỉ bên trong hàm khởi tạo.

3. Kiểm tra các phương thức sử dụng các trường của trạng thái ngoại lai. Đối với mỗi trường được sử dụng trong phương thức, giới thiệu một tham số mới và sử dụng nó thay cho trường.

4. Tùy chọn, tạo một lớp factory để quản lý bể flyweights. Nó nên kiểm tra xem flyweight đã tồn tại trước khi tạo một flyweight mới. Một khi factory đã được đặt vào vị trí, các client chỉ được yêu cầu flyweights thông qua nó. Họ nên mô tả flyweight mong muốn bằng cách truyền trạng thái nội tại của nó cho factory.

5. Client phải lưu trữ hoặc tính toán các giá trị của trạng thái ngoại lai (ngữ cảnh) để có thể gọi các phương thức của các đối tượng flyweight. Vì tiện lợi, trạng thái ngoại lai cùng với trường tham chiếu flyweight có thể được di chuyển vào một lớp ngữ cảnh riêng.

3.6.4 Ưu và nhược điểm

Ưu điểm: Bạn có thể tiết kiệm rất nhiều RAM, giả sử chương trình của bạn có rất nhiều đối tượng tương tự.

Nhược điểm:

- Bạn có thể đang đổi RAM bằng các chu kỳ CPU khi một số dữ liệu ngữ cảnh cần được tính toán lại mỗi khi có ai đó gọi một phương thức flyweight.
- Mã trở nên phức tạp hơn nhiều. Các thành viên mới của nhóm luôn tự hỏi tại sao trạng thái của một thực thể được phân tách theo cách đó.

3.6.5 Mối liên hệ với các pattern khác

Bạn có thể triển khai các nút lá chung của cây Composite như Flyweights để tiết kiệm một số RAM.

Flyweight cho thấy cách tạo ra nhiều đối tượng nhỏ, trong khi Facade cho thấy cách tạo ra một đối tượng duy nhất đại diện cho toàn bộ hệ thống con.

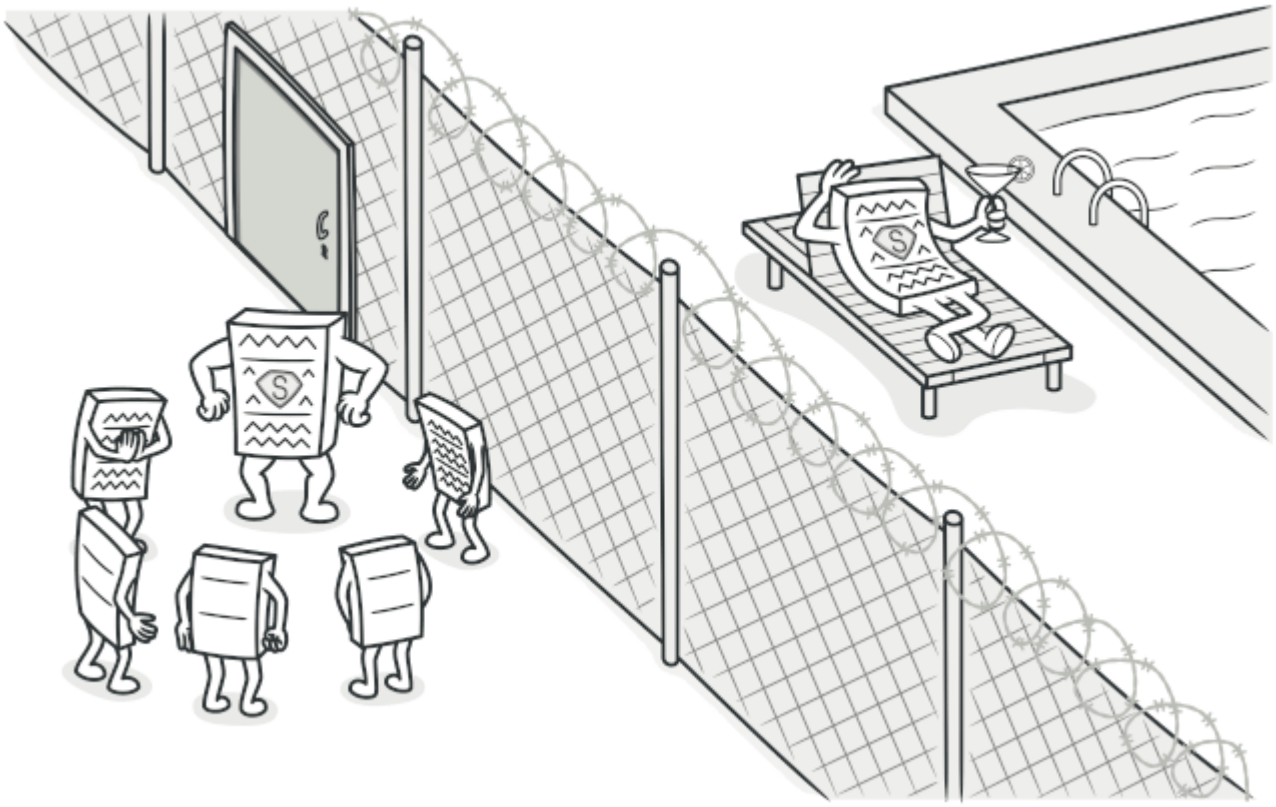
Flyweight sẽ giống như Singleton nếu bạn nào đó quản lý giảm tất cả các trạng thái chia sẻ của các đối tượng xuống chỉ một đối tượng flyweight. Nhưng có hai khác biệt cơ bản giữa các

mẫu này:

Chỉ nên có một thể hiện Singleton, trong khi một lớp Flyweight có thể có nhiều thể hiện với các trạng thái nội tại khác nhau.

Đối tượng Singleton có thể thay đổi. Đối tượng Flyweight là bất biến.

3.7 Proxy



3.7.1 Định nghĩa

Proxy là một mẫu thiết kế cấu trúc cho phép bạn cung cấp một thay thế hoặc nơi giữ chỗ cho một đối tượng khác. Một proxy kiểm soát quyền truy cập vào đối tượng gốc, cho phép bạn thực hiện một điều gì đó trước hoặc sau khi yêu cầu được gửi đến đối tượng gốc.

3.7.2 Ứng dụng

Khởi tạo lười biếng (proxy ảo). Điều này xảy ra khi bạn có một đối tượng dịch vụ nặng nề lãng phí tài nguyên hệ thống bằng cách luôn hoạt động, mặc dù bạn chỉ cần nó đôi khi.

Thay vì tạo đối tượng khi ứng dụng khởi động, bạn có thể trì hoãn việc khởi tạo đối tượng đến một thời điểm khi nó thực sự cần thiết.

Kiểm soát truy cập (proxy bảo vệ). Điều này xảy ra khi bạn chỉ muốn các khách hàng cụ thể mới có thể sử dụng đối tượng dịch vụ; ví dụ, khi các đối tượng của bạn là các phần quan trọng của hệ điều hành và khách hàng là các ứng dụng được khởi chạy khác nhau (bao gồm cả những ứng dụng độc hại).

Proxy có thể chuyển tiếp yêu cầu đến đối tượng dịch vụ chỉ khi thông tin xác thực của khách hàng khớp với một số tiêu chí nào đó.

Thực thi địa phương của một dịch vụ từ xa (proxy từ xa). Điều này xảy ra khi đối tượng dịch vụ được đặt trên một máy chủ từ xa.

Trong trường hợp này, proxy chuyển tiếp yêu cầu của khách hàng qua mạng, xử lý tất cả các chi tiết khó chịu của việc làm việc với mạng.

Ghi lại yêu cầu (proxy ghi log). Điều này xảy ra khi bạn muốn giữ một lịch sử các yêu cầu đối với đối tượng dịch vụ.

Proxy có thể ghi log mỗi yêu cầu trước khi chuyển tiếp nó đến dịch vụ.

Lưu kết quả yêu cầu vào bộ nhớ cache (proxy caching). Điều này xảy ra khi bạn cần lưu kết quả của các yêu cầu từ khách hàng và quản lý vòng đời của bộ nhớ cache này, đặc biệt là nếu kết quả khá lớn.

Proxy có thể thực hiện việc lưu trữ cache cho các yêu cầu lặp lại luôn cho ra cùng một kết quả. Proxy có thể sử dụng các tham số của yêu cầu làm khóa của cache.

Tham chiếu thông minh. Điều này xảy ra khi bạn cần có khả năng bỏ đi một đối tượng nặng nề sau khi không còn khách hàng nào sử dụng nó.

Proxy có thể theo dõi các khách hàng đã nhận tham chiếu đến đối tượng dịch vụ hoặc kết quả của nó. Đôi khi, proxy có thể kiểm tra qua các khách hàng và kiểm tra xem họ vẫn còn hoạt động hay không. Nếu danh sách khách hàng trở nên trống, proxy có thể bỏ qua đối tượng dịch vụ và giải phóng tài nguyên hệ thống ẩn.

Proxy cũng có thể theo dõi xem khách hàng có đã sửa đổi đối tượng dịch vụ hay không. Sau đó, các đối tượng không thay đổi có thể được tái sử dụng bởi các khách hàng khác.

3.7.3 Cách hiện thực

1. Tạo giao diện dịch vụ nếu không có giao diện dịch vụ sẵn có, để tạo sự thay thế linh hoạt giữa các đối tượng proxy và dịch vụ. Việc rút gọn giao diện từ lớp dịch vụ không phải lúc nào cũng khả thi, vì bạn sẽ cần phải thay đổi tất cả các khách hàng của dịch vụ để sử dụng giao diện đó. Kế hoạch B là tạo lớp proxy là một lớp con của lớp dịch vụ, và theo cách này, nó sẽ kế thừa giao diện của dịch vụ.

2. Tạo lớp proxy. Nó nên có một trường để lưu trữ một tham chiếu đến dịch vụ. Thường, các proxy tạo và quản lý toàn bộ vòng đời của dịch vụ của họ. Trong trường hợp hiếm, một dịch vụ được chuyển đến proxy thông qua một hàm tạo bởi khách hàng.

3. Thực hiện các phương thức proxy theo mục đích của chúng. Trong hầu hết các trường hợp, sau khi làm một số công việc, proxy nên ủy quyền công việc cho đối tượng dịch vụ.

4. Xem xét giới thiệu một phương thức tạo ra quyết định liệu khách hàng có nhận được một proxy hay một dịch vụ thực sự. Điều này có thể là một phương thức tĩnh đơn giản trong lớp proxy hoặc một phương thức nhà máy hoàn chỉnh.

5. Xem xét việc triển khai khởi tạo lười biếng cho đối tượng dịch vụ.

3.7.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể kiểm soát đối tượng dịch vụ mà không cần khách hàng biết về điều đó.
- Bạn có thể quản lý vòng đời của đối tượng dịch vụ khi khách hàng không quan tâm đến nó.
- Proxy hoạt động ngay cả khi đối tượng dịch vụ không sẵn sàng hoặc không có sẵn.
- Nguyên tắc Mở / Đóng. Bạn có thể giới thiệu các proxy mới mà không cần thay đổi dịch vụ hoặc khách hàng.

Nhược điểm:

- Mã có thể trở nên phức tạp hơn vì bạn cần giới thiệu nhiều lớp mới.
- Phản hồi từ dịch vụ có thể bị trì hoãn.

3.7.5 Mối liên hệ với các pattern khác

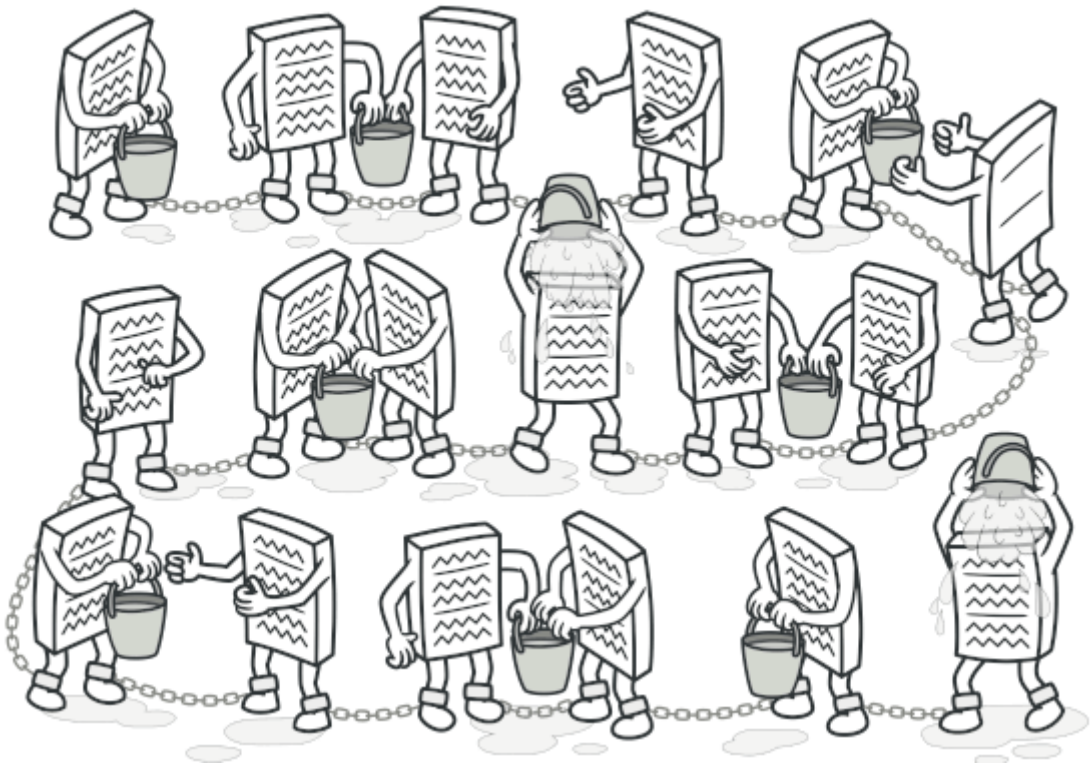
Với Adapter, bạn truy cập một đối tượng hiện có thông qua một giao diện khác. Với Proxy, giao diện vẫn giữ nguyên. Với Decorator, bạn truy cập đối tượng thông qua một giao diện được cải tiến.

Facade giống với Proxy ở chỗ cả hai đều đệm một thực thể phức tạp và khởi tạo nó một cách độc lập. Không giống như Facade, Proxy có cùng giao diện với đối tượng dịch vụ của nó, điều này làm cho chúng có thể thay thế lẫn nhau.

Decorator và Proxy có cấu trúc tương tự nhau, nhưng mục đích rất khác nhau. Cả hai mẫu thiết kế đều dựa trên nguyên tắc cấu thành, trong đó một đối tượng được cho là ủy thác một số công việc cho đối tượng khác. Sự khác biệt là Proxy thường tự quản lý vòng đời của đối tượng dịch vụ của nó, trong khi sự cấu thành của Decorators luôn được kiểm soát bởi khách hàng.

4 Behavioral Design Patterns

4.1 Chain of Responsibility



4.1.1 Định nghĩa

Chain of Responsibility là một mẫu thiết kế hành vi cho phép bạn chuyển các yêu cầu dọc theo một chuỗi các bộ xử lý. Khi nhận được yêu cầu, mỗi bộ xử lý sẽ quyết định xử lý yêu cầu hoặc chuyển nó cho bộ xử lý tiếp theo trong chuỗi.

4.1.2 Ứng dụng

Sử dụng mẫu thiết kế Chain of Responsibility khi chương trình của bạn cần xử lý các loại yêu cầu khác nhau theo nhiều cách khác nhau, nhưng các loại yêu cầu cụ thể và thứ tự của chúng chưa được biết trước.

Mẫu thiết kế này cho phép bạn liên kết nhiều bộ xử lý thành một chuỗi và, khi nhận được yêu cầu, "hỏi" mỗi bộ xử lý xem liệu nó có thể xử lý yêu cầu hay không. Bằng cách này, tất cả các bộ xử lý đều có cơ hội xử lý yêu cầu.

Sử dụng mẫu thiết kế này khi việc thực hiện một số bộ xử lý theo một thứ tự nhất định là điều cần thiết.

Vì bạn có thể liên kết các bộ xử lý trong chuỗi theo bất kỳ thứ tự nào, nên tất cả các yêu cầu sẽ được xử lý qua chuỗi đúng như bạn đã lên kế hoạch.

Sử dụng mẫu CoR khi tập hợp các bộ xử lý và thứ tự của chúng được dự kiến sẽ thay đổi khi chương trình đang chạy.

Nếu bạn cung cấp các phương thức setter cho một trường tham chiếu bên trong các lớp bộ xử lý, bạn sẽ có thể chèn, loại bỏ hoặc sắp xếp lại các bộ xử lý một cách động.

4.1.3 Cách hiện thực

1. Khai báo giao diện bộ xử lý và mô tả chữ ký của một phương thức để xử lý các yêu cầu.

Quyết định cách mà khách hàng sẽ truyền dữ liệu yêu cầu vào phương thức. Cách linh hoạt nhất là chuyển đổi yêu cầu thành một đối tượng và truyền nó vào phương thức xử lý như một đối số.

2. Để loại bỏ mã boilerplate trùng lặp trong các bộ xử lý cụ thể, có thể tạo một lớp bộ xử lý cơ sở trừu tượng, kế thừa từ giao diện bộ xử lý.

Lớp này nên có một trường để lưu trữ tham chiếu đến bộ xử lý tiếp theo trong chuỗi. Cân nhắc việc làm cho lớp này bất biến. Tuy nhiên, nếu bạn dự định thay đổi các chuỗi khi chương trình đang chạy, bạn cần định nghĩa một setter để thay đổi giá trị của trường tham chiếu.

Bạn cũng có thể triển khai hành vi mặc định thuận tiện cho phương thức xử lý, đó là chuyển tiếp yêu cầu đến đối tượng tiếp theo trừ khi không còn đối tượng nào nữa. Các bộ xử lý cụ thể sẽ có thể sử dụng hành vi này bằng cách gọi phương thức cha.

3. Lần lượt tạo các lớp con bộ xử lý cụ thể và triển khai các phương thức xử lý của chúng. Mỗi bộ xử lý nên đưa ra hai quyết định khi nhận được một yêu cầu:

Liệu nó sẽ xử lý yêu cầu.

Liệu nó sẽ chuyển yêu cầu tiếp theo trong chuỗi.

4. Khách hàng có thể tự lắp ráp các chuỗi hoặc nhận các chuỗi được lắp ráp sẵn từ các đối tượng khác. Trong trường hợp sau, bạn phải triển khai một số lớp nhà máy để xây dựng các chuỗi theo cấu hình hoặc cài đặt môi trường.

5. Khách hàng có thể kích hoạt bất kỳ bộ xử lý nào trong chuỗi, không chỉ bộ xử lý đầu tiên. Yêu cầu sẽ được chuyển dọc theo chuỗi cho đến khi một bộ xử lý từ chối chuyển tiếp yêu cầu hoặc cho đến khi nó đến cuối chuỗi.

6. Do tính chất động của chuỗi, khách hàng nên sẵn sàng xử lý các tình huống sau:

Chuỗi có thể chỉ bao gồm một liên kết duy nhất.

Một số yêu cầu có thể không đến được cuối chuỗi.

Một số yêu cầu khác có thể đến cuối chuỗi mà không được xử lý.

4.1.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể kiểm soát thứ tự xử lý yêu cầu.
- Nguyên tắc Trách nhiệm Đơn lẻ. Bạn có thể tách rời các lớp gọi thao tác khỏi các lớp thực hiện thao tác.
- Nguyên tắc Mở/Đóng. Bạn có thể giới thiệu các bộ xử lý mới vào ứng dụng mà không làm hỏng mã khách hàng hiện có.

Nhược điểm: Một số yêu cầu có thể không được xử lý

4.1.5 Mối liên hệ với các pattern khác

Chain of Responsibility, Command, Mediator và Observer giải quyết các cách kết nối khác nhau giữa các bộ gửi và bộ nhận yêu cầu:

Chain of Responsibility chuyển một yêu cầu tuần tự dọc theo một chuỗi động các bộ nhận tiềm năng cho đến khi một trong số chúng xử lý yêu cầu đó.

Command thiết lập các kết nối đơn hướng giữa các bộ gửi và bộ nhận.

Mediator loại bỏ các kết nối trực tiếp giữa các bộ gửi và bộ nhận, buộc chúng giao tiếp gián tiếp qua một đối tượng trung gian.

Observer cho phép các bộ nhận đăng ký và hủy đăng ký nhận yêu cầu một cách động.

Chain of Responsibility thường được sử dụng cùng với Composite. Trong trường hợp này, khi một thành phần lá nhận được yêu cầu, nó có thể chuyển yêu cầu qua chuỗi của tất cả các thành phần cha cho đến gốc của cây đối tượng.

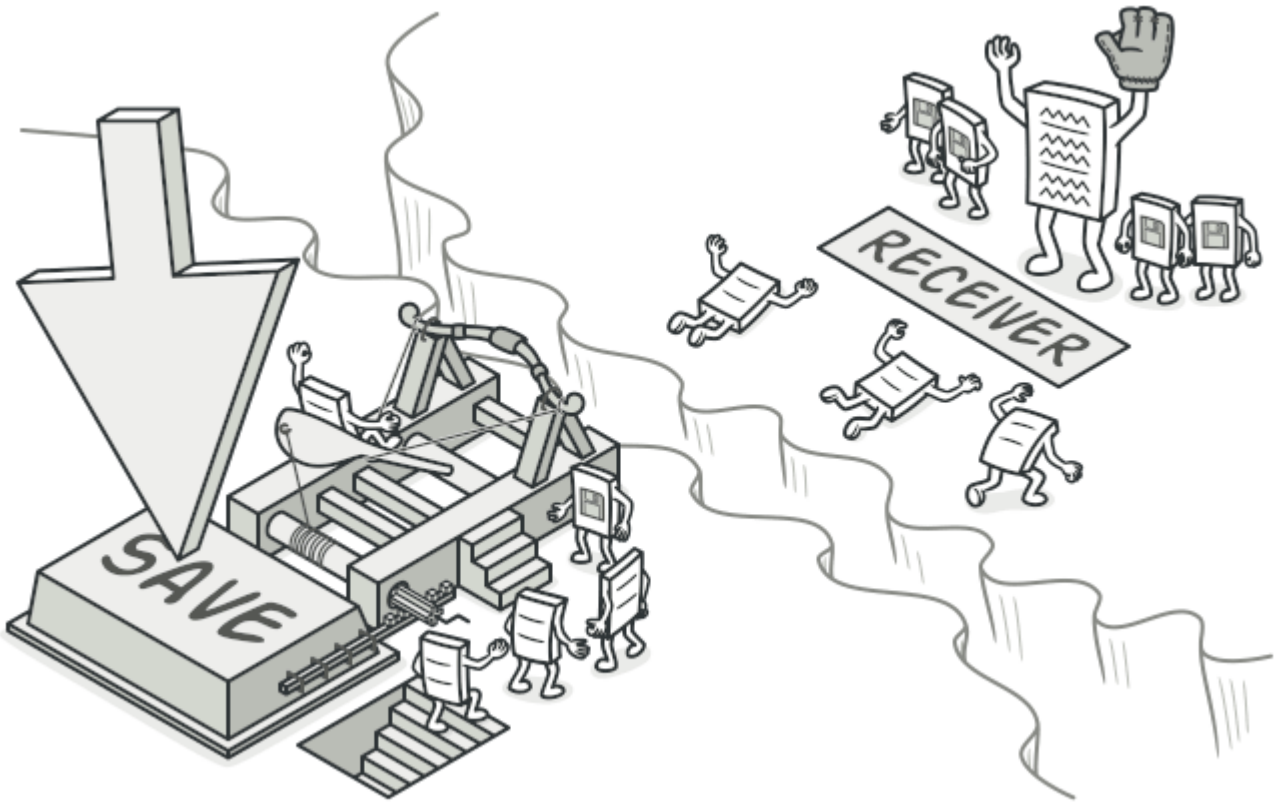
Các bộ xử lý trong Chain of Responsibility có thể được triển khai dưới dạng Commands. Trong trường hợp này, bạn có thể thực hiện nhiều thao tác khác nhau trên cùng một đối tượng ngữ cảnh, được đại diện bởi một yêu cầu.

Tuy nhiên, có một cách tiếp cận khác, trong đó yêu cầu tự nó là một đối tượng Command. Trong trường hợp này, bạn có thể thực hiện cùng một thao tác trong một chuỗi các ngữ cảnh khác nhau được liên kết thành một chuỗi.

Chain of Responsibility và Decorator có cấu trúc lớp rất giống nhau. Cả hai mẫu thiết kế đều dựa vào cấu thành đệ quy để chuyển tiếp thực thi qua một chuỗi các đối tượng. Tuy nhiên, có một số khác biệt quan trọng.

Các bộ xử lý CoR có thể thực hiện các thao tác tùy ý độc lập với nhau. Chúng cũng có thể ngừng chuyển tiếp yêu cầu tại bất kỳ điểm nào. Ngược lại, các Decorator khác nhau có thể mở rộng hành vi của đối tượng trong khi vẫn giữ cho nó nhất quán với giao diện cơ bản. Ngoài ra, các Decorator không được phép phá vỡ luồng của yêu cầu.

4.2 Command



4.2.1 Định nghĩa

Command là một mẫu thiết kế hành vi biến một yêu cầu thành một đối tượng độc lập chứa tất cả thông tin về yêu cầu. Sự biến đổi này cho phép bạn truyền các yêu cầu như các đối số của phương thức, trì hoãn hoặc xếp hàng thực hiện yêu cầu, và hỗ trợ các thao tác có thể hoàn tác.

4.2.2 Ứng dụng

Sử dụng mẫu thiết kế Command khi bạn muốn tham số hóa các đối tượng với các thao tác.

Mẫu thiết kế Command có thể biến một lời gọi phương thức cụ thể thành một đối tượng độc lập. Sự thay đổi này mở ra nhiều ứng dụng thú vị: bạn có thể truyền các lệnh như các đối số

của phương thức, lưu trữ chúng bên trong các đối tượng khác, thay đổi các lệnh liên kết khi chương trình đang chạy, v.v.

Đây là một ví dụ: bạn đang phát triển một thành phần GUI như menu ngữ cảnh, và bạn muốn người dùng có thể cấu hình các mục menu để kích hoạt các thao tác khi người dùng cuối nhấp vào một mục.

Sử dụng mẫu thiết kế Command khi bạn muốn xếp hàng các thao tác, lên lịch thực hiện chúng, hoặc thực hiện chúng từ xa.

Giống như bất kỳ đối tượng nào khác, một lệnh có thể được tuần tự hóa, nghĩa là chuyển đổi nó thành một chuỗi có thể dễ dàng ghi vào tệp hoặc cơ sở dữ liệu. Sau đó, chuỗi này có thể được khôi phục thành đối tượng lệnh ban đầu. Do đó, bạn có thể trì hoãn và lên lịch thực hiện lệnh. Nhưng còn nhiều hơn thế! Tương tự, bạn có thể xếp hàng, ghi lại hoặc gửi các lệnh qua mạng.

Sử dụng mẫu thiết kế Command khi bạn muốn triển khai các thao tác có thể hoàn tác.

Mặc dù có nhiều cách để triển khai undo/redo, mẫu thiết kế Command có lẽ là phổ biến nhất.

Để có thể hoàn tác các thao tác, bạn cần triển khai lịch sử của các thao tác đã thực hiện. Lịch sử lệnh là một ngăn xếp chứa tất cả các đối tượng lệnh đã thực thi cùng với các bản sao lưu liên quan của trạng thái ứng dụng.

Phương pháp này có hai nhược điểm. Thứ nhất, không dễ để lưu trạng thái của ứng dụng vì một số trạng thái có thể là riêng tư. Vấn đề này có thể được giảm thiểu bằng mẫu thiết kế Memento.

Thứ hai, các bản sao lưu trạng thái có thể tiêu tốn khá nhiều RAM. Do đó, đôi khi bạn có thể sử dụng một triển khai thay thế: thay vì khôi phục trạng thái trước đó, lệnh thực hiện thao tác ngược lại. Thao tác ngược lại cũng có giá trị: nó có thể khó hoặc thậm chí không thể thực hiện.

4.2.3 Cách hiện thực

1. Khai báo giao diện lệnh với một phương thức thực thi duy nhất.

2. Bắt đầu trích xuất các yêu cầu vào các lớp lệnh cụ thể triển khai giao diện lệnh. Mỗi lớp phải có một tập hợp các trường để lưu trữ các đối số yêu cầu cùng với một tham chiếu đến đối tượng nhận thực tế. Tất cả các giá trị này phải được khởi tạo thông qua hàm khởi tạo của lệnh.

3. Xác định các lớp sẽ đóng vai trò là bộ gửi. Thêm các trường để lưu trữ các lệnh vào các lớp này. Các bộ gửi nên giao tiếp với các lệnh của chúng chỉ thông qua giao diện lệnh. Các bộ gửi thường không tự tạo các đối tượng lệnh mà thay vào đó nhận chúng từ mã của khách hàng.

4. Thay đổi các bộ gửi để chúng thực thi lệnh thay vì gửi yêu cầu trực tiếp đến bộ nhận.

5. Khách hàng nên khởi tạo các đối tượng theo thứ tự sau:

Tạo các bộ nhận.

Tạo các lệnh và liên kết chúng với các bộ nhận nếu cần.

Tạo các bộ gửi và liên kết chúng với các lệnh cụ thể.

4.2.4 Ưu và nhược điểm

Ưu điểm:

- Nguyên tắc Trách nhiệm Đơn lẻ. Bạn có thể tách rời các lớp gọi thực hiện thao tác khỏi các lớp thực hiện các thao tác này.
- Nguyên tắc Mở/Đóng. Bạn có thể giới thiệu các lệnh mới vào ứng dụng mà không làm hỏng mã khách hàng hiện tại.
- Bạn có thể triển khai hoạt động hoàn tác/phục hồi.
- Bạn có thể triển khai thực thi hoạt động lệch (deferred execution).
- Bạn có thể tổ chức một tập hợp các lệnh đơn giản thành một lệnh phức tạp.

Nhược điểm: Mã có thể trở nên phức tạp hơn vì bạn đang giới thiệu một lớp hoàn toàn mới giữa bộ gửi và bộ nhận.

4.2.5 Mối liên hệ với các pattern khác

Mẫu thiết kế Chain of Responsibility, Command, Mediator và Observer giải quyết các cách kết nối khác nhau giữa bộ gửi và bộ nhận yêu cầu:

Chain of Responsibility chuyển một yêu cầu tuần tự dọc theo một chuỗi động các bộ nhận tiềm năng cho đến khi một trong số chúng xử lý nó. Command thiết lập các kết nối một chiều giữa bộ gửi và bộ nhận. Mediator loại bỏ các kết nối trực tiếp giữa bộ gửi và bộ nhận, buộc chúng giao tiếp gián tiếp qua một đối tượng trung gian. Observer cho phép các bộ nhận đăng ký và hủy đăng ký động nhận yêu cầu.

Các bộ xử lý trong Chain of Responsibility có thể được triển khai dưới dạng Commands. Trong trường hợp này, bạn có thể thực hiện nhiều thao tác khác nhau trên cùng một đối tượng ngữ cảnh, được đại diện bởi một yêu cầu.

Tuy nhiên, còn một phương pháp khác, trong đó yêu cầu chính nó là một đối tượng Command. Trong trường hợp này, bạn có thể thực hiện cùng một thao tác trong một loạt các ngữ cảnh khác nhau được liên kết thành một chuỗi.

Bạn có thể sử dụng Command và Memento cùng nhau khi triển khai "undo". Trong trường hợp này, các lệnh chịu trách nhiệm thực hiện các hoạt động khác nhau trên một đối tượng mục

tiêu, trong khi các mementos lưu trạng thái của đối tượng đó ngay trước khi một lệnh được thực thi.

Command và Strategy có vẻ giống nhau vì bạn có thể sử dụng cả hai để tham số hóa một đối tượng với một hành động nào đó. Tuy nhiên, chúng có mục đích rất khác nhau.

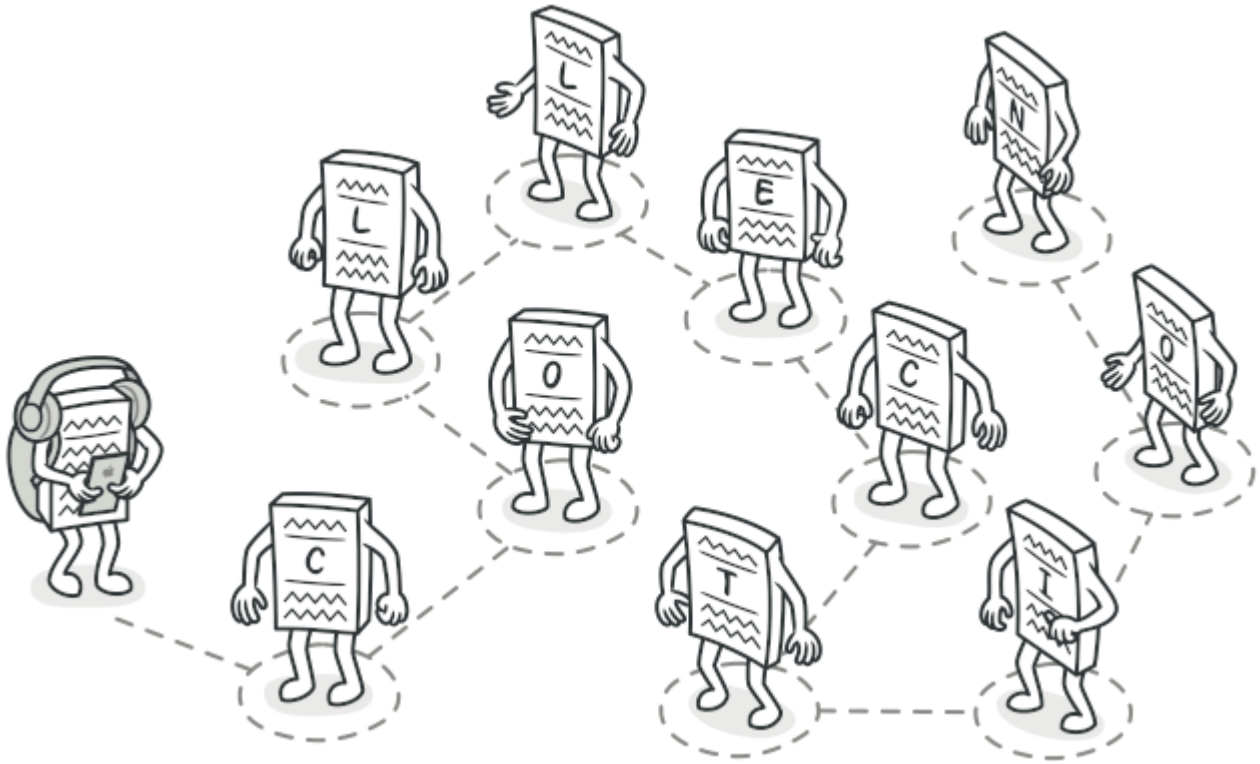
Bạn có thể sử dụng Command để chuyển đổi bất kỳ hoạt động nào thành một đối tượng. Các tham số của hoạt động trở thành các trường của đối tượng đó. Việc chuyển đổi cho phép bạn trì hoãn việc thực thi của hoạt động, xếp hàng nó, lưu trữ lịch sử của các lệnh, gửi lệnh đến dịch vụ từ xa, v.v.

Ngược lại, Strategy thường mô tả các cách khác nhau để làm cùng một việc, cho phép bạn thay đổi các thuật toán này trong một lớp ngữ cảnh duy nhất.

Prototype có thể hữu ích khi bạn cần lưu các bản sao của các lệnh vào lịch sử.

Bạn có thể xem Visitor như một phiên bản mạnh mẽ của mẫu thiết kế Command. Các đối tượng của nó có thể thực hiện các hoạt động trên các đối tượng khác nhau của các lớp khác nhau.

4.3 Iterator



4.3.1 Định nghĩa

Iterator là một mẫu thiết kế hành vi cho phép bạn duyệt qua các phần tử của một tập hợp mà không tiết lộ cấu trúc bên dưới của nó (danh sách, ngăn xếp, cây, v.v.).

4.3.2 Ứng dụng

Sử dụng mẫu Iterator khi bộ sưu tập của bạn có một cấu trúc dữ liệu phức tạp nhưng bạn muốn che giấu sự phức tạp của nó khỏi khách hàng (entitie) (hoặc vì lý do tiện lợi hoặc an ninh).

Bộ lặp (iterator) đóng gói các chi tiết của việc làm việc với một cấu trúc dữ liệu phức tạp, cung cấp cho khách hàng một số phương thức đơn giản để truy cập các phần tử của bộ sưu tập. Mặc dù cách tiếp cận này rất tiện lợi cho khách hàng, nó cũng bảo vệ bộ sưu tập khỏi các hành động vô tình hoặc độc hại mà khách hàng có thể thực hiện nếu làm việc trực tiếp với bộ sưu tập.

Sử dụng mẫu để giảm sự trùng lặp của mã duyệt qua ứng dụng của bạn.

Mã của các thuật toán duyệt qua không phổ biến thường rất phồng phéo. Khi đặt trong logic kinh doanh của một ứng dụng, nó có thể làm mờ trách nhiệm của mã gốc và làm cho nó ít dễ bảo trì hơn. Di chuyển mã duyệt qua các bộ lặp được chỉ định có thể giúp bạn làm cho mã của ứng dụng trở nên gọn gàng và sạch sẽ hơn.

Sử dụng Iterator khi bạn muốn mã của mình có thể duyệt qua các cấu trúc dữ liệu khác nhau hoặc khi các loại cấu trúc này không được biết trước.

Mẫu cung cấp một vài giao diện chung cho cả bộ sưu tập và bộ lặp. Với việc mã của bạn hiện đang sử dụng các giao diện này, nó vẫn hoạt động nếu bạn truyền nó các loại bộ sưu tập và bộ lặp khác nhau mà triển khai các giao diện này.

4.3.3 Cách hiện thực

1. Khai báo giao diện bộ lặp (iterator). Ít nhất, nó phải có một phương thức để lấy phần tử tiếp theo từ một bộ sưu tập. Nhưng vì tiện lợi, bạn có thể thêm một vài phương thức khác, chẳng hạn như lấy phần tử trước đó, theo dõi vị trí hiện tại và kiểm tra cuối của việc lặp.

2. Khai báo giao diện bộ sưu tập và mô tả một phương thức để lấy các bộ lặp. Kiểu trả về nên bằng với kiểu của giao diện bộ lặp. Bạn có thể khai báo các phương thức tương tự nếu bạn dự định có một vài nhóm bộ lặp riêng biệt.

3. Thực hiện các lớp bộ lặp cụ thể cho các bộ sưu tập mà bạn muốn duyệt qua bằng bộ lặp. Một đối tượng bộ lặp phải được liên kết với một bộ sưu tập duy nhất. Thông thường, liên kết này được thiết lập thông qua hàm tạo của bộ lặp.

4. Thực hiện giao diện bộ sưu tập trong các lớp bộ sưu tập của bạn. Ý tưởng chính là cung cấp

cho khách hàng một phím tắt để tạo ra các bộ lặp, được tùy chỉnh cho một lớp bộ sưu tập cụ thể. Đối tượng bộ sưu tập phải truyền chính nó vào hàm tạo của bộ lặp để thiết lập một liên kết giữa chúng.

5. Kiểm tra mã khách hàng để thay thế tất cả mã duyệt qua bộ sưu tập bằng việc sử dụng các bộ lặp. Khách hàng lấy một đối tượng bộ lặp mới mỗi khi cần lặp qua các phần tử của bộ sưu tập.

4.3.4 Ưu và nhược điểm

Ưu điểm:

- Nguyên tắc Single Responsibility. Bạn có thể làm sạch mã khách hàng và các bộ sưu tập bằng cách trích xuất các thuật toán duyệt phồng phéo vào các lớp riêng biệt.
- Nguyên tắc Mở/Closed. Bạn có thể triển khai các loại mới của các bộ sưu tập và bộ lặp và truyền chúng vào mã hiện có mà không làm hỏng bất cứ điều gì.
- Bạn có thể lặp qua cùng một bộ sưu tập một cách song song vì mỗi đối tượng bộ lặp chứa trạng thái lặp riêng của nó.
- Vì cùng một lý do, bạn có thể trì hoãn một lần lặp và tiếp tục nó khi cần thiết.

4.3.5 Mối liên hệ với các pattern khác

Bạn có thể sử dụng Bộ lặp (Iterator) để duyệt qua các cây Composite.

Bạn có thể sử dụng Phương thức Factory kèm với Bộ lặp để cho phép các lớp con của bộ sưu tập trả về các loại bộ lặp khác nhau mà tương thích với các bộ sưu tập.

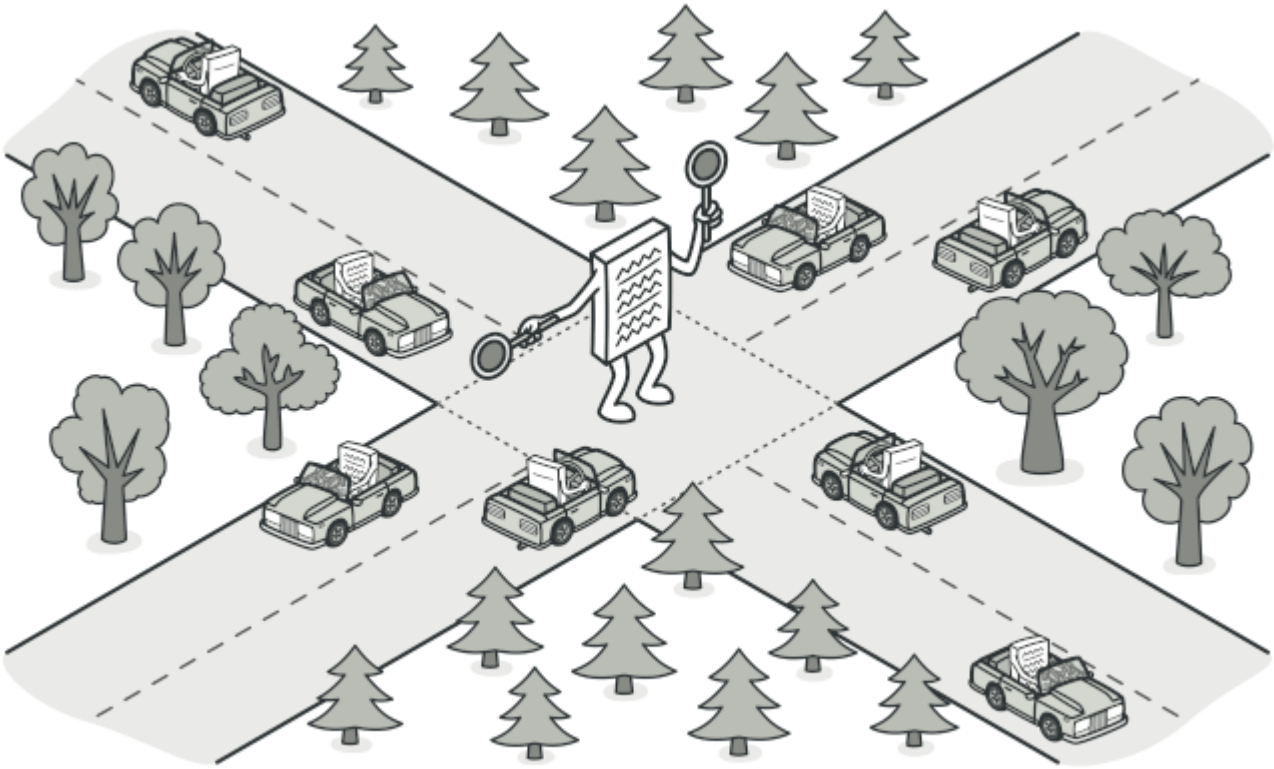
Bạn có thể sử dụng Memento kèm với Bộ lặp để ghi lại trạng thái lặp hiện tại và quay trở lại nó nếu cần thiết.

Bạn có thể sử dụng Visitor kèm với Bộ lặp để duyệt qua một cấu trúc dữ liệu phức tạp và thực hiện một số thao tác trên các phần tử của nó, ngay cả khi chúng có các lớp khác nhau.

4.4 Mediator

4.4.1 Định nghĩa

Mediator là một mẫu thiết kế hành vi cho phép bạn giảm thiểu sự phụ thuộc hỗn loạn giữa các đối tượng. Mẫu này hạn chế việc giao tiếp trực tiếp giữa các đối tượng và buộc chúng phải hợp tác chỉ thông qua một đối tượng trung gian.



4.4.2 Ứng dụng

Sử dụng mẫu Mediator khi việc thay đổi một số lớp là khó khăn do chúng liên kết chặt chẽ với một nhóm lớp khác.

Mẫu này cho phép bạn trích xuất tất cả các mối quan hệ giữa các lớp vào một lớp riêng biệt, cách ly bất kỳ thay đổi nào đối với một thành phần cụ thể khỏi phần còn lại của các thành phần.

Sử dụng mẫu khi bạn không thể tái sử dụng một thành phần trong một chương trình khác vì nó quá phụ thuộc vào các thành phần khác.

Sau khi áp dụng Mediator, các thành phần riêng lẻ trở nên không nhận thức được các thành phần khác. Họ vẫn có thể giao tiếp với nhau, tuy nhiên thông qua một đối tượng trung gian. Để tái sử dụng một thành phần trong một ứng dụng khác, bạn cần cung cấp cho nó một lớp trung gian mới.

Sử dụng Mediator khi bạn thấy mình đang tạo ra hàng loạt các lớp con thành phần chỉ để tái sử dụng một số hành vi cơ bản trong các ngữ cảnh khác nhau.

Vì tất cả các mối quan hệ giữa các thành phần được chứa trong mediator, nên rất dễ định nghĩa các cách hoàn toàn mới cho các thành phần này để hợp tác bằng cách giới thiệu các lớp trung gian mới, mà không cần phải thay đổi các thành phần chính chúng.

4.4.3 Cách hiện thực

1. Xác định một nhóm các lớp liên kết chặt chẽ mà sẽ được hưởng lợi từ việc độc lập hơn (ví dụ, để bảo trì dễ dàng hơn hoặc để việc sử dụng lại các lớp này trở nên đơn giản hơn).

2. Khai báo giao diện mediator và mô tả giao thức truyền thông mong muốn giữa mediator và các thành phần khác nhau. Trong hầu hết các trường hợp, một phương thức duy nhất để nhận thông báo từ các thành phần là đủ.

Giao diện này rất quan trọng khi bạn muốn sử dụng lại các lớp thành phần trong các ngữ cảnh khác nhau. Miễn là thành phần làm việc với mediator của nó thông qua giao diện chung, bạn có thể liên kết thành phần với một cài đặt khác của mediator.

3. Thực hiện lớp mediator cụ thể. Xem xét việc lưu trữ các tham chiếu đến tất cả các thành phần bên trong mediator. Điều này cho phép bạn gọi bất kỳ thành phần nào từ các phương thức của mediator.

4. Bạn có thể đi xa hơn và làm cho mediator chịu trách nhiệm cho việc tạo và phá hủy các đối tượng thành phần. Sau đó, mediator có thể giống như một nhà máy hoặc một facade.

5. Các thành phần nên lưu trữ một tham chiếu đến đối tượng mediator. Kết nối thường được thiết lập trong hàm tạo của thành phần, nơi một đối tượng mediator được truyền như một đối số.

6. Thay đổi mã của các thành phần sao cho chúng gọi phương thức thông báo của mediator thay vì các phương thức trên các thành phần khác. Trích xuất mã liên quan đến việc gọi các thành phần khác vào lớp mediator. Thực thi mã này mỗi khi mediator nhận thông báo từ thành phần đó.

4.4.4 Ưu và nhược điểm

Ưu điểm:

- Nguyên tắc Single Responsibility. Bạn có thể trích xuất các giao tiếp giữa các thành phần khác nhau vào một nơi duy nhất, làm cho việc hiểu và bảo trì dễ dàng hơn.
- Nguyên tắc Mở/Closed. Bạn có thể giới thiệu các mediator mới mà không cần phải thay đổi các thành phần thực tế.
- Bạn có thể giảm sự liên kết giữa các thành phần khác nhau của một chương trình.
- Bạn có thể tái sử dụng các thành phần riêng lẻ một cách dễ dàng hơn.

Nhược điểm: Theo thời gian, một mediator có thể phát triển thành một Đối tượng Thần.

4.4.5 Mối liên hệ với các pattern khác

Chain of Responsibility, Command, Mediator và Observer giải quyết các cách kết nối giữa người gửi và người nhận yêu cầu theo các cách khác nhau:

Chain of Responsibility chuyển tiếp một yêu cầu theo tuần tự dọc theo một chuỗi động của người nhận tiềm năng cho đến khi một trong số họ xử lý nó. Command thiết lập kết nối một chiều giữa người gửi và người nhận. Mediator loại bỏ các kết nối trực tiếp giữa người gửi và người nhận, buộc họ phải giao tiếp gián tiếp thông qua một đối tượng trung gian. Observer cho phép người nhận đăng ký và hủy đăng ký động để nhận yêu cầu. Facade và Mediator có công việc tương tự: họ cố gắng tổ chức sự hợp tác giữa nhiều lớp liên kết chặt chẽ.

Facade định nghĩa một giao diện đơn giản cho một hệ thống con của các đối tượng, nhưng nó không giới thiệu bất kỳ chức năng mới nào. Hệ thống con không nhận thức được về facade. Các đối tượng trong hệ thống con có thể giao tiếp trực tiếp. Mediator tập trung giao tiếp giữa các thành phần của hệ thống. Các thành phần chỉ biết về đối tượng trung gian và không giao tiếp trực tiếp. Sự khác biệt giữa Mediator và Observer thường khó nhận biết. Trong hầu hết các trường hợp, bạn có thể triển khai bất kỳ mẫu nào trong số những mẫu này; nhưng đôi khi bạn có thể áp dụng cả hai đồng thời. Hãy xem cách chúng ta có thể làm điều đó.

Mục tiêu chính của Mediator là loại bỏ sự phụ thuộc lẫn nhau giữa một tập hợp các thành phần của hệ thống. Thay vào đó, những thành phần này trở nên phụ thuộc vào một đối tượng trung gian duy nhất. Mục tiêu của Observer là thiết lập các kết nối một chiều động giữa các đối tượng, trong đó một số đối tượng hành động như cấp dưới của các đối tượng khác.

Có một cách triển khai phổ biến của mẫu Mediator dựa trên Observer. Đối tượng trung gian đóng vai trò nhà xuất bản, và các thành phần đóng vai trò những người đăng ký đăng ký và hủy đăng ký từ các sự kiện của trung gian. Khi Mediator được triển khai theo cách này, nó có thể rất giống với Observer.

Khi bạn bối rối, hãy nhớ rằng bạn có thể triển khai mẫu Mediator theo các cách khác nhau. Ví dụ, bạn có thể liên kết vĩnh viễn tất cả các thành phần với cùng một đối tượng trung gian. Triển khai này sẽ không giống như Observer nhưng vẫn là một ví dụ của mẫu Mediator.

Bây giờ hãy tưởng tượng một chương trình trong đó tất cả các thành phần đều trở thành nhà xuất bản, cho phép kết nối động giữa chúng. Sẽ không có một đối tượng trung gian tập trung, chỉ có một tập hợp phân phối của các quan sát viên.

4.5 Memento

4.5.1 Định nghĩa

Memento là một mẫu thiết kế hành vi cho phép bạn lưu và khôi phục trạng thái trước đó của một đối tượng mà không tiết lộ chi tiết về cách thức thực hiện của nó.

4.5.2 Ứng dụng

Sử dụng mẫu Memento khi bạn muốn tạo ra các bản chụp của trạng thái của đối tượng để có thể khôi phục trạng thái trước đó của đối tượng.

Mẫu Memento cho phép bạn tạo ra các bản sao đầy đủ của trạng thái của một đối tượng, bao gồm các trường riêng tư, và lưu trữ chúng một cách riêng biệt khỏi đối tượng. Trong khi hầu hết mọi người nhớ về mẫu này nhờ vào trường hợp sử dụng "hoàn tác", nó cũng không thể thiếu khi xử lý giao dịch (tức là, nếu bạn cần quay lại một hoạt động khi xảy ra lỗi).

Sử dụng mẫu khi việc truy cập trực tiếp vào các trường/getters/setters của đối tượng vi phạm tính bao đóng của nó.

Memento khiến cho chính đối tượng trở thành trách nhiệm cho việc tạo ra một bản chụp của trạng thái của nó. Không có đối tượng nào khác có thể đọc bản chụp, làm cho dữ liệu trạng thái của đối tượng gốc trở nên an toàn và bảo mật.

4.5.3 Cách hiện thực

1. Xác định lớp nào sẽ đảm nhận vai trò của người khởi tạo. Quan trọng là phải biết liệu chương trình sử dụng một đối tượng trung tâm của loại này hay nhiều đối tượng nhỏ hơn.

2. Tạo lớp memento. Lần lượt, khai báo một tập hợp các trường mô phỏng các trường được khai báo bên trong lớp người khởi tạo.

3. Làm cho lớp memento không thay đổi. Một memento nên chấp nhận dữ liệu chỉ một lần, thông qua hàm tạo. Lớp không nên có bất kỳ phương thức thiết lập nào.

4. Nếu ngôn ngữ lập trình của bạn hỗ trợ lớp lồng, hãy lồng lớp memento vào trong người khởi tạo. Nếu không, hãy trích xuất một giao diện trống từ lớp memento và làm cho tất cả các đối tượng khác sử dụng nó để tham chiếu đến memento. Bạn có thể thêm một số hoạt động siêu dữ liệu vào giao diện, nhưng không gì làm lộ ra trạng thái của người khởi tạo.

5. Thêm một phương thức để tạo ra các memento cho lớp người khởi tạo. Người khởi tạo nên truyền trạng thái của mình vào memento thông qua một hoặc nhiều đối số của hàm tạo của memento.

Kiểu trả về của phương thức nên là giao diện bạn đã trích xuất trong bước trước (giả sử bạn đã trích xuất nó). Dưới nền, phương thức tạo ra memento nên hoạt động trực tiếp với lớp memento.

6. Thêm một phương thức để khôi phục lại trạng thái của người khởi tạo vào lớp của nó. Nó nên chấp nhận một đối tượng memento như một đối số. Nếu bạn trích xuất một giao diện trong bước trước, hãy làm cho nó trở thành kiểu của tham số. Trong trường hợp này, bạn cần chuyển đổi kiểu đối tượng đầu vào thành lớp memento, vì người khởi tạo cần truy cập đầy đủ vào đối tượng đó.

7. Người chăm sóc, cho dù nó đại diện cho một đối tượng lệnh, một lịch sử, hoặc một cái gì đó hoàn toàn khác, nên biết khi nào yêu cầu memento mới từ người khởi tạo, cách lưu trữ chúng và khi nào khôi phục lại người khởi tạo với một memento cụ thể.

8. Liên kết giữa người chăm sóc và người khởi tạo có thể được chuyển vào lớp memento. Trong trường hợp này, mỗi memento phải được kết nối với người khởi tạo đã tạo ra nó. Phương thức khôi phục cũng sẽ di chuyển vào lớp memento. Tuy nhiên, điều này chỉ có ý nghĩa khi lớp memento được lồng vào người khởi tạo hoặc lớp người khởi tạo cung cấp đủ phương thức thiết lập để ghi đè trạng thái của nó.

4.5.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể tạo ra các bản chụp của trạng thái của đối tượng mà không vi phạm tính bao đóng của nó.
- Bạn có thể đơn giản hóa mã của người khởi tạo bằng cách để người chăm sóc duy trì lịch sử trạng thái của người khởi tạo.

Nhược điểm:

- Ứng dụng có thể tiêu tốn nhiều RAM nếu các khách hàng tạo mementos quá thường xuyên.

- Người chăm sóc nên theo dõi vòng đời của người khởi tạo để có thể hủy bỏ các mementos lỗi thời.
- Hầu hết các ngôn ngữ lập trình động, như PHP, Python và JavaScript, không thể đảm bảo rằng trạng thái trong memento được giữ nguyên.

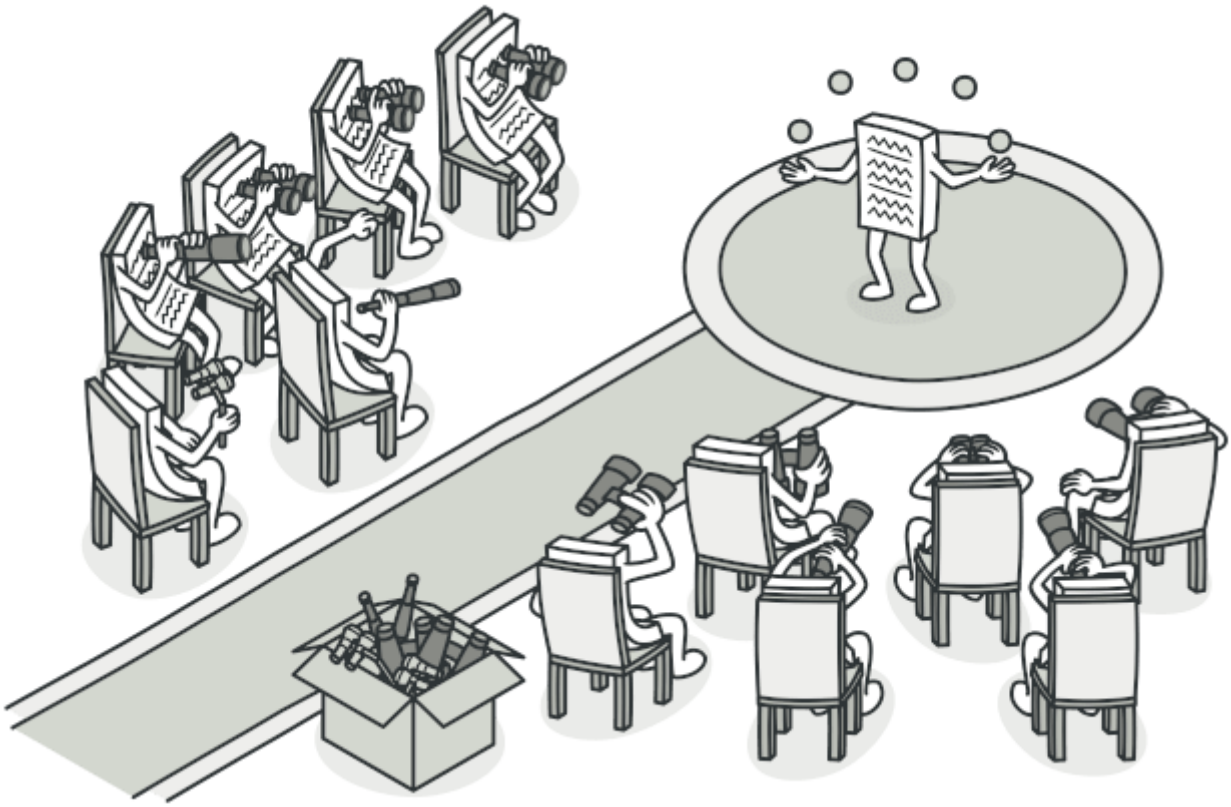
4.5.5 Mối liên hệ với các pattern khác

Bạn có thể sử dụng Command và Memento cùng nhau khi triển khai tính năng "hoàn tác". Trong trường hợp này, các lệnh chịu trách nhiệm thực hiện các hoạt động khác nhau trên một đối tượng mục tiêu, trong khi mementos lưu trạng thái của đối tượng đó ngay trước khi một lệnh được thực thi.

Bạn có thể sử dụng Memento cùng với Iterator để ghi lại trạng thái lặp hiện tại và quay trở lại nó nếu cần thiết.

Đôi khi Prototype có thể là một lựa chọn đơn giản hơn so với Memento. Điều này hoạt động nếu đối tượng, trạng thái của nó bạn muốn lưu trong lịch sử, khá đơn giản và không có liên kết với tài nguyên bên ngoài, hoặc các liên kết dễ tái thiết lập.

4.6 Observer



4.6.1 Định nghĩa

Observer là một mẫu thiết kế hành vi cho phép bạn xác định một cơ chế đăng ký để thông báo cho nhiều đối tượng về bất kỳ sự kiện nào xảy ra với đối tượng mà chúng đang quan sát.

4.6.2 Ứng dụng

Sử dụng mẫu Observer khi các thay đổi vào trạng thái của một đối tượng có thể đòi hỏi thay đổi các đối tượng khác, và tập hợp thực sự các đối tượng không được biết trước hoặc thay đổi một cách động.

Bạn thường gặp vấn đề này khi làm việc với các lớp của giao diện người dùng đồ họa. Ví dụ, bạn tạo ra các lớp nút tùy chỉnh, và bạn muốn cho phép các khách hàng kết nối một số mã tùy chỉnh của họ vào các nút của bạn để nó kích hoạt mỗi khi một người dùng nhấn vào một nút.

Mẫu Observer cho phép bất kỳ đối tượng nào thực hiện giao diện người đăng ký đăng ký để nhận thông báo sự kiện trong các đối tượng phát hành. Bạn có thể thêm cơ chế đăng ký vào các nút của bạn, cho phép các khách hàng kết nối mã tùy chỉnh của họ thông qua các lớp người đăng ký tùy chỉnh.

Sử dụng mẫu khi một số đối tượng trong ứng dụng của bạn phải quan sát các đối tượng khác, nhưng chỉ trong một khoảng thời gian hạn chế hoặc trong các trường hợp cụ thể.

Danh sách đăng ký là động, vì vậy các người đăng ký có thể tham gia hoặc rời khỏi danh sách bất cứ khi nào họ cần.

4.6.3 Cách hiện thực

1. Nhìn lại logic kinh doanh của bạn và cố gắng chia nó thành hai phần: chức năng cốt lõi, độc lập với mã khác, sẽ hoạt động như người phát hành; phần còn lại sẽ trở thành một tập hợp các lớp người đăng ký.

2. Khai báo giao diện người đăng ký. Tối thiểu, nó nên khai báo một phương thức cập nhật duy nhất.

3. Khai báo giao diện người phát hành và mô tả một cặp phương thức để thêm một đối tượng người đăng ký vào và loại bỏ nó khỏi danh sách. Hãy nhớ rằng những người phát hành phải làm việc với người đăng ký chỉ qua giao diện người đăng ký.

4. Quyết định nơi đặt danh sách đăng ký thực tế và cài đặt các phương thức đăng ký. Thông thường, mã này trông giống nhau cho tất cả các loại người phát hành, vì vậy nơi dễ nhìn nhất để đặt nó là trong một lớp trừu tượng được phát sinh trực tiếp từ giao diện người phát hành. Người phát hành cụ thể mở rộng lớp đó, thừa kế hành vi đăng ký.

Tuy nhiên, nếu bạn áp dụng mẫu vào một phân cấp lớp hiện tại, hãy xem xét một phương

pháp dựa trên hợp thành: đặt logic đăng ký vào một đối tượng riêng biệt và khiến tất cả các người phát hành thực sự sử dụng nó.

5. Tạo các lớp người phát hành cụ thể. Mỗi khi có điều quan trọng xảy ra bên trong một người phát hành, nó phải thông báo cho tất cả các người đăng ký của mình.

6. Thực hiện các phương thức thông báo cập nhật trong các lớp người đăng ký cụ thể. Hầu hết người đăng ký sẽ cần một số dữ liệu ngữ cảnh về sự kiện. Nó có thể được truyền như một đối số của phương thức thông báo.

Nhưng còn một lựa chọn khác. Khi nhận được một thông báo, người đăng ký có thể truy xuất bất kỳ dữ liệu nào trực tiếp từ thông báo. Trong trường hợp này, người phát hành phải chuyển nó qua phương thức cập nhật. Lựa chọn ít linh hoạt hơn là liên kết một người phát hành với người đăng ký một cách vĩnh viễn thông qua hàm tạo.

7. Khách hàng phải tạo tất cả các người đăng ký cần thiết và đăng ký chúng với người phát hành phù hợp.

4.6.4 Ưu và nhược điểm

Ưu điểm:

- Nguyên lý Mở/Đóng. Bạn có thể giới thiệu các lớp người đăng ký mới mà không cần phải thay đổi mã của người phát hành (và ngược lại nếu có một giao diện người phát hành).
- Bạn có thể thiết lập mối quan hệ giữa các đối tượng tại thời điểm chạy.

Nhược điểm: Những người đăng kí được thông báo theo thứ tự ngẫu nhiên

4.6.5 Mối liên hệ với các pattern khác

Chain of Responsibility, Command, Mediator và Observer đề cập đến các cách kết nối khác nhau giữa người gửi và người nhận các yêu cầu:

Chain of Responsibility chuyển tiếp yêu cầu theo thứ tự qua một chuỗi động của các người nhận tiềm năng cho đến khi một trong số họ xử lý nó. Command thiết lập các kết nối một chiều giữa người gửi và người nhận. Mediator loại bỏ các kết nối trực tiếp giữa người gửi và người nhận, buộc họ giao tiếp gián tiếp thông qua một đối tượng trung gian. Observer cho phép người nhận đăng ký và hủy đăng ký động để nhận các yêu cầu. Sự khác biệt giữa Mediator và Observer thường không rõ ràng. Trong hầu hết các trường hợp, bạn có thể triển khai bất kỳ mẫu nào trong số các mẫu này; nhưng đôi khi bạn có thể áp dụng cả hai đồng thời. Hãy xem cách chúng ta có thể làm điều đó.

Mục tiêu chính của Mediator là loại bỏ sự phụ thuộc lẫn nhau giữa một tập hợp các thành phần hệ thống. Thay vào đó, những thành phần này trở nên phụ thuộc vào một đối tượng

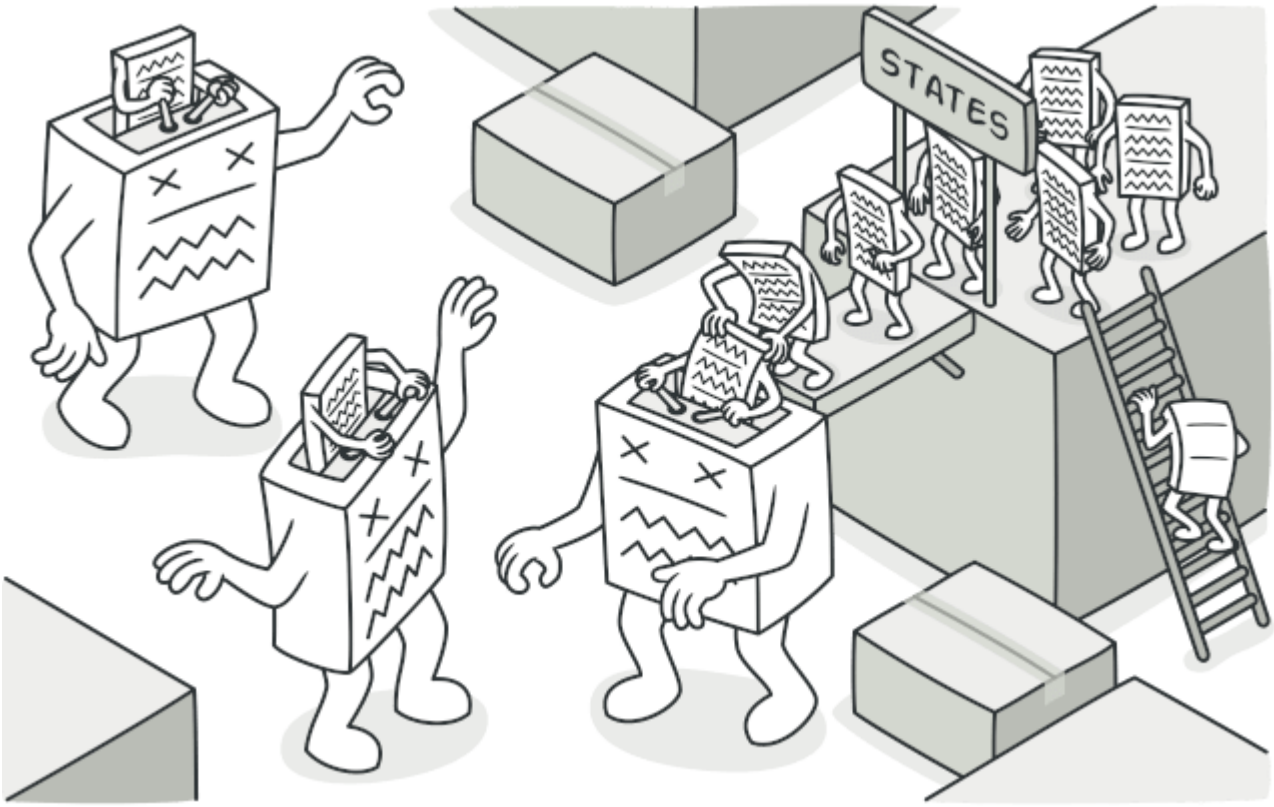
trung gian duy nhất. Mục tiêu của Observer là thiết lập các kết nối một chiều động giữa các đối tượng, trong đó một số đối tượng hành động như các cấp dưới của các đối tượng khác.

Có một cách triển khai phổ biến của mẫu Mediator dựa trên Observer. Đối tượng trung gian đóng vai trò của nhà xuất bản, và các thành phần hành động như các người đăng ký đăng ký và hủy đăng ký từ sự kiện của trung gian. Khi Mediator được triển khai theo cách này, nó có thể trông rất giống với Observer.

Khi bạn bối rối, hãy nhớ rằng bạn có thể triển khai mẫu Mediator theo các cách khác nhau. Ví dụ, bạn có thể liên kết tất cả các thành phần với cùng một đối tượng trung gian. Triển khai này sẽ không giống với Observer nhưng vẫn là một trường hợp của mẫu Mediator.

Bây giờ hãy tưởng tượng một chương trình trong đó tất cả các thành phần đã trở thành nhà xuất bản, cho phép kết nối động giữa chúng. Sẽ không có một đối tượng trung gian tập trung, chỉ có một tập hợp phân phối của các người quan sát.

4.7 State



4.7.1 Định nghĩa

State là một mẫu thiết kế hành vi cho phép một đối tượng thay đổi hành vi của nó khi trạng thái nội bộ của nó thay đổi. Nó trông như là nếu đối tượng đã thay đổi lớp của nó.

4.7.2 Ứng dụng

Sử dụng mẫu State khi bạn có một đối tượng hoạt động khác nhau tùy thuộc vào trạng thái hiện tại của nó, số lượng trạng thái rất lớn, và mã cụ thể cho từng trạng thái thay đổi thường xuyên.

Mẫu này đề xuất rằng bạn trích xuất toàn bộ mã cụ thể cho từng trạng thái vào một tập hợp các lớp riêng biệt. Kết quả là, bạn có thể thêm các trạng thái mới hoặc thay đổi các trạng thái hiện có mà không phụ thuộc vào nhau, giảm chi phí bảo trì.

Sử dụng mẫu khi bạn có một lớp bị ô nhiễm bởi các điều kiện phức tạp thay đổi cách lớp hoạt động tùy thuộc vào các giá trị hiện tại của các trường của lớp.

Mẫu State cho phép bạn trích xuất các nhánh của các điều kiện này thành các phương thức của các lớp trạng thái tương ứng. Khi làm như vậy, bạn cũng có thể làm sạch các trường tạm thời và phương thức trợ giúp liên quan đến mã cụ thể cho trạng thái ra khỏi lớp chính của bạn.

Sử dụng State khi bạn có rất nhiều mã trùng lặp trên các trạng thái và chuyển đổi tương tự của một máy trạng thái dựa vào điều kiện.

Mẫu State cho phép bạn sáng tạo các cấu trúc phân cấp của các lớp trạng thái và giảm mã trùng lặp bằng cách trích xuất mã chung vào các lớp cơ sở trừu tượng.

4.7.3 Cách hiện thực

1. Quyết định lớp nào sẽ hoạt động như ngữ cảnh. Đó có thể là một lớp hiện có đã có mã phụ thuộc vào trạng thái; hoặc một lớp mới, nếu mã cụ thể cho từng trạng thái được phân tán trên nhiều lớp.

2. Khai báo giao diện trạng thái. Mặc dù nó có thể phản ánh tất cả các phương thức được khai báo trong ngữ cảnh, nhưng chỉ nhằm mục đích cho những phương thức có thể chứa hành vi cụ thể cho từng trạng thái.

3. Đối với mỗi trạng thái thực tế, tạo một lớp kế thừa từ giao diện trạng thái. Sau đó, đi qua các phương thức của ngữ cảnh và trích xuất toàn bộ mã liên quan đến trạng thái đó vào lớp mới tạo của bạn.

Trong quá trình chuyển mã vào lớp trạng thái, bạn có thể phát hiện rằng nó phụ thuộc vào các thành viên riêng tư của ngữ cảnh. Có một số biện pháp phòng ngừa:

- Làm cho các trường hoặc phương thức này public.
- Biến hành vi bạn đang trích xuất thành một phương thức public trong ngữ cảnh và gọi nó từ lớp trạng thái. Cách này không đẹp nhưng nhanh chóng, và bạn luôn có thể sửa nó sau này.
- Lồng các lớp trạng thái vào lớp ngữ cảnh, nhưng chỉ nếu ngôn ngữ lập trình của bạn hỗ trợ lồng các lớp.

4. Trong lớp ngữ cảnh, thêm một trường tham chiếu kiểu giao diện trạng thái và một setter public cho phép ghi đè giá trị của trường đó.

5. Đi qua phương thức của ngữ cảnh một lần nữa và thay thế các điều kiện trạng thái trống với cuộc gọi đến các phương thức tương ứng của đối tượng trạng thái.

6. Để chuyển đổi trạng thái của ngữ cảnh, tạo một phiên bản của một trong các lớp trạng thái và chuyển nó vào ngữ cảnh. Bạn có thể làm điều này bên trong ngữ cảnh chính, hoặc ở các trạng thái khác nhau, hoặc ở phía khách hàng. Bất cứ nơi nào điều này được thực hiện, lớp sẽ phụ thuộc vào lớp trạng thái cụ thể mà nó khởi tạo.

4.7.4 Ưu và nhược điểm

Ưu điểm:

- Nguyên lý Trách nhiệm Đơn lẻ. Tổ chức mã liên quan đến các trạng thái cụ thể vào các lớp riêng biệt.
- Nguyên lý Mở/Đóng. Giới thiệu các trạng thái mới mà không thay đổi các lớp trạng thái hiện có hoặc ngữ cảnh.
- Đơn giản hóa mã của ngữ cảnh bằng cách loại bỏ các điều kiện máy trạng thái nặng nề.

Nhược điểm: Áp dụng mẫu này có thể quá mức nếu máy trạng thái chỉ có vài trạng thái hoặc thay đổi ít khi.

4.7.5 Mối liên hệ với các pattern khác

Cầu, Trạng thái, Chiến lược (và một mức độ nào đó Cáp kê) có cấu trúc rất tương tự nhau. Thực sự, tất cả các mẫu này đều dựa trên sự hợp thành, tức là giao công việc cho các đối tượng khác. Tuy nhiên, chúng đều giải quyết các vấn đề khác nhau. Một mẫu không chỉ là một công thức để cấu trúc mã của bạn theo một cách cụ thể. Nó cũng có thể truyền thông với các nhà phát triển khác vấn đề mà mẫu giải quyết.

Trạng thái có thể được coi là một phần mở rộng của Chiến lược. Cả hai mẫu đều dựa trên sự hợp thành: chúng thay đổi hành vi của ngữ cảnh bằng cách giao việc cho các đối tượng trợ giúp. Chiến lược làm cho các đối tượng này hoàn toàn độc lập và không nhận biết về nhau. Tuy nhiên, Trạng thái không hạn chế các phụ thuộc giữa các trạng thái cụ thể, cho phép chúng thay đổi trạng thái của ngữ cảnh theo ý muốn.

4.8 Strategy

4.8.1 Định nghĩa

Chiến lược là một mẫu thiết kế hành vi cho phép bạn định nghĩa một họ thuật toán, đặt mỗi thuật toán vào một lớp riêng biệt, và làm cho các đối tượng của chúng có thể hoán đổi được.

4.8.2 Ứng dụng

Sử dụng mẫu Chiến lược khi bạn muốn sử dụng các biến thể khác nhau của một thuật toán trong một đối tượng và có thể chuyển từ một thuật toán sang một thuật toán khác trong quá trình chạy.

Mẫu Chiến lược cho phép bạn thay đổi hành vi của đối tượng một cách gián tiếp trong quá trình chạy bằng cách liên kết nó với các đối tượng con khác nhau có thể thực hiện các công việc phụ cụ thể theo các cách khác nhau.

Sử dụng Chiến lược khi bạn có rất nhiều lớp tương tự chỉ khác nhau trong cách thực hiện một số hành vi.

Mẫu Chiến lược cho phép bạn trích xuất hành vi biến đổi vào một cấu trúc lớp riêng biệt và kết hợp các lớp ban đầu thành một, từ đó giảm mã trùng lặp.

Sử dụng mẫu này để cô lập logic kinh doanh của một lớp khỏi chi tiết triển khai

của các thuật toán có thể không quan trọng trong ngữ cảnh của logic đó.

Mẫu Chiến lược cho phép bạn cô lập mã, dữ liệu nội bộ và phụ thuộc của các thuật toán khác nhau khỏi phần còn lại của mã. Các khách hàng khác nhau nhận được một giao diện đơn giản để thực thi các thuật toán và chuyển đổi chúng trong quá trình chạy.

Sử dụng mẫu này khi lớp của bạn có một câu lệnh điều kiện lớn mạnh chuyển đổi giữa các biến thể khác nhau của cùng một thuật toán.

Mẫu Chiến lược cho phép bạn loại bỏ điều kiện như vậy bằng cách trích xuất tất cả các thuật toán vào các lớp riêng biệt, tất cả chúng đều thực thi cùng một giao diện. Đối tượng ban đầu ủy quyền thực thi cho một trong các đối tượng này, thay vì triển khai tất cả các biến thể của thuật toán.

4.8.3 Cách hiện thực

1. Trong lớp ngữ cảnh, xác định một thuật toán dễ thay đổi thường xuyên. Điều này cũng có thể là một điều kiện lớn mạnh chọn và thực thi một biến thể của cùng một thuật toán trong quá trình chạy.

2. Khai báo giao diện chiến lược chung cho tất cả các biến thể của thuật toán.

3. Một cách lần lượt, trích xuất tất cả các thuật toán vào các lớp riêng của chúng. Tất cả chúng nên thực thi giao diện chiến lược.

4. Trong lớp ngữ cảnh, thêm một trường để lưu trữ một tham chiếu đến một đối tượng chiến lược. Cung cấp một phương thức thiết lập để thay thế các giá trị của trường đó. Ngữ cảnh chỉ nên làm việc với đối tượng chiến lược thông qua giao diện chiến lược. Ngữ cảnh có thể định nghĩa một giao diện cho phép chiến lược truy cập dữ liệu của nó.

5. Các khách hàng của ngữ cảnh phải liên kết nó với một chiến lược phù hợp phù hợp với cách họ mong đợi ngữ cảnh thực hiện công việc chính của nó.

4.8.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể thay đổi các thuật toán được sử dụng trong một đối tượng trong quá trình chạy.
- Bạn có thể cô lập các chi tiết triển khai của một thuật toán khỏi mã sử dụng nó.
- Bạn có thể thay thế kế thừa bằng sự hợp thành.
- Nguyên lý Mở/Đóng. Bạn có thể giới thiệu các chiến lược mới mà không cần phải thay đổi ngữ cảnh.

Nhược điểm:

- Nếu bạn chỉ có một vài thuật toán và chúng ít khi thay đổi, không có lý do thực sự để làm phức tạp chương trình với các lớp và giao diện mới đi kèm với mẫu thiết kế.
- Các khách hàng phải nhận biết sự khác biệt giữa các chiến lược để có thể chọn một chiến lược phù hợp.
- Nhiều ngôn ngữ lập trình hiện đại có hỗ trợ loại hàm mà cho phép bạn triển khai các phiên bản khác nhau của một thuật toán bên trong một tập hợp các hàm vô danh. Sau đó, bạn có thể sử dụng các hàm này chính xác như bạn đã sử dụng các đối tượng chiến lược, nhưng không làm phình ra mã của bạn với các lớp và giao diện phụ thêm.

4.8.5 Mối liên hệ với các pattern khác

Cầu, Trạng thái, Chiến lược (và một mức độ nào đó Cáp kê) có cấu trúc rất tương tự nhau. Thực sự, tất cả các mẫu này đều dựa trên sự hợp thành, tức là giao công việc cho các đối tượng khác. Tuy nhiên, chúng đều giải quyết các vấn đề khác nhau. Một mẫu không chỉ là một công thức để cấu trúc mã của bạn theo một cách cụ thể. Nó cũng có thể truyền thông với các nhà phát triển khác vấn đề mà mẫu giải quyết.

Lệnh và Chiến lược có vẻ tương tự vì bạn có thể sử dụng cả hai để tham số hóa một đối tượng với một số hành động. Tuy nhiên, chúng có mục đích rất khác nhau.

Bạn có thể sử dụng Lệnh để chuyển đổi bất kỳ thao tác nào thành một đối tượng. Các tham số của thao tác trở thành các trường của đối tượng đó. Việc chuyển đổi cho phép bạn hoãn việc thực thi thao tác, xếp hàng nó, lưu trữ lịch sử các lệnh, gửi các lệnh đến dịch vụ từ xa, v.v.

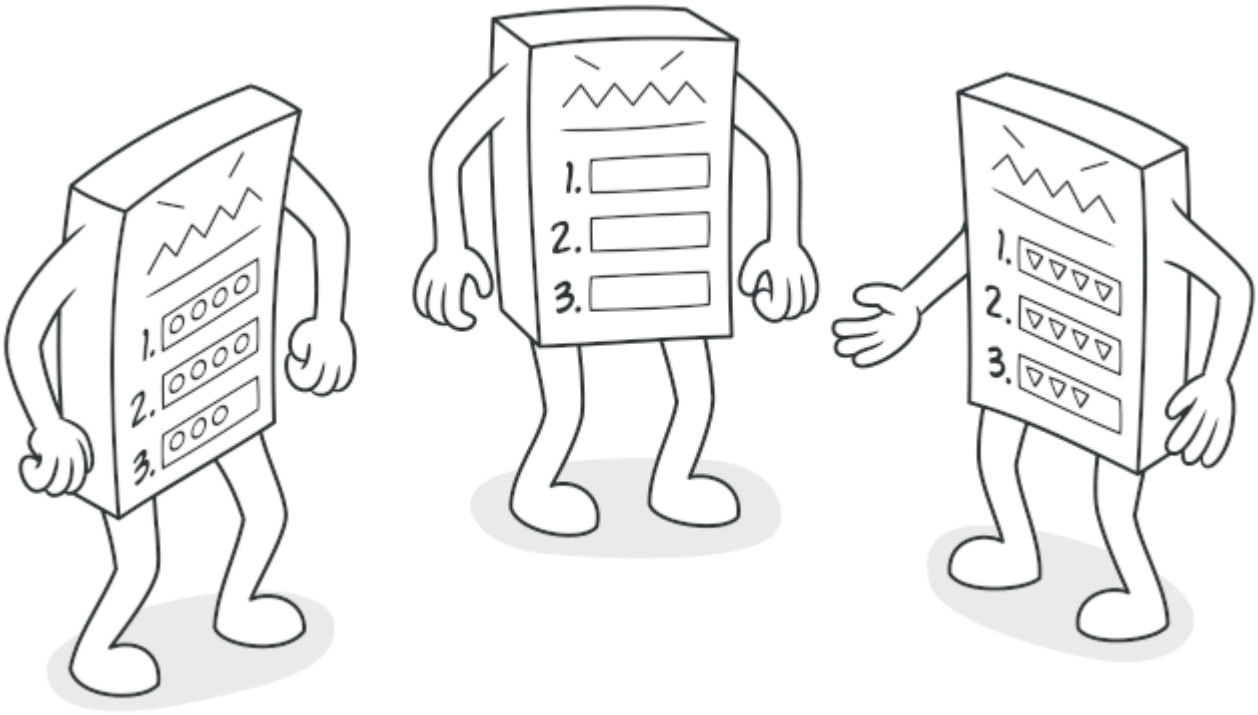
Ngược lại, Chiến lược thường mô tả các cách khác nhau để làm cùng một điều, cho phép bạn thay đổi các thuật toán này trong một lớp ngữ cảnh duy nhất.

Decorator cho phép bạn thay đổi giao diện của một đối tượng, trong khi Chiến lược cho phép bạn thay đổi nội tạng.

Template Method dựa trên kế thừa: nó cho phép bạn thay đổi các phần của một thuật toán bằng cách mở rộng các phần đó trong các lớp con. Chiến lược dựa trên sự hợp thành: bạn có thể thay đổi các phần của hành vi của đối tượng bằng cách cung cấp cho nó các chiến lược khác nhau tương ứng với hành vi đó. Template Method hoạt động ở cấp độ lớp, vì vậy nó là tĩnh. Chiến lược hoạt động ở cấp độ đối tượng, cho phép bạn chuyển đổi hành vi trong quá trình chạy.

Trạng thái có thể được coi là một phần mở rộng của Chiến lược. Cả hai mẫu đều dựa trên sự hợp thành: chúng thay đổi hành vi của ngữ cảnh bằng cách giao việc cho các đối tượng trợ giúp. Chiến lược làm cho các đối tượng này hoàn toàn độc lập và không nhận biết về nhau. Tuy nhiên, Trạng thái không hạn chế các phụ thuộc giữa các trạng thái cụ thể, cho phép chúng thay đổi trạng thái của ngữ cảnh theo ý muốn.

4.9 Template method



4.9.1 Định nghĩa

Template Method là một mẫu thiết kế hành vi mà xác định khung của một thuật toán trong lớp cơ sở nhưng cho phép các lớp con ghi đè các bước cụ thể của thuật toán mà không thay đổi cấu trúc của nó.

4.9.2 Ứng dụng

Sử dụng mẫu Template Method khi bạn muốn cho phép các khách hàng mở rộng chỉ các bước cụ thể của một thuật toán, nhưng không phải là toàn bộ thuật toán hoặc cấu trúc của nó.

Mẫu Template Method cho phép bạn biến một thuật toán lớn thành một loạt các bước cá nhân có thể được mở rộng dễ dàng bởi các lớp con trong khi vẫn giữ nguyên cấu trúc được xác định trong một lớp cơ sở.

Sử dụng mẫu này khi bạn có một số lớp chứa các thuật toán gần như giống nhau với một số khác biệt nhỏ. Kết quả là, bạn có thể cần sửa đổi tất cả các lớp khi thuật toán thay đổi.

Khi bạn biến một thuật toán như vậy thành một phương thức mẫu, bạn cũng có thể kéo các bước có cùng thực thi vào một lớp cơ sở, loại bỏ việc lặp lại mã. Mã khác biệt giữa các lớp con có thể được giữ lại trong các lớp con.

4.9.3 Cách hiện thực

1. Phân tích thuật toán mục tiêu để xem liệu bạn có thể chia nó thành các bước hay không. Xem xét xem những bước nào là chung cho tất cả các lớp con và những bước nào sẽ luôn là duy nhất.

2. Tạo lớp cơ sở trừu tượng và khai báo phương thức mẫu và một tập hợp các phương thức trừu tượng đại diện cho các bước của thuật toán. Phác thảo cấu trúc của thuật toán trong phương thức mẫu bằng cách thực thi các bước tương ứng. Xem xét việc làm cho phương thức mẫu trở nên cuối cùng để ngăn các lớp con ghi đè lên nó.

3. Không sao nếu tất cả các bước kết thúc đều là trừu tượng. Tuy nhiên, một số bước có thể được hưởng lợi từ việc có một triển khai mặc định. Các lớp con không cần phải triển khai những phương thức đó.

4. Hãy suy nghĩ về việc thêm hooks giữa các bước quan trọng của thuật toán.

5. Đối với mỗi biến thể của thuật toán, tạo một lớp con cụ thể mới. Nó phải triển khai tất cả các bước trừu tượng, nhưng cũng có thể ghi đè lên một số bước tùy chọn.

4.9.4 Ưu và nhược điểm

Ưu điểm:

- Bạn có thể cho phép khách hàng ghi đè chỉ một số phần của một thuật toán lớn, làm cho họ ít bị ảnh hưởng bởi các thay đổi xảy ra đối với các phần khác của thuật toán. Bạn có thể kéo mã trùng lặp vào một lớp cơ sở.
- Bạn có thể kéo mã trùng lặp vào một lớp cơ sở.

Nhược điểm:

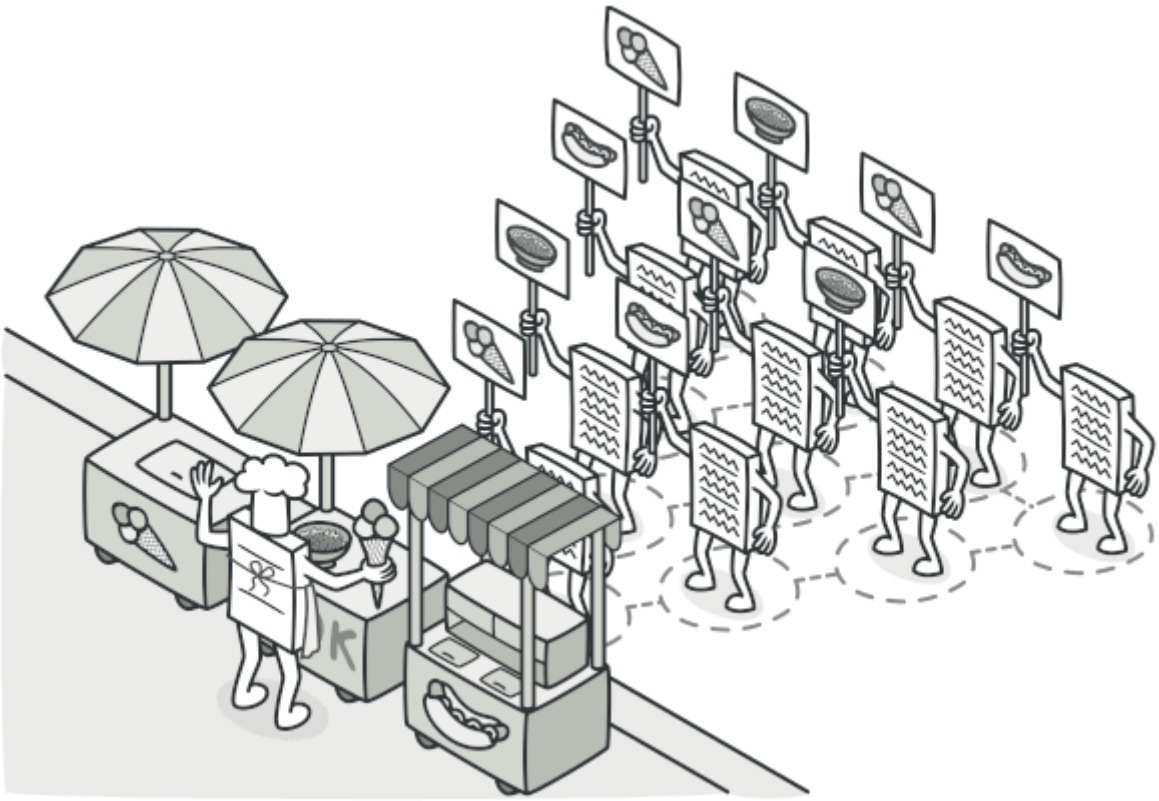
- Một số khách hàng có thể bị hạn chế bởi cấu trúc cơ bản của một thuật toán đã được cung cấp.
- Bạn có thể vi phạm Nguyên tắc Thay thế Liskov bằng cách ngăn cản một triển khai bước mặc định thông qua một lớp con.
- Phương thức mẫu thường khó bảo trì hơn càng nhiều bước nó có.

4.9.5 Mối liên hệ với các pattern khác

Phương thức Factory là một sự chuyên biệt của Phương thức Mẫu. Đồng thời, một Phương thức Factory có thể phục vụ như một bước trong một Phương thức Mẫu lớn.

Phương thức Mẫu dựa trên kế thừa: nó cho phép bạn thay đổi các phần của một thuật toán bằng cách mở rộng các phần đó trong các lớp con. Chiến lược dựa trên sự hợp thành: bạn có thể thay đổi các phần của hành vi của đối tượng bằng cách cung cấp cho nó các chiến lược khác nhau tương ứng với hành vi đó. Phương thức Mẫu hoạt động ở cấp độ lớp, vì vậy nó là tĩnh. Chiến lược hoạt động ở cấp độ đối tượng, cho phép bạn chuyển đổi hành vi trong quá trình chạy.

4.10 Visitor



4.10.1 Định nghĩa

Visitor là một mẫu thiết kế hành vi cho phép bạn tách các thuật toán ra khỏi các đối tượng mà chúng hoạt động.

4.10.2 Ứng dụng

Sử dụng mẫu Visitor khi bạn cần thực hiện một hoạt động trên tất cả các phần tử của một cấu trúc đối tượng phức tạp (ví dụ, một cây đối tượng).

Mẫu Visitor cho phép bạn thực thi một hoạt động trên một tập hợp các đối tượng có các lớp khác nhau bằng cách có một đối tượng visitor thực hiện một số biến thể của cùng một hoạt động, tương ứng với tất cả các lớp mục tiêu.

Sử dụng Visitor để làm sạch logic kinh doanh của các hành vi phụ.

Mẫu cho phép bạn làm cho các lớp chính của ứng dụng của bạn tập trung hơn vào công việc chính của họ bằng cách trích xuất tất cả các hành vi khác vào một tập hợp các lớp visitor.

Sử dụng mẫu khi một hành vi chỉ hợp lý trong một số lớp của một phân cấp lớp, nhưng không trong các lớp khác.

Bạn có thể trích xuất hành vi này vào một lớp visitor riêng và chỉ triển khai các phương thức ghé thăm mà chấp nhận các đối tượng của các lớp liên quan, để phần còn lại trống.

4.10.3 Cách hiện thực

1. Khai báo giao diện visitor với một tập hợp các phương thức "ghé thăm", một cho mỗi lớp phần tử cụ thể tồn tại trong chương trình.

2. Khai báo giao diện phần tử. Nếu bạn đang làm việc với một phân cấp lớp phần tử hiện có, hãy thêm phương thức trừu tượng "chấp nhận" vào lớp cơ sở của phân cấp. Phương thức này nên chấp nhận một đối tượng visitor như một đối số.

3. Triển khai các phương thức chấp nhận trong tất cả các lớp phần tử cụ thể. Các phương thức này chỉ cần chuyển hướng cuộc gọi đến một phương thức ghé thăm trên đối tượng visitor đang đến phù hợp với lớp của phần tử hiện tại.

4. Các lớp phần tử chỉ nên làm việc với các visitor thông qua giao diện visitor. Tuy nhiên, các visitor phải nhận biết tất cả các lớp phần tử cụ thể, được tham chiếu dưới dạng các loại tham số của các phương thức ghé thăm.

5. Đối với mỗi hành vi không thể triển khai bên trong phân cấp phần tử, tạo một lớp visitor cụ thể mới và triển khai tất cả các phương thức ghé thăm.

Bạn có thể gặp phải tình huống mà visitor sẽ cần truy cập vào một số thành viên riêng tư của lớp phần tử. Trong trường hợp này, bạn có thể làm cho các trường hoặc phương thức này trở nên công cộng, vi phạm tính đóng gói của phần tử, hoặc nhúng lớp visitor vào lớp phần tử. Việc cuối cùng chỉ có thể thực hiện nếu bạn may mắn làm việc với một ngôn ngữ lập trình hỗ trợ lớp lồng.

6. Khách hàng phải tạo các đối tượng visitor và truyền chúng vào các phần tử thông qua các phương thức "chấp nhận".

4.10.4 Ưu và nhược điểm

Ưu điểm:

- Nguyên lý Mở/Đóng. Bạn có thể giới thiệu một hành vi mới có thể làm việc với các đối tượng của các lớp khác nhau mà không cần phải thay đổi các lớp này.

- Nguyên lý Đơn trách nhiệm. Bạn có thể di chuyển nhiều phiên bản của cùng một hành vi vào cùng một lớp.
- Một đối tượng visitor có thể tích lũy một số thông tin hữu ích trong khi làm việc với các đối tượng khác nhau. Điều này có thể hữu ích khi bạn muốn duyệt qua một cấu trúc đối tượng phức tạp nào đó, chẳng hạn như một cây đối tượng, và áp dụng visitor cho mỗi đối tượng của cấu trúc này.

Nhược điểm:

- Bạn cần cập nhật tất cả các visitor mỗi khi một lớp được thêm vào hoặc loại bỏ khỏi phân cấp phần tử.
- Các visitor có thể thiếu quyền truy cập cần thiết vào các trường và phương thức riêng tư của các phần tử mà chúng dự định làm việc.

4.10.5 Mối liên hệ với các pattern khác

Bạn có thể coi Visitor như một phiên bản mạnh mẽ của mẫu Command. Các đối tượng của nó có thể thực thi các hoạt động trên các đối tượng khác nhau của các lớp khác nhau.

Bạn có thể sử dụng Visitor để thực thi một hoạt động trên toàn bộ cây Composite.

Bạn có thể sử dụng Visitor cùng với Iterator để duyệt qua một cấu trúc dữ liệu phức tạp và thực hiện một số hoạt động trên các phần tử của nó, ngay cả khi chúng có các lớp khác nhau.

Tài liệu

[1] Guido van Rossum and the Python development team, "Functional Programming HOWTO", March 05, 2017

[2] `https://refactoring.guru/design-patterns/catalog`