

Kỹ Thuật Thiết Kế Hướng Đối Tượng

Cuốn sách thảo luận về việc viết mã hiệu quả về mặt chi phí và dễ bảo trì dành cho các lập trình viên

- Bắt đầu bằng những điều đơn giản
 - Shameless Green là gì
 - Các vấn đề trong bài toán "99 Bottles of Beer"
 - Thuật ngữ: Method và Sending Messages
 - Viết code theo kiểu suy đoán(Speculatively General)

Bắt đầu bằng những điều đơn giản

Chương đầu tiên này bắt đầu bằng cách bóc tách code từ lớp sương mù của sự phức tạp và định nghĩa ý nghĩa của việc viết mã đơn giản.

Bắt đầu bằng những điều đơn giản

Shameless Green là gì

Trong quá trình phát triển phần mềm theo TDD, **Shameless Green** là giai đoạn mà lập trình viên viết mã chỉ đủ để làm cho các bài kiểm thử vượt qua. Điều này có nghĩa là:

- Mã được viết một cách trực tiếp và đơn giản nhất.
- Không quan tâm đến việc mã có cấu trúc tốt hay không.
- Mục tiêu duy nhất là làm cho bài kiểm thử chuyển từ trạng thái thất bại (**red**) sang thành công (**green**)

Sau khi đạt được Shameless Green, lập trình viên có thể tiến hành **refactor** (tái cấu trúc) mã để cải thiện chất lượng và cấu trúc mà vẫn đảm bảo các bài kiểm thử vẫn vượt qua.

Ví dụ

```
def add(a, b):  
    return 3 # Mã "Shameless Green" chỉ để vượt qua bài kiểm thử cụ thể
```

Nếu bài kiểm thử chỉ kiểm tra `add(1, 2) == 3`, thì mã trên sẽ vượt qua. Tuy nhiên, nó không đúng cho các trường hợp khác. Sau khi bài kiểm thử vượt qua, bạn sẽ refactor hàm để xử lý đúng cho mọi trường hợp:

```
def add(a, b):  
    return a + b # Mã sau khi refactor để xử lý đúng cho mọi trường hợp
```

Lợi ích của Shameless Green

- Giúp tiến nhanh trong việc phát triển phần mềm.
- Tập trung vào việc đảm bảo các chức năng hoạt động đúng trước khi tối ưu hóa mã.
- Giảm thiểu thời gian dành cho việc viết mã phức tạp mà chưa chắc đã cần thiết.

Đỗ Ngọc Tú

Công Ty Phần Mềm VHTSoft

Bắt đầu bằng những điều đơn giản

Các vấn đề trong bài toán

"99 Bottles of Beer"

Xem xét các vấn đề bất cập cho **99 Bottles of Beer on the Wall**

Bài toán này dựa trên một bài hát dân gian tiếng Anh có tên là "**99 Bottles of Beer on the Wall**". Nội dung bài hát (và cũng là nội dung của bài toán) như sau:

- Bắt đầu với 99 chai bia trên tường.
- Mỗi lần, bạn hát một đoạn như sau:

99 bottles of beer on the wall, 99 bottles of beer.
Take one down and pass it around, 98 bottles of beer on the wall.

- Sau đó lặp lại với 98, 97, ... cho đến khi còn 0 chai:

No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.

Giải pháp đầu tiên

```
class BottlesOfBeer:
    def song(self):
        return self.verses(99, 0)

    def verses(self, hi, lo):
        return '\n'.join(self.verse(n) for n in range(hi, lo - 1, -1))

    def verse(self, n):
        return (
            f'{"No more" if n == 0 else n} bottle{"s" if n != 1 else ""}'
            f' of beer on the wall, '
            f'{"no more" if n == 0 else n} bottle{"s" if n != 1 else ""} of beer.\n' +
            (f'Go to the store and buy some more, ' if n == 0 else f'Take {"one" if n != 1 else "it"} down and pass it'
            around, ') +
```

```
f{"99" if n - 1 < 0 else ("no more" if n - 1 == 0 else n - 1)} bottle{"s" if n - 1 != 1 else ""}'  
f' of beer on the wall.\n'  
)
```

Đoạn code trên thực hiện một thủ thuật gọn gàng. Nó có thể cô đọng đến mức không ai có thể hiểu được trong khi vẫn giữ lại rất nhiều sự trùng lặp. Mã này khó hiểu vì nó không nhất quán và trùng lặp, và vì nó chứa các khái niệm ẩn mà nó không đặt tên. Và nhúng rất nhiều logic vào chuỗi câu thơ.

Sự nhất quán(Consistency)

Các toán tử ba ngôi đang gây nhầm lẫn. Phần lớn sử dụng dạng đơn giản, như ở dòng 10:

```
"No more" if n == 0 else n
```

Và một số lại lồng toán tử ba ngôi bên trong một toán tử ba ngôi khác như

```
f{"99" if n - 1 < 0 else ("no more" if n - 1 == 0 else n - 1)} bottle{"s" if n - 1 != 1 else ""}'
```

Việc lồng nhiều toán tử ba ngôi như vậy khiến mã khó đọc hơn đối với con người; kiểu viết này làm **tăng chi phí bảo trì mà không mang lại lợi ích gì**.

Lặp lại (Duplication)

Đoạn code này lặp lại cả dữ liệu và logic. Việc lặp lại chuỗi như `"of beer"` hay `"on the wall"` thì dễ thấy và dễ hiểu, nhưng lặp lại logic thì khó hiểu hơn nhiều — đặc biệt khi nó được nhúng bên trong chuỗi.

Ví dụ, logic để thêm chữ `"s"` vào `"bottle"` xuất hiện tới **3 lần**. Hai chỗ đầu giống hệt nhau:

```
"s" if n != 1 else ""
```

Nhưng đến chỗ thứ ba thì lại khác đi chút:

```
"s" if n - 1 != 1 else ""
```

Sự lặp lại này cho thấy: **có những ý tưởng tiềm ẩn trong code** (như việc chia số nhiều, hay phân biệt giữa `n` và `n - 1`) nhưng lại **chưa được đặt tên rõ ràng**. Code như vậy khiến người đọc phải tự đoán xem từng phần đang thực sự làm gì.

Tên biến và hàm (Names)

Điều dễ nhận ra nhất trong hàm `verse` là: **hầu như không có cái tên nào cả**. Mọi logic đều được nhúng trực tiếp vào trong chuỗi string.

Mỗi đoạn logic đó đều mang một ý nghĩa nào đó — nhưng lại **không được đặt tên**, khiến bạn phải tự hình dung trong đầu từng phần đang làm gì.

Nếu muốn code dễ hiểu hơn, đừng bắt người đọc phải đoán. Thay vì nhét logic vào chuỗi, hãy tách nó ra thành các phương thức có tên rõ ràng. Khi đó, `verse` chỉ cần gọi các hàm đó để lấp đầy nội dung — đơn giản, dễ đọc và dễ bảo trì.

Bắt đầu bằng những điều đơn giản

Thuật ngữ: Method và Sending Messages

Trong lập trình hướng đối tượng (OOP), có hai khái niệm quan trọng: **method** và **sending messages(gửi thông điệp)**.

- **Method (phương thức)**: là một hành vi được định nghĩa bên trong một đối tượng. Ví dụ trong lớp **BottlesOfBeer**, có một method tên là `song`.
- **Sending a message(gửi thông điệp)**: nghĩa là **yêu cầu một đối tượng thực hiện một hành vi cụ thể**. Trong ví dụ trước **BottlesOfBeer**, khi gọi `self.verses(...)` bên trong `song`, tức là `song` **gửi thông điệp "verses"** đến chính đối tượng `self`.

Nói ngắn gọn: **method là thứ bạn định nghĩa**, còn **message là thứ bạn gửi để kích hoạt hành vi**.

Tại sao lại dùng từ sending messages(gửi thông điệp) thay vì "gọi hàm"?

Trong nhiều ngôn ngữ lập trình, người ta hay dùng từ "**function**" thay cho "method", hoặc "**call**" thay vì "send". Nghe thì có vẻ giống nhau, nhưng thật ra hơi khác:

- Khi bạn nói "**gọi một hàm**", nghe như bạn **đang điều khiển** hành vi đó, bạn biết chính xác nó sẽ làm gì.
- Nhưng nếu bạn nói "**gửi một thông điệp**", nghĩa là **bạn chỉ đưa ra yêu cầu**, còn việc xử lý ra sao là chuyện của đối tượng.

Cách suy nghĩ này giúp **giảm sự phụ thuộc** giữa các phần trong chương trình. Người gửi thông điệp **không cần biết chi tiết bên trong người nhận xử lý ra sao**, và điều đó giúp cho code dễ bảo trì, dễ mở rộng hơn.

Trong cuốn sách này (và trong nhiều tài liệu về OOP), người ta thường dùng khái niệm "**gửi thông điệp**" thay vì "**gọi hàm**", vì nó thể hiện đúng tinh thần của lập trình hướng đối tượng: **giao tiếp giữa các đối tượng một cách linh hoạt và độc lập**.

Ví dụ: Gọi một hàm - kiểm soát hành vi

```
def make_coffee():  
    print("Boil water")  
    print("Grind coffee beans")  
    print("Brew coffee")  
    print("Pour into cup")  
    print("Done!")  
  
make_coffee()
```

Ở đây, bạn **gọi hàm** `make_coffee()` và bạn **biết chắc từng bước bên trong nó là gì**: đun nước, xay cà phê, pha, rót ra ly...

Bạn kiểm soát hoàn toàn logic bên trong. Không có sự “ẩn ý”, bạn **phụ thuộc vào chi tiết cụ thể** của hàm đó.

Ví dụ: Gửi thông điệp - yêu cầu hành vi, không quan tâm chi tiết

```
class CoffeeMachine:  
    def make_coffee(self):  
        self._heat_water()  
        self._grind_beans()  
        self._brew()  
        self._pour()  
  
    # Các method private bên trong  
    def _heat_water(self): ...  
    def _grind_beans(self): ...  
    def _brew(self): ...  
    def _pour(self): ...  
  
machine = CoffeeMachine()  
machine.make_coffee()
```

Lúc này, bạn **gửi thông điệp** `make_coffee` **đến** `machine`, yêu cầu nó pha cà phê. Nhưng **bạn không cần biết** nó sẽ đun nước hay pha như thế nào, chỉ cần biết nó **sẽ pha được cà phê** là đủ.

Ví dụ 2 "Gọi hàm" vs "Gửi thông điệp" trong bối cảnh phần mềm doanh nghiệp (ERP) - cụ thể là quy trình tạo đơn đặt hàng (Purchase Order).

Gọi hàm - kiểu procedural, kiểm soát mọi bước

Bạn viết một hàm và kiểm soát chi tiết quy trình tạo đơn hàng.

```
def create_purchase_order(supplier, items):  
    # Kiểm tra tồn kho  
    for item in items:  
        if not check_inventory(item):  
            raise Exception(f"{item} is not available in inventory")  
  
    # Tính tổng tiền  
    total = sum(item.price * item.quantity for item in items)  
  
    # Tạo đơn hàng  
    po = {  
        "supplier": supplier,  
        "items": items,  
        "total": total,  
    }  
  
    # Gửi email xác nhận  
    send_email(supplier.email, f"PO Created: Total = {total}")  
    return po
```

Bạn kiểm soát hết mọi bước. Nếu muốn thay đổi logic, ví dụ thay đổi cách gửi email hay kiểm kho, bạn phải sửa trực tiếp trong hàm → **khó bảo trì, không theo tư duy hướng đối tượng**.

Gửi thông điệp - kiểu OOP, giao trách nhiệm cho đối tượng

```
class PurchaseOrder:  
    def __init__(self, supplier, items):  
        self.supplier = supplier  
        self.items = items  
        self.total = self.calculate_total()  
  
    def calculate_total(self):  
        return sum(item.price * item.quantity for item in self.items)  
  
    def validate(self):  
        for item in self.items:  
            item.validate_inventory()
```

```
def confirm(self):
    self.validate()
    self.send_confirmation_email()

def send_confirmation_email(self):
    EmailService.send(to=self.supplier.email, subject="PO Confirmed", body=str(self))

# Cách sử dụng
po = PurchaseOrder(supplier, items)
po.confirm() # <== GỬI THÔNG ĐIỆP "confirm"
```

Lợi ích:

- Gửi thông điệp `confirm()` mà không cần biết chi tiết nội bộ.
- Có thể thay `EmailService` bằng service khác mà không ảnh hưởng nơi gọi.
- Tách logic thành nhiều lớp – mỗi lớp có trách nhiệm rõ ràng.

Tóm lại:

- **Gọi hàm:** Bạn biết (và phải biết) nó làm gì – **bạn điều khiển chi tiết.**
- **Gửi thông điệp:** Bạn chỉ cần kết quả – **bạn tin rằng đối tượng sẽ làm đúng nhiệm vụ**, còn chi tiết xử lý thì để nó lo.

Cách “gửi thông điệp” chính là tư duy quan trọng của lập trình hướng đối tượng – tạo nên **sự độc lập, dễ thay đổi, và dễ bảo trì** trong phần mềm.

Đỗ Ngọc Tú
Công Ty Phần Mềm VHTSoft

Bắt đầu bằng những điều đơn giản

Viết code theo kiểu suy đoán(Speculatively General)

Viết code theo kiểu suy đoán trong lập trình gọi là một **code mùi(code smell)** — tức là một **dấu hiệu** cho thấy có thể bạn đang **thiết kế hệ thống quá phức tạp so với nhu cầu hiện tại**, vì bạn **dự đoán tương lai** và **tạo ra những tính năng chung chung chưa cần thiết**. hiểu đơn giản là khi bạn viết mã không phục vụ yêu cầu hiện tại, mà để “phòng hờ” cho những gì *có thể* xảy ra trong tương lai."

Hãy trở lại bài toán **99 Bottles of Beer**

Giải pháp tiếp theo này đi theo một hướng khác. Nó thực hiện tốt nhiều việc nhưng không thể cưỡng lại việc đắm chìm vào sự phức tạp không cần thiết. Hãy xem mã bên dưới:

```
class BottlesOfBeer:
    def song(self):
        return self.verses(99, 0)

    def verses(self, high, low):
        return '\n'.join(
            self.verse(verse_number) for verse_number in sorted(range(low, high+1), reverse=True)
        )

    def verse(self, number):
        return self.verse_for(number).text()

    def verse_for(self, number):
        def no_more(verse):
            return (
                'No more bottles of beer on the wall, '
                'no more bottles of beer.\n'
                'Go to the store and buy some more, '
                '99 bottles of beer on the wall.\n'
            )
```

```

def last_one(verse):
    return (
        '1 bottle of beer on the wall, '
        '1 bottle of beer.\n'
        'Take it down and pass it around, '
        'no more bottles of beer on the wall.\n'
    )

def penultimate(verse):
    return (
        '2 bottles of beer on the wall, '
        '2 bottles of beer.\n'
        'Take one down and pass it around, '
        '1 bottle of beer on the wall.\n'
    )

def default(verse):
    return (
        f'{verse.number} bottles of beer on the wall, '
        f'{verse.number} bottles of beer.\n'
        f'Take one down and pass it around, '
        f'{verse.number - 1} bottles of beer on the wall.\n'
    )

lyric_functions = [no_more, last_one, penultimate, default]
chosen_lyric = (
    lyric_functions[0] if number == 0 else
    lyric_functions[1] if number == 1 else
    lyric_functions[2] if number == 2 else
    lyric_functions[3]
)
return Verse(number, chosen_lyric)

class Verse:
    def __init__(self, number, lyrics):
        self.number = number
        self.lyrics = lyrics
    def text(self):

```

```
return self.lyrics(self)
```

Trong đoạn code, mục đích ban đầu **rất đơn giản**: in ra bài hát “99 Bottles of Beer” với một số đoạn đặc biệt khi còn lại 2, 1, và 0 chai.

Tuy nhiên, tác giả đã **tổng quát hóa quá mức cần thiết**, ví dụ như:

1. Việc sử dụng lớp Verse

```
class Verse:
    def __init__(self, number, lyrics):
        self.number = number
        self.lyrics = lyrics
    def text(self):
        return self.lyrics(self)
```

Vấn đề:

Nhưng ở đây, người viết đã tạo:

- Một lớp `Verse`
- Truyền vào số chai và function tạo lyrics
- Gọi method `text()` để in

→ **Dự đoán rằng trong tương lai sẽ cần tách dữ liệu và logic ra như vậy** (có thể để mở rộng? kế thừa? tái sử dụng?). Nhưng hiện tại thì chưa cần. Việc này **gây phức tạp không cần thiết**.

Chỉ cần viết một hàm như sau là đủ

```
def generate_verse(number):
    if number == 0:
        return "No more bottles of beer..."
    elif number == 1:
        return "1 bottle of beer..."
    ...
```

Mục tiêu thực tế là chỉ cần in ra đoạn nhạc tương ứng với 99, 98, ..., 0 chai bia.

Chưa có dấu hiệu cần tái sử dụng, kế thừa

Bạn tạo class khi:

- Có nhiều thuộc tính liên quan cần gói gọn
- Có nhiều hành vi (method) tương tác với dữ liệu đó
- Hoặc bạn muốn tái sử dụng, kế thừa

Nhưng ở đây `Verse` chỉ có 2 thành phần:

- Một con số
- Một function đơn giản trả về chuỗi

Và `text()` thì chỉ gọi đúng 1 dòng:

```
return self.lyrics(self)
```

Không đáng để tạo class. Nếu bạn không **chắc chắn** sẽ cần mở rộng như vậy, thì đó là **suy đoán**

2. Việc chọn lyrics qua `lyrics_functions` list

```
lyric_functions = [no_more, last_one, penultimate, default]
chosen_lyric = (
    lyric_functions[0] if number == 0 else
    lyric_functions[1] if number == 1 else
    lyric_functions[2] if number == 2 else
    lyric_functions[3]
)
```

Vấn đề:

- Việc lưu các function vào list rồi chọn qua index tạo ra một dạng “tổng quát hóa” **khó hiểu** hơn việc viết thẳng `if...elif...else`.
- Cách này **giống như bạn đang chuẩn bị để xử lý 100 kiểu câu hát khác nhau**, nhưng thực tế chỉ có **4 trường hợp** rõ ràng.

Việc này chỉ thực sự có ích **khi số lượng function cần chọn lớn hoặc có cấu trúc lặp đi lặp lại**, ví dụ:

```
lyric_functions[number] # khi có hàng trăm trường hợp
```

Nhưng trong trường hợp này, chỉ có 4 trường hợp nhỏ và đã biết trước, nên viết như sau **vừa đủ, dễ đọc hơn**:

```
if number == 0:
    chosen_lyric = no_more
elif number == 1:
    chosen_lyric = last_one
elif number == 2:
    chosen_lyric = penultimate
else:
```

```
chosen_lyric = default
```

3. Đặt tên phương thức `verse_for()`

```
def verse_for(self, number):
```

→ Vấn đề:

Cái tên `verse_for()` làm người đọc tưởng rằng: hàm này có thể được mở rộng/phân nhánh thành hàng tá loại `verse` khác nhau trong tương lai. Nhưng hiện tại nó chỉ dùng để chọn 1 trong 4 đoạn văn bản đơn giản.

Tóm lại: Đây là phần “Speculatively General”?

Vị trí	Tổng quát hóa gì?	Có cần thiết không?
Verse class	Đóng gói logic và data	<input type="checkbox"/> Không – quá phức tạp cho tác vụ in chuỗi
List lyric_functions	Tạo mảng function để chọn	<input type="checkbox"/> Không – dùng <code>if-elif</code> đơn giản rõ ràng hơn
Hàm verse_for()	Tên gợi ý khả năng mở rộng lớn	<input type="checkbox"/> Không – hiện tại chỉ có 4 case đơn giản