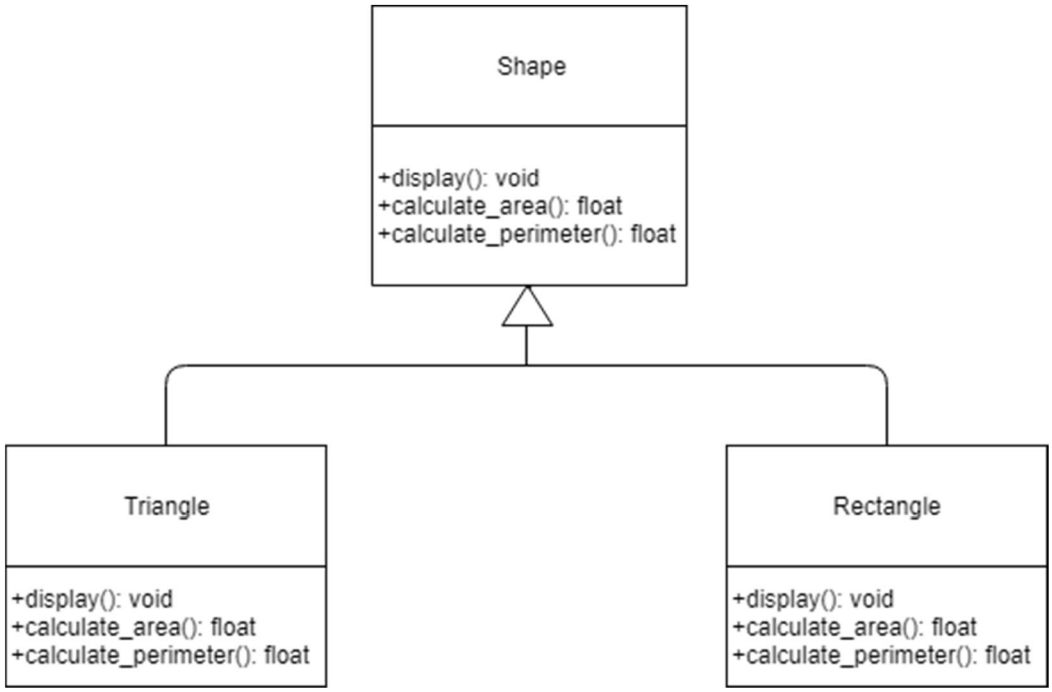
VISITOR PATTERN

I. Ý định

Visitor pattern cho phép định nghĩa các thao tác (operations) mới trên các yếu tố (elements) của một cấu trúc đối tượng (object structure) mà không làm thay đổi các lớp (class) của các yếu tố đó.

II. Ví dụ đơn giản

Xét một class Shape, được kế thừa bởi 2 class Triangle và Rectangle (thực tế sẽ có rất nhiều class có thể kế thừa Shape). Ở đây ta chỉ xét đến các thao tác trên các class này, bao gồm display, calculate_area và calculate_perimeter. Dưới đây là class diagram của cây phân cấp kế thừa class Shape:



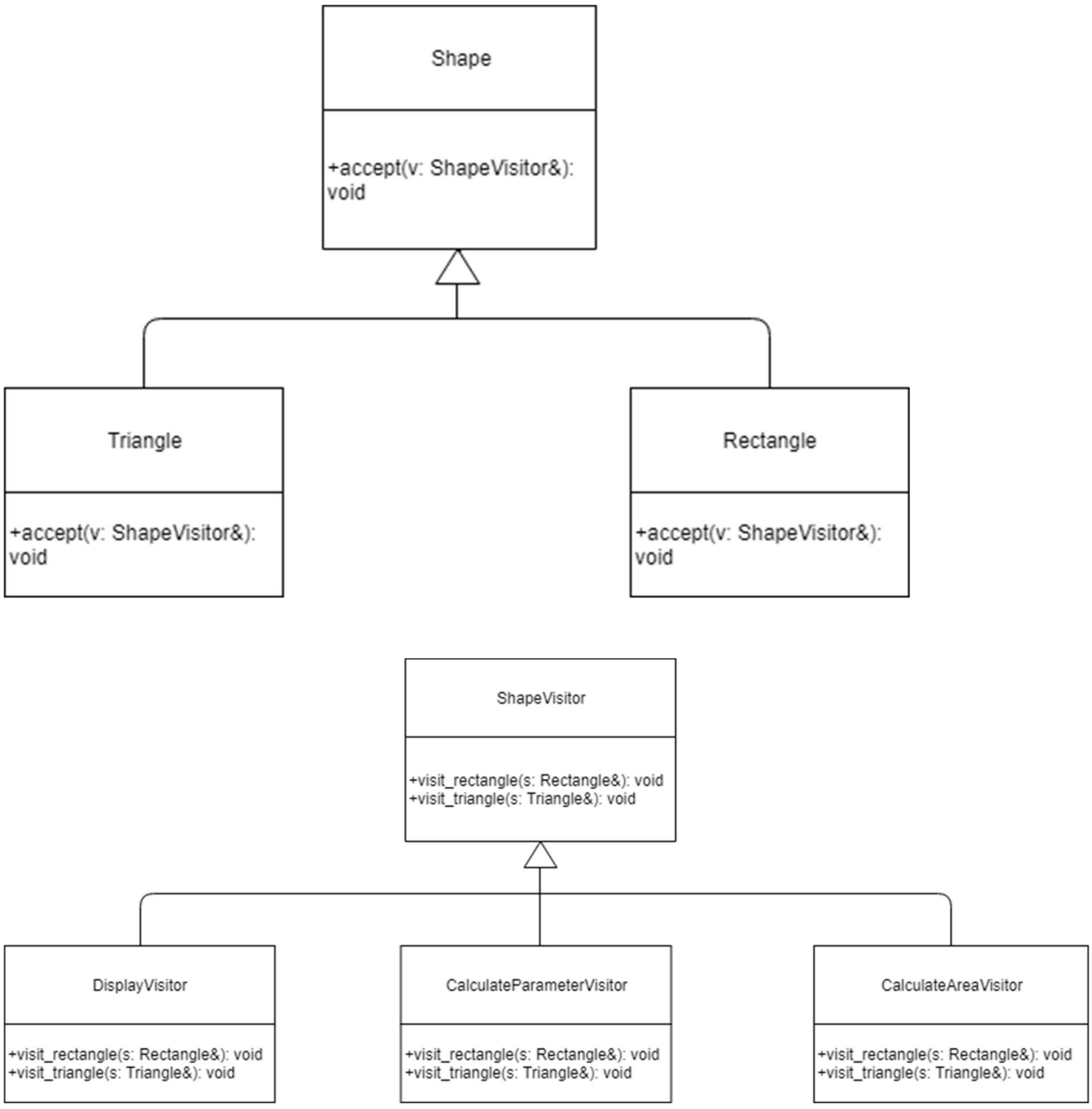
Vấn đề ở đây là phân phối những thao tác này trên các class kế thừa Shape khác nhau sẽ dẫn tới một hệ thống khó hiểu, khó duy trì và thay đổi. Nó sẽ gây nhầm lẫn khi code của các thao tác display(), calculate_area() và calculate_perimeter() trộn lẫn vào nhau. Hơn thế nữa, nếu chúng ta thêm các thao tác mới (thường sẽ cần biên dịch lại hết tất cả các class này). Nếu mỗi thao tác mới có thể được thêm vào

một cách độc lập thì sẽ tốt hơn và như thế các class sẽ không phụ thuộc vào các thao tác thực hiện trên các class đó.

Chúng ta có thể làm được điều này nếu đóng gói các thao tác liên quan với nhau từ mỗi class trong một đối tượng riêng biệt và ta gọi đối tượng này là visitor. Khi một yếu tố "accepts" (chấp nhận) một visitor, nó gửi yêu cầu đến visitor chịu trách nhiệm cho lớp của yếu tố đó. Visitor này sẽ bao gồm yếu tố đó như là một đối số. Lúc này visitor thực hiện thao tác cho yếu tố đó - thao tác này đã từng nằm trong class của yếu tố đó (trước khi có visitor).

Theo ví dụ trên, khi chưa sử dụng visitor pattern, ta sẽ tính chu vi hoặc diện tích bằng cách gọi calculate_perimeter() và calculate_area() của từng đối tượng thuộc class Triangle hoặc Rectangle. Khi sử dụng visitor pattern, nếu muốn tính diện tích của một hình, ta sẽ gọi thao tác accept (của class hình đó) với đối số của thao tác này là một đối tượng thuộc class CalculateAreaVisitor (accept(visitor& v)). Thao tác accept sẽ gọi lại một thao tác tương ứng của visitor: class Triangle sẽ gọi thao tác visit_triangle(), class Rectangle gọi visit_rectangle(). Thao tác calculate_area() của 2 class Triangle và Rectangle lúc này là thao tác visit_triangle() và visit_rectangle() của class CalculateAreaVisitor.

Để visitor pattern có thể hoạt động tốt với nhiều thao tác khác nhau. Ta cần định nghĩa một abstract class (hay base class) cho tất cả các visitor của cây phân cấp kế thừa. Ví dụ ,theo cây phân cấp kế thừa Shape như trên, ta cần một class ShapeVisitor. Trong class ShapeVisitor phải khai báo một thao tác visit cho mỗi class kế thừa của Shape. Nếu ta cần thêm vào một thao tác mới cho cấu trúc đối tượng, ta không cần thêm code vào các class của Shape, ta chỉ cần tạo thêm 1 class kế thừa của ShapeVisitor. Visitor pattern sẽ đóng gói các thao tác liên quan với nhau vào trong cùng một lớp.



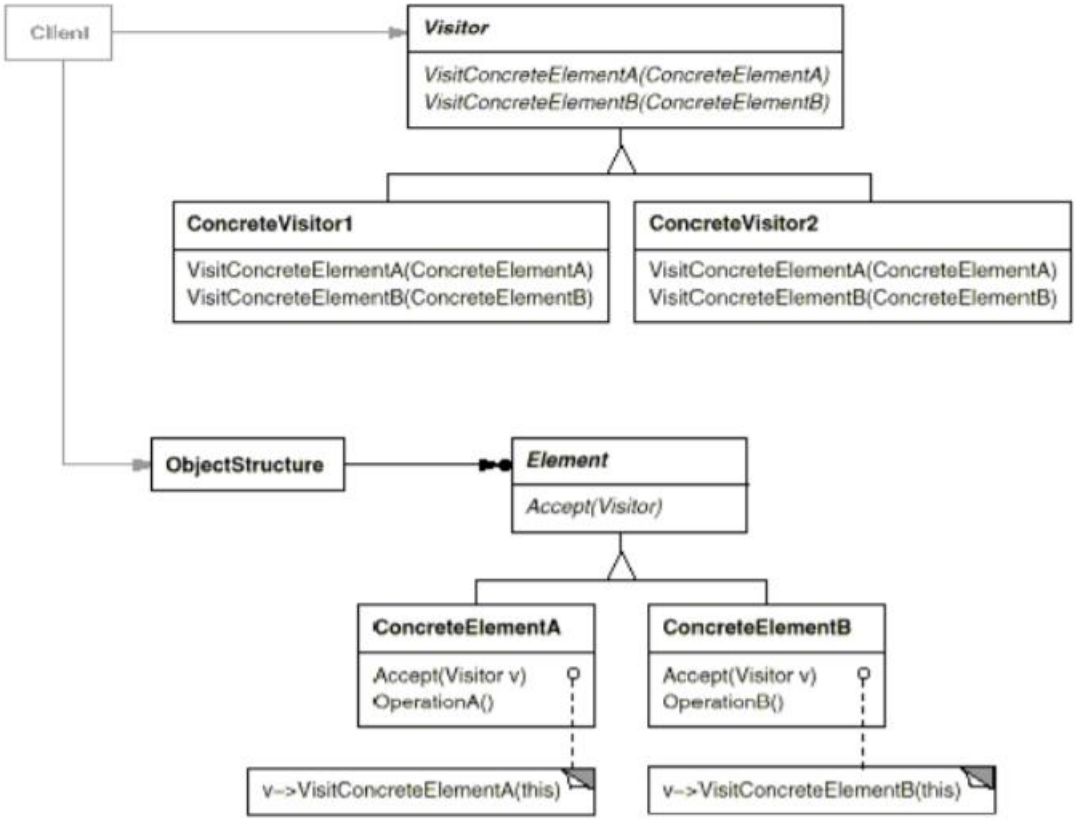
Với visitor pattern, ta định nghĩa hai cây phân cấp kế thừa: một cho các class đang được thao tác (cây phân cấp kế thừa Shape), một cho các visitor định nghĩa các thao tác đó (cây phân cấp kế thừa ShapeVisitor). Lúc này, ta tạo thao tác mới bằng cách tạo lớp kế thừa của class ShapeVisitor.

III. Khả năng ứng dụng

Dùng visitor pattern khi:

- Một cấu trúc đối tượng chứa nhiều class của nhiều đối tượng với giao diện khác nhau, và bạn muốn thực hiện những thao tác trên các đối tượng này mà những thao tác đó dựa trên concrete class cụ thể.
- Nhiều thao tác khác biệt và không liên quan cần được thực hiện trong một cấu trúc đối tượng, và bạn muốn tránh các class của các thao tác đó bị "ô nhiễm" bởi các thao tác này. Visitor giúp bạn giữ những thao tác liên quan trong cùng một class. Khi một cấu trúc đối tượng được sử dụng bởi nhiều ứng dụng khác nhau, ta sử dụng visitor cho việc để các thao tác chỉ ở những ứng dụng cần chúng.
- Những class định nghĩa cấu trúc đối tượng hiếm khi thay đổi, nhưng bạn thường xuyên định nghĩa những thao tác mới trên cấu trúc đối tượng này. Thay đổi các class của cấu trúc đối tượng yêu cầu định nghĩa lại giao diện cho tất cả các visitor, điều này có khả năng rất tốn kém. Nếu các class của cấu trúc đối tượng thay đổi thường xuyên thì sẽ tốt hơn nếu định nghĩa các thao tác trên chính những lớp đó.

IV. Cấu trúc



V. Thành phần tham gia

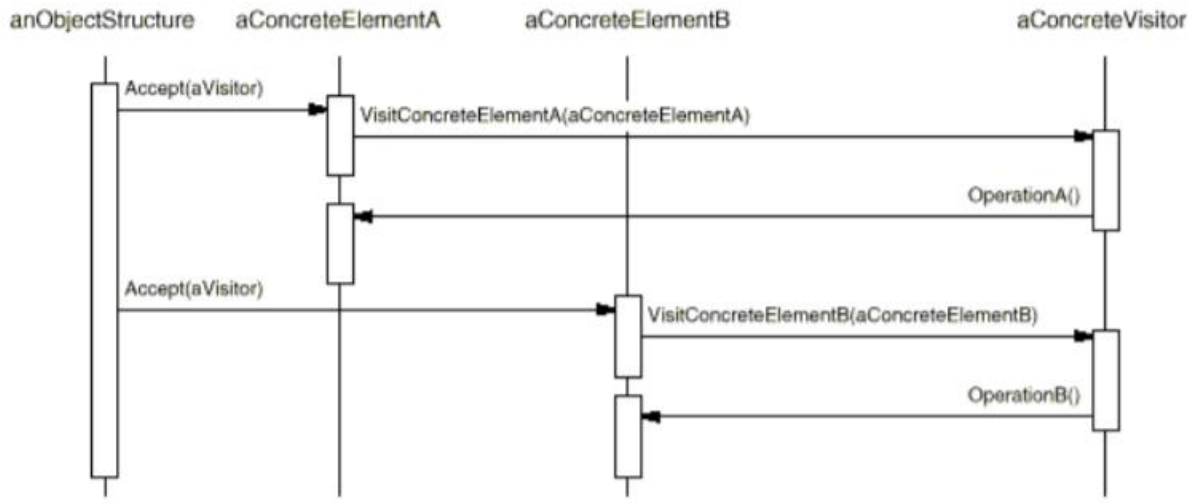
- Visitor (ShapeVisitor):
  - Định nghĩa các thao tác visit cho từng class thuộc ConcreteElement trong cấu trúc đối tượng. Tên và signature của thao tác xác định class sẽ gửi yêu cầu visit đến visitor. Điều này cho visitor quyết định concrete class của phần tử (element) được visit. Lúc đó visitor có thể truy cập trực tiếp element qua giao diện riêng của phần tử đó.
- ConcreteVisitor (DisplayVisitor): (visitor cụ thể)
  - Triển khai các thao tác được khai báo bởi visitor. Mỗi thao tác triển khai một đoạn của thuật toán được định nghĩa cho class của đối tượng tương ứng trong cấu trúc. ConcreteVisitor cung cấp một context cho thuật toán và lưu trạng thái cục bộ (local state) của nó. Trạng thái này thường tích trữ các kết quả trong quá trình hoạt động của cấu trúc.
- Element (Shape):
  - Định nghĩa một thao tác accept có đối số là visitor.
- ConcreteElement (Triangle, Rectangle): (phần tử cụ thể)
  - Triển khai thao tác accept có đối số là visitor
- ObjectStructure (Program)
  - Có thể liệt kê các phần tử của nó
  - Có khả năng cung cấp giao diện cấp cao cho visitor visit các phần tử của nó.
  - Có thể là một composite hoặc một collection như là list hoặc set.

VI. Sự phối hợp

Một client sử dụng visitor pattern phải tạo ra một đối tượng ConcreteVisitor và sau đó đi qua cấu trúc đối tượng, visit mỗi phần tử với visitor.

Khi một phần tử được visit, nó gọi một thao tác của visitor tương ứng với class của nó. phần tử này cung cấp bản thân như là một đối số cho thao tác này để visitor có thể truy cập trạng thái của phần tử này nếu cần.

Sơ đồ tương tác dưới đây thể hiện sự phối hợp giữa cấu trúc đối tượng,



visitor, và 2 phần tử:

VII. Ưu điểm

1. *Visitor giúp thêm thao tác dễ dàng hơn.* Visitor khiến việc thêm những thao tác phụ thuộc vào thành phần của các đối tượng phức tạp trở nên dễ dàng hơn. Ta có thể định nghĩa một thao tác mới một cách đơn giản là thêm một visitor mới. Ngược lại, nếu ta trải rộng các function trên các class thì ta phải thay đổi từng class để định nghĩa một thao tác mới.
2. *Visitor tập trung những thao tác liên quan và tách biệt những thao tác không liên quan với nhau.* Những hành động liên quan với nhau không bị trải rộng trên các class định nghĩa cấu trúc đối tượng mà được tập trung lại trong một visitor. Những tập hợp hành động không liên quan với nhau được phân vùng trong visitor derived class của chính các tập hợp đó. Điều đó đơn giản hóa cả những class định nghĩa các phần tử và các thuật toán được định nghĩa trong các visitor. Bất kì cấu trúc dữ liệu dành riêng cho thuật toán đều có thể được giấu trong visitor.

3. *Visit khắp hệ thống phân cấp lớp*. Một Iterator (xem Iterator (257)) có thể truy cập các đối tượng trong một cấu trúc khi nó đi qua chúng bằng cách gọi các thao tác của chúng. Nhưng Iterator không thể làm việc trên các cấu trúc đối tượng với các loại phần tử khác nhau. Ví dụ: giao diện Iterator được xác định trên trang 263 chỉ có thể truy cập các đối tượng thuộc loại Item:

```
template <class Item>
```

```
class Iterator {
        // ...
        Item CurrentItem () const;
};
```

Điều này ngụ ý rằng tất cả các element mà Iterator có thể visit đều có một parent class chung. Visitor không có hạn chế này. Nó có thể truy cập các đối tượng không có parent class chung. Bạn có thể thêm bất kỳ loại đối tượng nào vào giao diện của visitor. Ví dụ, trong class

```
Visitor {
Public:
        // ... void VisitMyType (MyType *);
        void VisitYourType (YourType *);
};
```

MyType và YourType hoàn toàn không phải liên quan đến tính kế thừa.

4. *Tích trữ trạng trái.* Các visitor có thể tích trữ trạng thái khi nó visit từng phần tử của cấu trúc đối tượng. Nếu không có visitor, trạng thái này sẽ được truyền như là một đối số bổ sung cho những thao tác thực hiện việc truyền tải, hoặc nó có thể xuất hiện dưới dạng các biến toàn cục.

VIII. Khuyết điểm

1. *Việc thêm các class ConcreteElement sẽ khó.* Visitor pattern khiến cho việc thêm các subclass của Element trở nên khó khăn. Mỗi ConcreteElement mới làm tăng thêm số lượng thao tác trừu tượng (abstact operation) trên Visitor và sự triển khai tương ứng trong mỗi class ConcreteVisitor. Đôi khi một sự triển khai mặc định có thể được cung cấp trong visitor có thể được kế thừa bởi đa số ConcreteVisitor, nhưng điều này được xem như là một ngoại lệ hơn là một luật lệ.

2. Vì thế nên sự suy xét thiết yếu trong việc áp dụng visitor pattern là bạn nhiều khả năng sẽ thay đổi thuật toán được áp dụng trên một cấu trúc đối tượng hoặc các class của những đối tượng làm nên cấu trúc. Hệ thống phân cấp class Visitor (hoặc cây kế thừa phân cấp class vistor) có thể trở nên khó khăn để duy trì nếu các class ConcreteElement mới được thêm vào thường xuyên. Trong những trường hợp như vậy, sẽ dễ hơn nếu định nghĩa các thao

tác trên các class tạo nên cấu trúc đó. Nếu hệ thống phân cấp Element class ổn định, nhưng bạn đang liên tục thêm thao tác hoặc thay đổi thuật toán, thì visitor pattern sẽ giúp bạn quản lý những sự thay đổi đó.

3. Phá vỡ tính đóng gói (encapsulation). Cách tiếp cận của visitor giả định rằng giao diện của ConcreteElement đủ quyền lực để các visitor làm việc của chúng. Kết quả là pattern này thường bắt buộc bạn cung cấp các thao tác public mà truy cập trạng thái nội bộ của phần tử (element), điều này có thể làm ảnh hưởng đến tính đóng gói của nó.