VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

OBJECT ORIENTED PROGRAMMING (22_5)

ASSIGNMENT:

GROUP ASSIGNMENT WEEK 07

Advisor(s)::	Hồ Tuấn Thanh	
Student(s)::	Nguyễn Hoàng Anh	22120014
	Trần Hùng Anh	22120016 (Leader)
	Trương Tiến Anh	22120017
	Đoàn Minh Cường	22120043
	Nông Quốc Việt	22120432

Ho Chi Minh City, 12/2023

Mục lục

Phần I
Giới thiệu nhóm và tiến độ công việc

Tên nhóm: **ClownIT**

Thông tin các thành viên nhóm:

STT	Họ Tên	MSSV	Email
1	Nguyễn Hoàng Anh	22120014	hoanganhik172510@gmail.com
2	Trần Hùng Anh	22120016	anhth5659@gmail.com
3	Trương Tiến Anh	22120017	truongtienanh16@gmail.com
4	Đoàn Minh Cường	22120043	dmcuong02072004@gmail.com
5	Nông Quốc Việt	22120432	quocviett1302@gmail.com

Bảng 1: *Thông tin các thành viên trong nhóm*

Công việc được phân chia và tiến độ công việc:

STT	Họ Tên	Công việc được giao	Tiến độ công việc	Vấn đề gặp phải
1	Nguyễn Hoàng Anh	Tìm hiểu và báo cáo composite design pattern	Hoàn thành	Không có
2	Trần Hùng Anh	Lấy ví dụ thực tế và vẽ class diagram	Hoàn thành	Không có
3	Trương Tiến Anh	Cài đặt code	Hoàn thành	Không có
4	Đoàn Minh Cường	Cài đặt code	Hoàn thành	Không có
5	Nông Quốc Việt	Cài đặt code	Hoàn thành	Không có

Bảng 2: *Công việc nhóm*

Phần II
Báo cáo bài tập

1 Composite design pattern

1.1 Design pattern

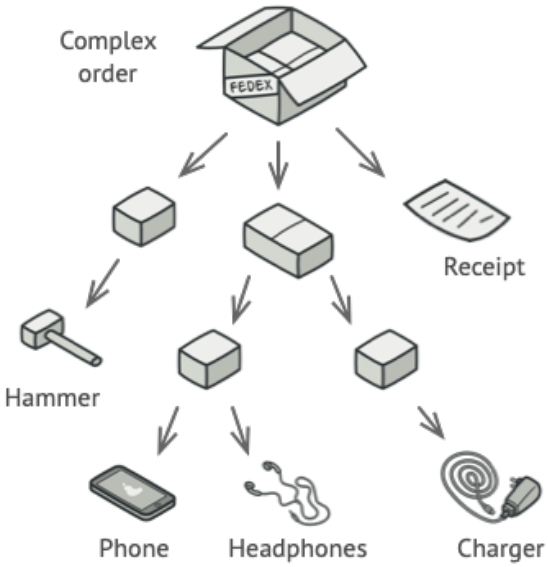
Design pattern là các giải pháp tổng thể đã được tối ưu hóa, được tái sử dụng cho các vấn đề phổ biến trong thiết kế phần mềm mà chúng ta thường gặp phải hàng ngày. Đây là tập các giải pháp đã được suy nghĩ, đã giải quyết trong tình huống cụ thể. Các mẫu thiết kế cung cấp một cách tiêu chuẩn hoặc được chấp nhận rộng rãi để giải quyết các vấn đề thiết kế, giúp tạo ra các hệ thống linh hoạt, dễ bảo trì và mở rộng.

1.2 Composite Design Pattern

Mẫu thiết kế Composite là một mẫu thiết kế cấu trúc (Structural Design Pattern) cho phép bạn xây dựng cây đối tượng phân cấp và làm việc với các phần của nó như là một đối tượng đơn lẻ. Mục tiêu chính của mẫu này là "tạo một giao diện thống nhất cho cả các đối tượng đơn lẻ và các tập hợp của chúng."

1.3 Vấn đề

Ví dụ, tưởng tượng bạn có hai loại đối tượng: Product và Box. Box có thể bao gồm nhiều Product và một số lượng Box nhỏ hơn.Các box nhỏ hơn đó có thể bao gồm Product hoặc Box nhỏ hơn khác nữa. Khi bạn định tạo một hệ thống đặt hàng sử dụng các lớp đấy. Đơn hàng (Order) có thể bao gồm một sản phẩm đơn giản không bị bọc, cũng có thể là một cái hộp đựng sản phẩm và các hộp khác. Vậy làm thế nào để bạn tính tổng giá trị của đơn hàng?

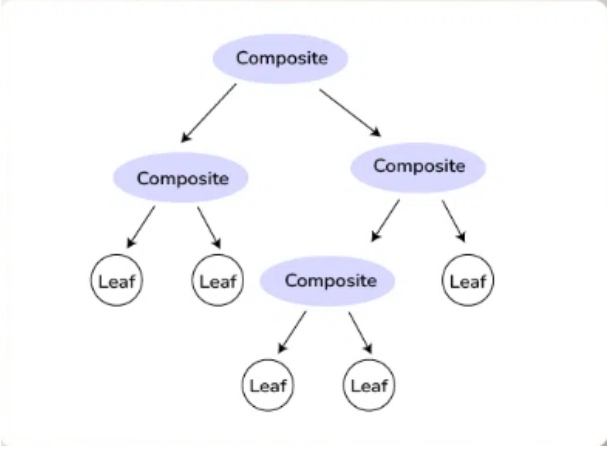


Bạn có thể giải quyết trực tiếp: mở tất cả các hộp và cộng tất cả các sản phẩm lại với nhau. Đây là cách giải quyết trong đời sống thực, nhưng trong các chương trình máy tính nó không đơn giản như vậy. Bạn phải biết lớp của Product và Box mà bạn cần, cấp độ lồng nhau của hộp và các chi tiết khó hiểu khác. Tất cả điều này khiến cách tiếp cận trực tiếp trở nên khó khăn thậm chí là không thể.
=>Sử dụng pattern Composite chỉ hợp lý khi mô hình cốt lõi của ứng dụng bạn có thể biểu diễn dạng cây.

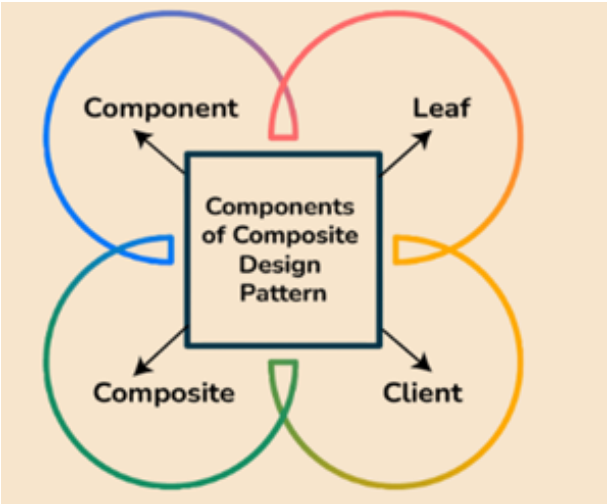
1.4 Giải quyết

- Với sản phẩm đơn giản là trả về giá tiền của các sản phẩm đó.
- Với hộp thì nó sẽ đệ quy đi qua các thành phần sản phẩm trong đó và trả về giá trị tổng của cái hộp (Composite cho phép bạn chạy đệ quy trên tất cả các thành phần của cây đối tượng).
- Lợi ích tuyệt vời của giải pháp này là bạn không cần quan tâm lớp cụ thể của đối tượng cấu thành cây. Bạn không cần biết đối tượng đó là một sản phẩm đơn giản hay một hộp phức tạp. Bạn có thể xử lý như nhau thông qua interface chung. Khi bạn gọi phương thức, đối tượng sẽ tự động chuyển yêu cầu xuống phía dưới cây.

Hình ảnh rõ hơn:

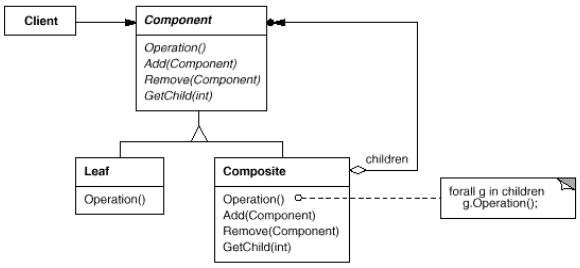


1.5 Các thành phần của Composite Design Patterns



- **Component:** Là giao diện hoặc lớp cơ sở trừu tượng mà tất cả các thành phần trong cây kế thừa (Interface). Nó cung cấp các phương thức chung cho việc quản lý các thành phần con và là thành phần gốc của cây.
- **Leaf:** Là lớp cuối cùng trong cây, không có các thành phần con. Nó triển khai các phương thức của Component.
- **Composite:** Là lớp chứa các thành phần con. Nó triển khai các phương thức của Component và cung cấp cách để thêm hoặc loại bỏ các thành phần con.
- **Client** làm việc với tất cả các phần tử thông qua interface component. Do đó, client có thể làm việc theo cùng một cách với cả các phần tử đơn giản hoặc phức tạp của cây.

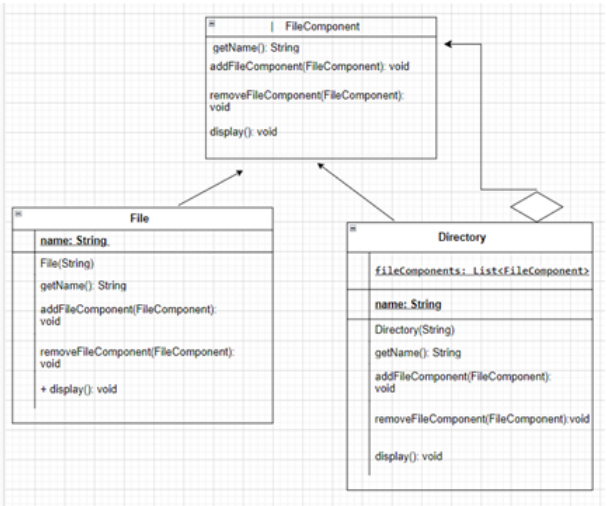
Hình ảnh cấu trúc:



1.6 Triển khai

- Đảm bảo rằng mô hình cốt lõi của ứng dụng có thể biểu diễn cấu trúc cây. Cố gắng chia nhỏ thành các phần tử đơn giản và container. Nhớ là container có thể bao gồm cả phần tử đơn giản và container khác.
- Khai báo interface component với danh sách phương thức hợp lý cho cả component đơn giản và phức tạp.
- Tạo lớp leaf để biểu diễn phần tử đơn giản. Chương trình có thể có nhiều lớp leaf khác nhau.
- Tại lớp container biểu diễn phần tử phức tạp. Trong lớp này, cung cấp mảng để lưu trữ tham chiếu đến phần tử. Mảng có thể lưu cả leaf và container khác, chắc chắn rằng nó được khai báo với kiểu interface component. Trong khi triển khai phương thức của interface component, hãy nhớ container thường uỷ thác phần lớn công việc cho phần tử con.
- Cuối cùng định nghĩa phương thức thêm và xóa phần tử con ở container. Hãy ghi nhớ rằng các hoạt động có thể khai báo ở interface component. Điều này sẽ vi phạm Nguyên tắc Phân tách Interface vì phương thức sẽ trống trong lớp leaf. Tuy nhiên, client sẽ xử lý tất cả phần tử như nhau, khi cấu thành cây.

Ta có ví dụ về quản lý tệp tin:



Trong đó:

- **FileComponent** là giao diện hoặc lớp cơ sở trừu tượng.
- **File** và **Directory** là các lớp cụ thể, **File** là lớp lá (leaf) và **Directory** là lớp composite.
- **Directory** chứa một danh sách các **FileComponent** khác, bao gồm cả các File và Directory.
- Cả **File** và **Directory** đều triển khai các phương thức của **FileComponent**, như **getName()**, **addFileComponent()**, **removeFileComponent()**, và **display()**.

1.7 Ưu nhược điểm của composite design pattern

Ưu điểm:

- **Tính linh hoạt và mở rộng:** Composite cho phép xây dựng các cấu trúc phân cấp phức tạp từ các thành phần đơn giản, giúp tạo ra các cấu trúc linh hoạt và dễ dàng mở rộng.
- **Giao diện thống nhất:** Cung cấp một giao diện thống nhất cho cả các đối tượng và cấu trúc phân cấp, giúp đơn giản hóa mã máy khách và giảm nhu cầu sử dụng các câu lệnh có điều kiện.

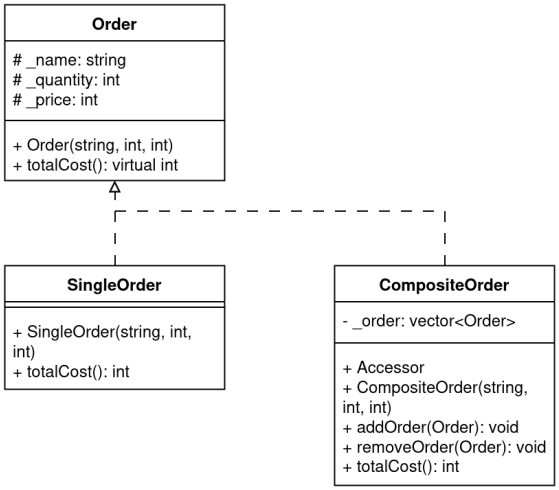
- **Giảm trùng lặp mã:** Bằng cách xác định các hoạt động chung ở cấp độ thành phần, Composite giảm trùng lặp mã và thúc đẩy cách tiếp cận nhất quán để xử lý cả đối tượng lá và đối tượng tổng hợp.
- **Hỗ trợ các hoạt động chung:** Composite hỗ trợ các hoạt động chung trong cấu trúc phân cấp, giúp làm giảm sự phức tạp của mã và tăng tính mô-đun hóa.

Nhược điểm:

- **Độ phức tạp:** Composite có thể trở nên phức tạp khi cấu trúc phân cấp trở nên quá lớn và phức tạp, gây khó khăn trong việc hiểu và bảo trì mã.
- **Khó khăn trong việc hạn chế loại thành phần:** Việc hạn chế loại thành phần của cấu trúc phân cấp có thể trở nên khó khăn và phức tạp hơn khi sử dụng Composite.
- **Công việc cần thêm kiểm tra và xử lý:** Khi muốn một cấu trúc chỉ chứa một số thành phần nhất định, việc thêm kiểm tra và xử lý tăng thêm sự phức tạp cho mã.

2 Ví dụ thực tế

Một đơn hàng shopee có thể chỉ có 1 loại hàng hóa duy nhất hoặc cùng lúc đặt nhiều món hàng hóa khác nhau. Khi này, để tính tiền một đơn hàng, ta triển khai Composite design pattern để xử lý công việc tính tiền cho đơn hàng. Mô hình được triển khai như sau:



Trong đó:

- **Oder:** interface
- **SingleOrder** và **CompositeOder** là các lớp cụ thể, SingleOrder là leaf class, CompositeOder là composite class
- **_name:** tên sản phẩm, **_quantity:** số lượng, **_price:** đơn giá
- **totalCost():** tính tổng tiền đơn hàng = _quantity * _price