Design pattern

1.Dependency injection pattern

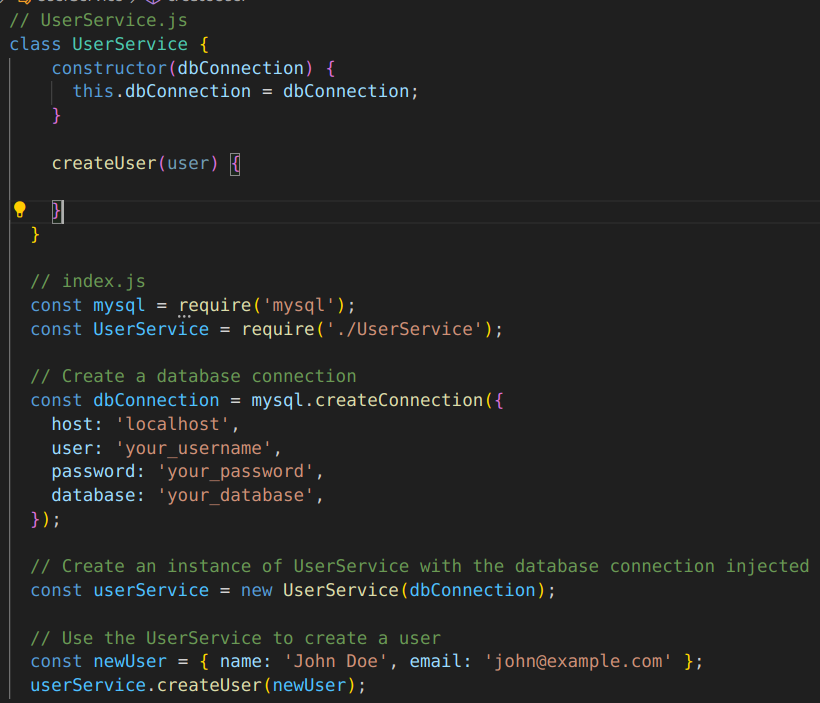
- Dependency Injection(DI) là một mẫu thiết kế phần mềm được sử dụng trong lập trình hướng đối tượng để giảm sự phụ thuộc giữa các thành phần của ứng dụng

- Nó được xây dựng dựa trên nguyên tắc Dependency inversion

-Nó giúp chúng ta có thể linh hoạt thay đổi các phụ thuộc mà không cần phải tạo ra đối tượng mới

-Có 3 cách để triển khai DI được sử dụng là:

+ Thứ nhất: Constructor Injection các phụ thuộc của một đối tượng được chuyển vào thông qua hàm khởi tạo của nó thay vì được tạo ra hoặc truy cập trực tiếp từ bên trong đối tượng đó

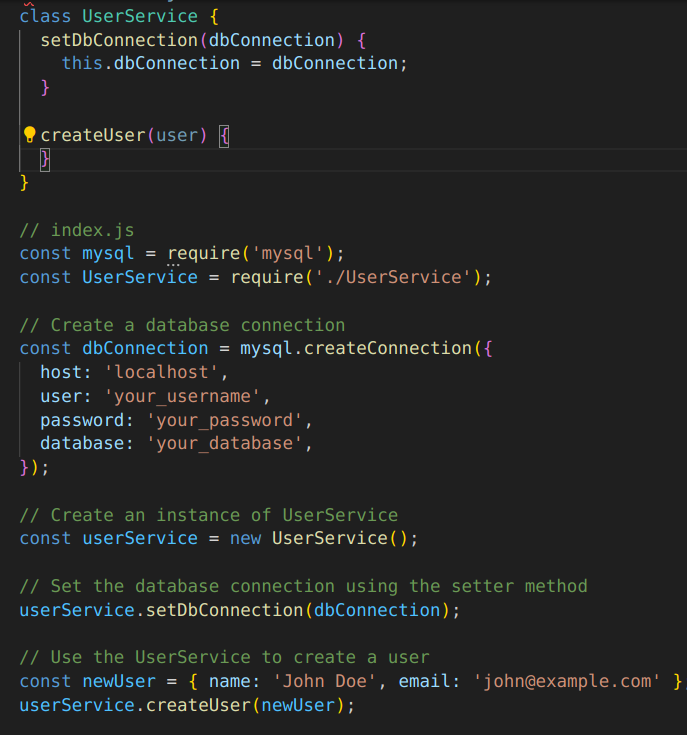
VD: Một ứng dụng web ta có thể triển khai trong các trường hợp

như ví dụ này chúng ta đã tạo ra constructor UserService với chức năng kết nối đến database với phụ thuộc dbconnection. Việc sử dụng constructor injection chúng ta đã truyền dbconnection như 1 tham số của UserService.

Trong file index.js. Đầu tiên chúng ta tạo một kết nối cơ sở dữ liệu bằng thư viện MySQL (dbConnection). Sau đó, chúng ta tạo một đối tượng UserService truyền vào đối tượng dbConnection như một tham số vào hàm tạo. Khi gọi phương thức createUser của đối tượng userService, nó sử dụng dbConnection được inject để thực hiện thao tác cơ sở dữ liệu, trong trường hợp này, là chèn một bản ghi người dùng mới.

Trong constructor injection, các phụ thuộc được chỉ định trong hàm tạo, làm cho ta thấy về những phụ thuộc nào sử dụng trong một lớp. Hơn nữa, nó làm cho các thành phần và module liên kết lỏng lẻo và đơn giản hóa quá trình kiểm thử bằng cách cho phép sử dụng các đối tượng mock hoặc stub để thay cho các phụ thuộc trong quá trình kiểm thử đơn vị(unit testing).

+ Thứ 2: Setter Injection: các phụ thuộc của một đối tượng được chuyển vào thông qua các phương thức setter thay vì thông qua hàm khởi tạo.

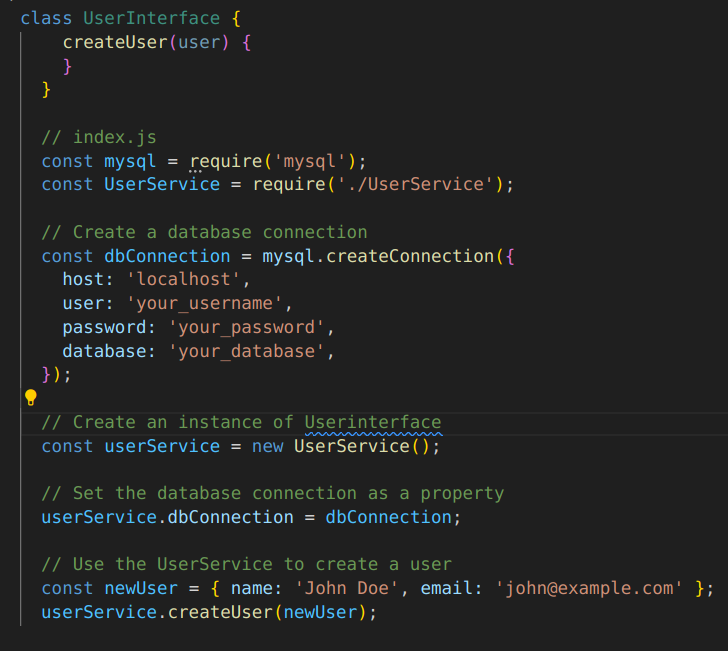


Trong đoạn mã trên, chúng ta có một lớp UserService đòi hỏi một phụ thuộc là dbConnection. Trong tình huống này, chúng ta sử dụng phương thức set của thuộc tính dbConnection để truyền phụ thuộc.

Lớp UserService có một phương thức setDbConnection cho phép chúng ta thiết lập phụ thuộc dbConnection từ bên ngoài. Trong tập tin index.js, chúng ta đầu tiên tạo một kết nối cơ sở dữ liệu bằng thư viện MySQL. Sau đó chúng ta tạo một đối tượng của lớp UserService và sử dụng phương thức **setDbConnection** để truyền vào đối tượng dbConnection. Sau đó, khi gọi phương thức createUser trên đối tượng userService, nó sử dụng dbConnection được inject để thực hiện thao tác cơ sở dữ liệu.

Injection thông qua setter cho phép nhà phát triển sử dụng các phương thức thiết lập để truyền các phụ thuộc. Điều này cho phép bạn cung cấp các phụ thuộc sau khi đối tượng đã được tạo ra, làm cho nó linh hoạt để thiết lập hoặc thay đổi các phụ thuộc khi cần thiết.

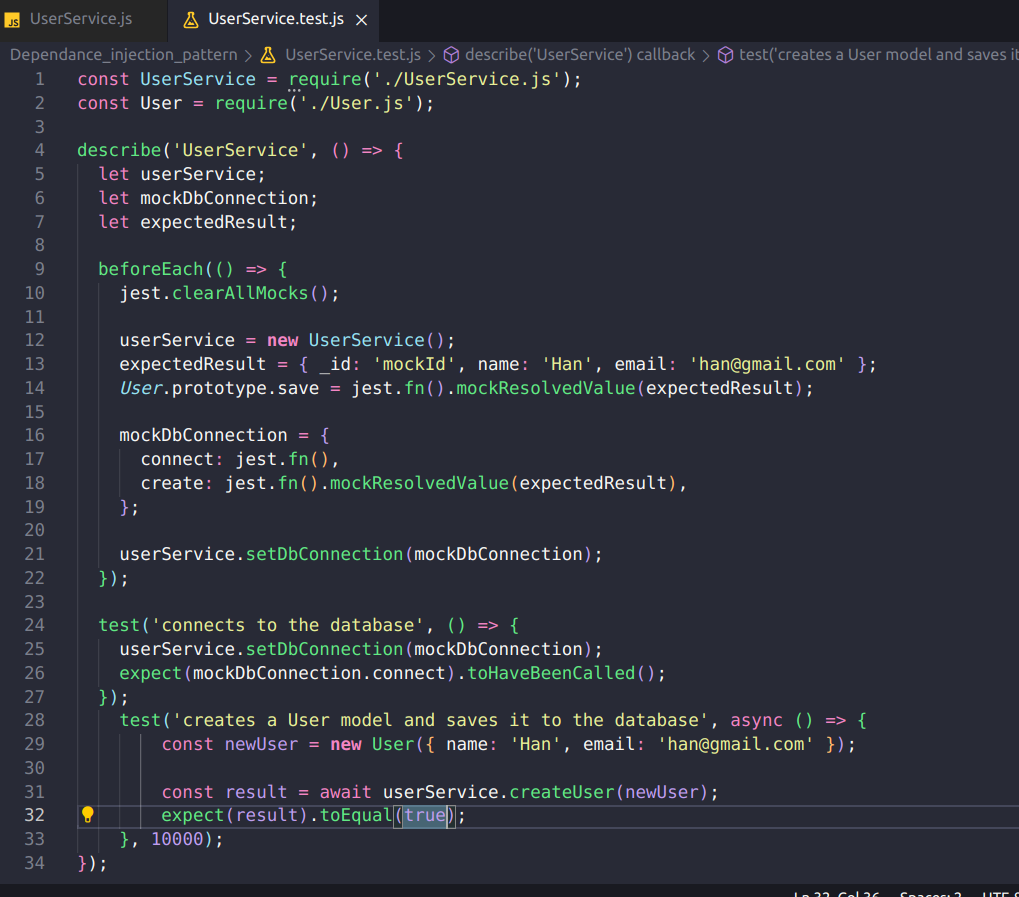
+ Thứ 3: Interface Injection: các phụ thuộc của một đối tượng được chuyển vào thông qua các phương thức interface thay vì thông qua hàm khởi tạo hoặc phương thức setter.



Trong đoạn mã trên, chúng ta có một interface UserInterface đòi hỏi một phụ thuộc là dbConnection để tạo người dùng trong cơ sở dữ liệu. Ở đây, chúng ta đang sử dụng interface injection để truyền phụ thuộc cho UserInterface.

Lớp UserService có một thuộc tính dbConnection, mà chúng ta có thể gán trực tiếp đối tượng dbConnection. Khi gọi phương thức createUser của đối tượng userInterface, nó sử dụng dbConnection được inject để thực hiện thao tác cơ sở dữ liệu.

Như được minh họa trong ví dụ, interface injection cho phép bạn gán trực tiếp các phụ thuộc cho các thuộc tính của một đối tượng. Nó cho sự linh hoạt thay đổi các phụ thuộc và cho phép việc sửa đổi hoặc thay đổi phụ thuộc dễ dàng trong quá trình chạy.

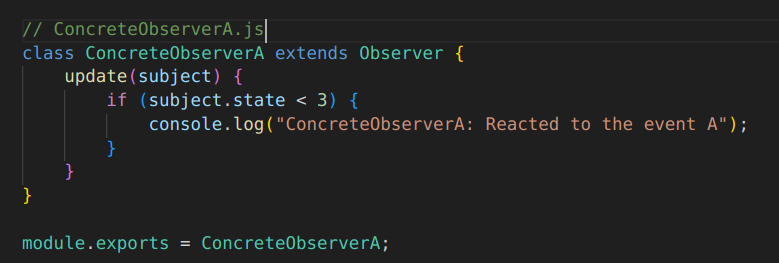
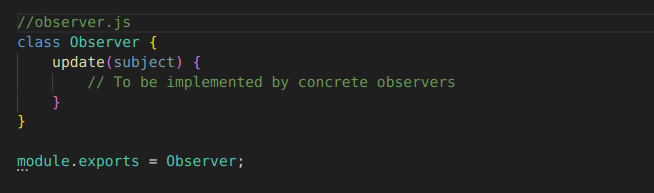
-Không chỉ có vậy việc sử dụng DI còn giúp chúng ta có thể dễ dàng triển khai mock để có thể test chương trình như ví dụ trên ta có thể test các kết nối đến database và hàm createUser() để xem việc tạo các đối tượng mới có lỗi không

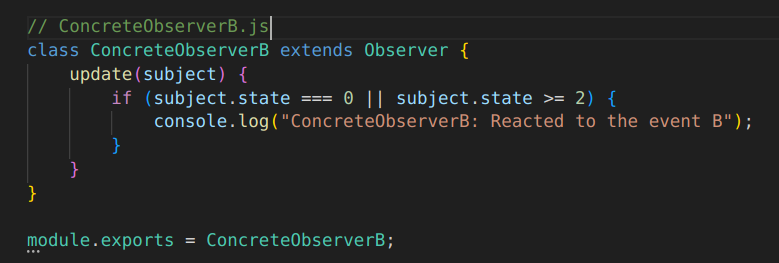
-Sau khi cài đặt thư viện jest chúng ta có thể chạy lệnh npm test để kiểm tra xem việc tạo một đối tượng có thành công hay không nếu thành công đối tượng trả về true nếu sau khi tạo đối tượng kết quả trả về là true thì thành công nếu không sẽ báo lỗi

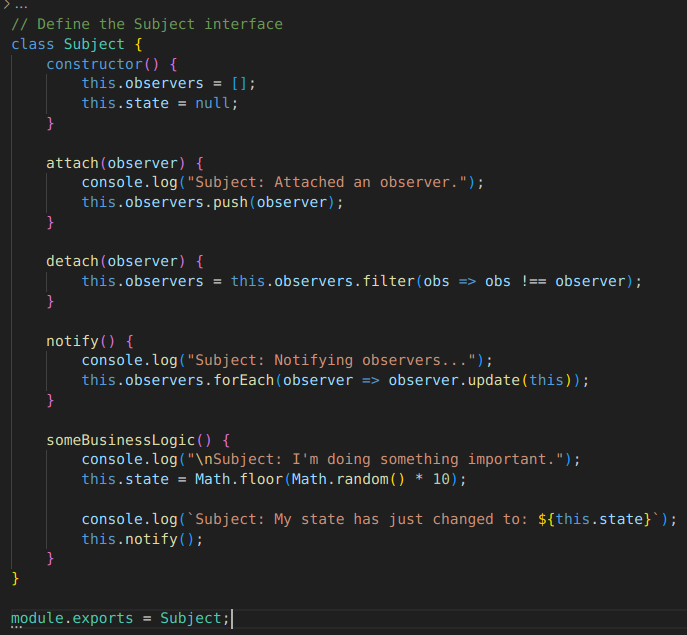
2.Observer pattern

-Observer là một mẫu thiết kế hành vi cho phép bạn xác định cơ chế đăng ký để thông báo cho nhiều đối tượng về bất kỳ sự kiện nào xảy ra trong một đối tượng nào đó.

-Nó là một cơ chế thông báo đến chính xác các đối tượng cần nhận thông tin mỗi khi có gì đó thay đổi

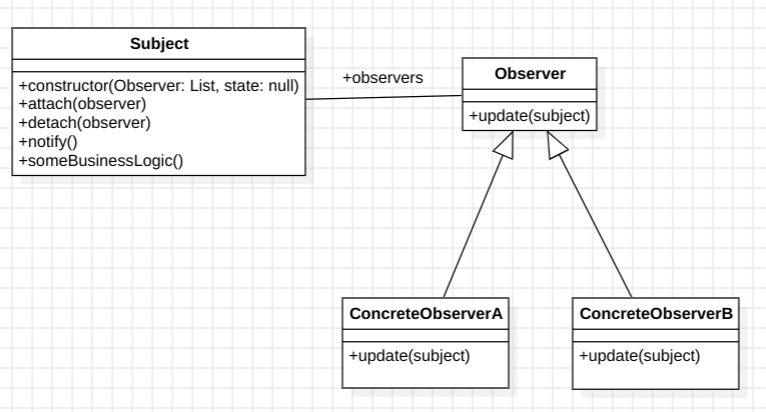








đây là một ví dụ của triển khai observer pattern trong thư viện nodojs



nhìn vào lược đồ này ta thấy việc có thêm 2 đối tượng Subject và Observer giúp cho mỗi khi giá trị hàm **someBusinessLogic()** thay đổi thì **ConcreateObserverA() ConcreateObserverB()** đều nhận được sự thay đổi đó. Nếu sau nay có thêm những thay đổi khác như **ConcreateObserverC**() thì nó cũng sẽ nhận được thông báo thay đổi đó. Việc này giúp việc gửi thông báo những thay đổi trở nên hiệu quả hơn.

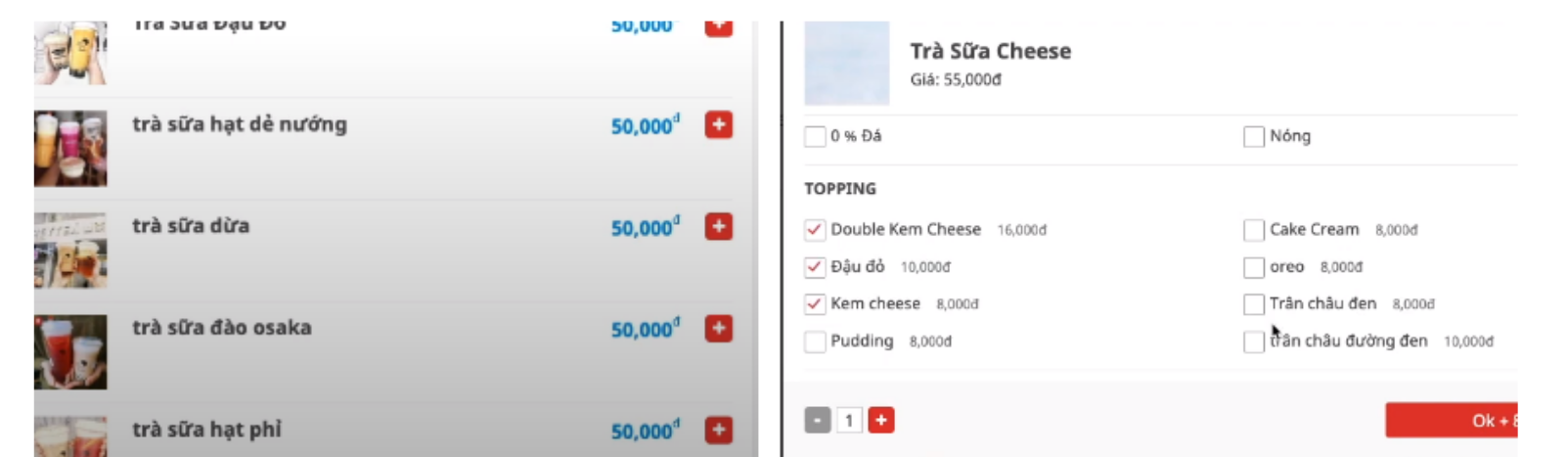
3. Decorator pattern

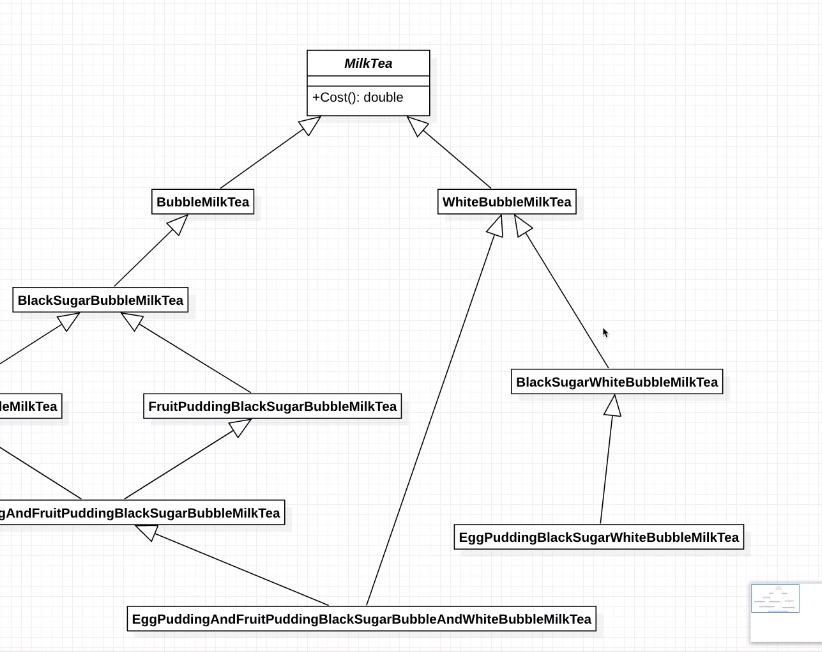
Decorator là một mẫu thiết kế cấu trúc cho phép bạn thêm các hành vi hoặc trạng thái

mới vào các đối tượng bằng cách đặt các đối tượng này vào bên trong các swapper object

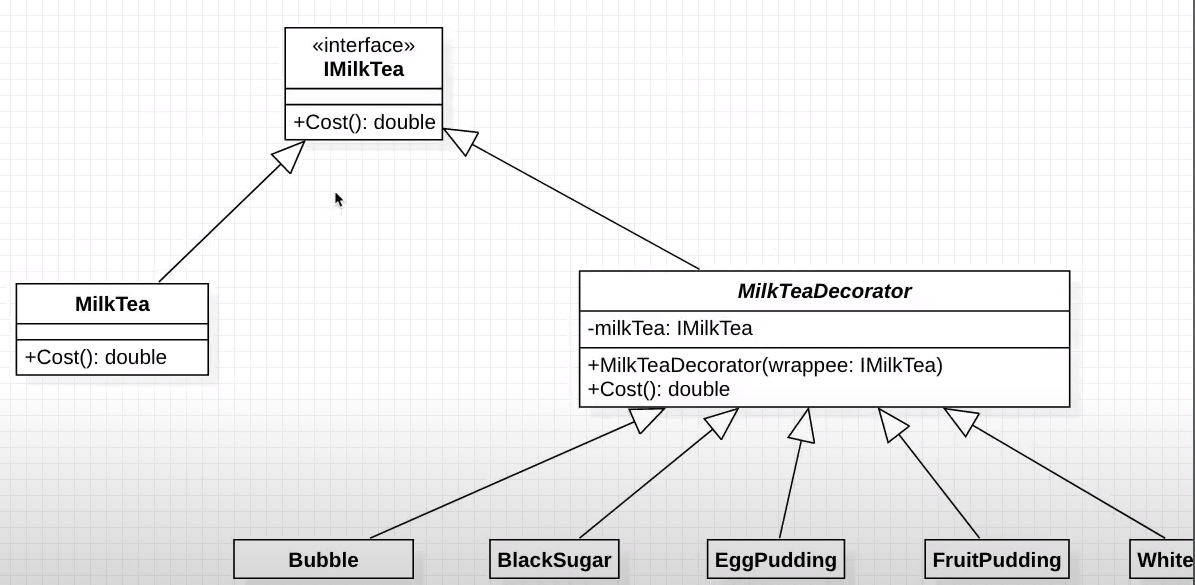
đặc biệt chứa các hành vi hoặc trạng thái.

Decorator pattern được sử dụng trong trường hợp chúng ta muốn tạo ra các đối tượng mới nhưng vẫn kế thừa những thêm các thuộc tính mới mà không phải tạo ra một đối tượng mới. Hãy xét ví dụ này.

Chúng ta có 1 web bán trà sữa nếu chúng ta muốn thêm topping cho các loại trà sữa nếu không áp dụng decorator pattern chúng ta sẽ tạo ra rất nhiều đối tượng vì mỗi loại topping chúng ta sẽ phải tạo ra 1 số lượng vô cùng lớn các loại trà sữa như ví dụ này thì ban đầu chúng ta sẽ khời tạo các đối tượng như sơ đồ UML này:



-Vẫn là vấn đề này nếu chúng ta tổ chức lại chương trình theo decorator pattern thì chương trình chúng ta sẽ rất gọn gàng. Chúng ta có sơ đồ UML mới như này



Để triển khai decorator pattern chúng ta phải khời tạo Interface và 1 core class(MilkTea) và decorator class(MilkTeaDecorator) cùng kế thừa vào interface. Trong constructor của decorator phải có 1 biến là wrappee và nó phải là 1 interface. wrappee nó sẽ lấy hết các thuộc tính ban đầu của milkTea và thêm các thuộc tính mới và milkTea. Khi chạy nếu chúng ta thêm BlackSuger nó sẽ chạy wrappee lấy hết các tính của milkTea và bọc lên thêm các thuộc tính giá trị, thuộc tính mới rồi chạy các tính toán mà wrappee đã có và trả về milkTea kết quả là đối tượng đã được thêm mới giá trị, thuộc tính.