**DECORATOR PATTERN**

Một trong những khái niệm quan trọng khi nói tới Decorator Pattern là tránh chỉnh sữa, dễ mở rộng (closed for modification, open for extension) Chúng ta hãy cùng đi qua một ví dụ và cách sử dụng Decorator Pattern để hiểu rõ về khái niệm này.

Sau đây là ví dụ một trường hợp ta xây dựng 1 class computer.

public class Computer

{

public Computer()

{

}

public String description()

{

**return “You’re getting a computer.”;**

}

}

Khi đối tượng được xây dựng, nó sẽ thể hiện description là “You’re getting a computer”. Tuy nhiên sau một thời gian, computer được nâng cấp lên và bao gồm cả 1 hard disk. Từ đó class computer sẽ được sửa thành

public class Computer

{

public Computer()

{

}

public String description()

{

**return “You’re getting a computer and a disk.”;**

}

}

Lại vẫn vấn đề tương tự khi computer được nâng cấp lên và chứa cả monitor. Lúc đó class computer lại sửa thành

public class Computer

{

public Computer()

{

}

public String description()

{

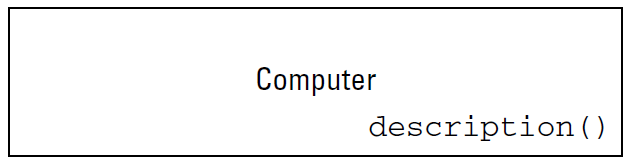
**return “You’re getting a computer and a disk and a monitor.”;**

}

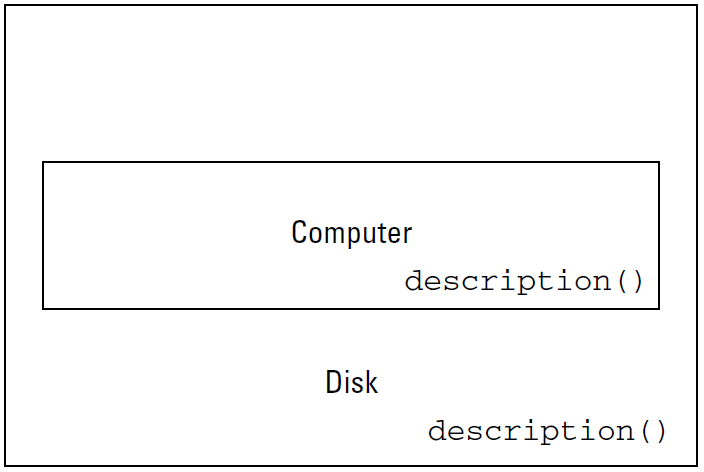
}

Rõ ràng việc thay đổi liên tục trong class computer để đáp ứng với nhu cầu không phải là một cách làm hay. Khi ta xây dựng computer như 1 class core, rõ ràng ta không muốn thay đổi nó quá nhiều. Decorator Pattern giúp ta có thể sử lý việc này và tạo ra một khái niệm code rất bổ ích, tránh sửa chữa core và dễ dàng mở rộng nó. Vậy làm thế nào để có thể áp dụng Decorator Pattern.

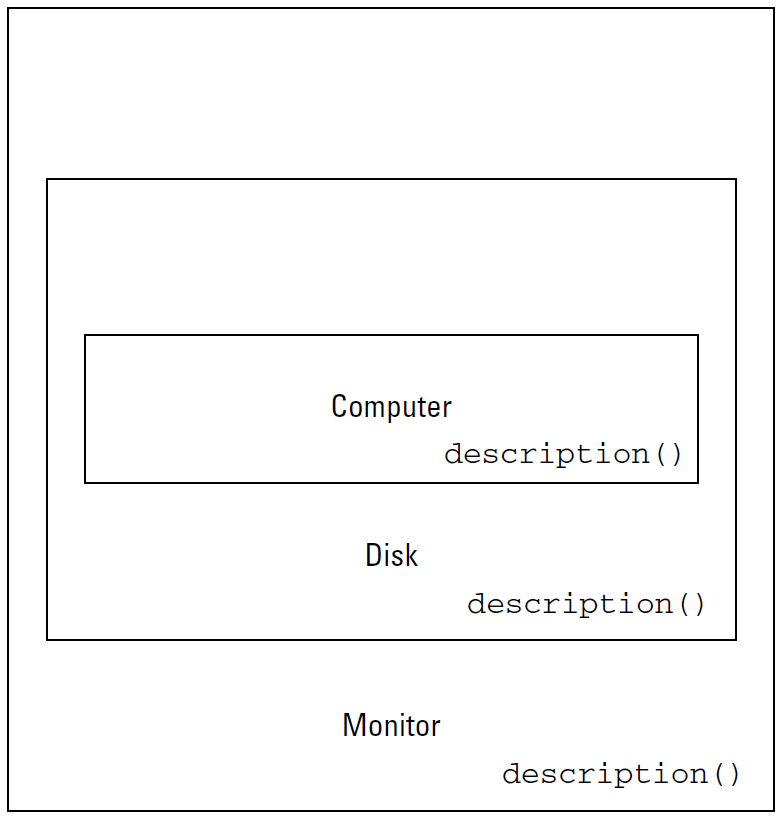
Chúng ta cùng phân tích mô hình và cách thực hiện Decorator Pattern. Ban đầu chúng ta có class Computer



Khi chúng ta gọi description thì computer sẽ trả về “You’re getting a computer”. Bây giờ ta thêm vào hard disk, điều này sẽ được mô hình đơn giản là ta bọc thêm 1 lớp vỏ mới cho computer (bọc bên ngoài để trang trí thêm cho nó)



Bây giờ khi bạn gọi lại description trong computer thì nó phải trả về “You’re getting a computer and disk”. Tiếp theo khi thêm vào tiếp 1 monitor thì ta lại lấy monitor bọc bên ngoài.



Và lúc này khi gọi lại description trong computer thì nó phải trả về “You’re getting a computer and disk and monitor”. Như vậy là sau 2 lần “trang trí” thêm cho computer ta nhận thấy hàm description của nó đã thay đổi khi có thêm 2 “vỏ bọc” disk và monitor. Vậy làm thế nào mà ta có thể thực hiện được việc này, ta hãy cùng phân tích đoạn code thực thi.

Đầu tiên là ta có 1 class core computer

public class Computer

{

public Computer()

{

}

public String description()

{

return “computer”;

}

}

Tiếp theo, ta phân tích một chút, bản thân các vỏ bọc disk và monitor thực ra chỉ thay đổi phần description của computer, do đó các class disk và monitor làm vỏ bọc sẽ chỉ override lại phần description, tuy nhiên ở đây ta cần một cách thực thi tổng quát cho vấn đề này, ta sẽ tạo trước một class abstract thực hiện việc override hàm description()

public abstract class ComponentDecorator extends Computer

{

public abstract String description();

}

Vấn đề tiếp theo của chúng ta chỉ là kế thừa và override hàm description trong các class Disk và Monitor, tuy nhiên ta để ý thêm vấn đề là Disk và Monitor thực ra chỉ là vỏ bọc cho class Computer và phát triển hàm description từ bản thân hàm description đã có sẵn của Computer, do đó ta phải khéo léo xây dựng class disk và monitor có bao gồm của computer bên trong nó

public class Disk extends ComponentDecorator

{

Computer computer;

public Disk(Computer c)

{

computer = c;

}

**public String description()**

**{**

**return computer.description() + “ and a disk”;**

**}**

}

public class Monitor extends ComponentDecorator

{

Computer computer;

public Monitor(Computer c)

{

computer = c;

}

public String description()

{

return computer.description() + “ and a monitor”;

}

}

Như vậy ta thấy là trong class Monitor và Disk có 2 công việc, 1 là xác định con trỏ tới đối tượng computer (thực ra con trỏ là nói trong C++, còn với C# và Java thì ta có thể hiểu ngầm) để lấy ra những thông tin cần thiết (vì thực tế disk và monitor chỉ là decorator), sau đó override lại description() (hàm này cũng của đối tượng computer kế thừa từ class abstract ở trên nên những thay đổi trong hàm này sẽ ảnh hưởng trực tiếp tới hàm description của computer) với các thông số hợp lý (tất nhiên nên chứa các thông số cơ bản của computer vì ta đang decoration nó)

Với cách này ta còn có thể thêm 1 “vỏ bọc” khác nữa ví dụ như thêm CD vào

public class CD extends ComponentDecorator

{

Computer computer;

public CD(Computer c)

{

computer = c;

}

public String description()

{

return computer.description() + “ and a CD”;

}

}

Tiếp theo đây là cách sử dụng, khởi động các decorator này. Ta coi ví dụ sau

public class Test

{

public static void main(String args[])

{

Computer computer = new Computer();

computer = new Disk(computer);

computer = new Monitor(computer);

**computer = new CD(computer);**

**computer = new CD(computer);**

**System.out.println(“You’re getting a “ + computer.description()**

**+ “.”);**

}

}

Đầu tiên, tất nhiên ta phải có đối tượng core là Computer, khi muốn sử dụng một decorator nào, ta đơn giản chỉ việc new đối tượng (decorator đó), và giá trị của computer sẽ thay đổi theo số lượng decorator mà ta xài. Trong ví dụ trên ta thấy ta sử dụng 4 vỏ bọc lần lượt là disk, monitor, CD, CD, do đó kết quả ta nhận được sẽ là

You’re getting a computer and a disk and a monitor and a CD and a CD.

**Tổng kết**

Rõ ràng đây là một pattern rất có ích, trong khi Strategy Pattern giúp ta có những thay đổi phù hợp với các class kế thừa khác nhau từ 1 class core với khái niệm “has-a” và bộ algorithms dễ dàng thay đổi thì Decorator Pattern cho ta khả năng đa dạng hóa class core mà không cần phải thay đổi bản thân class core nhiều. Do đó nói tới Decorator Pattern là nói tới khái niệm “Closed for Modification, Open for Extension”

Một vài lưu ý trong cách sử dụng Decorator Pattern

* Dùng 1 class abstract để override class core, do đó những thành phần trong class abstract phải là những thành phần của class core cần thay đổi (tuy nhiên abstract có nên chứa những hàm riêng hay không thì còn tùy tình huống)
* Class làm decorator thừa kế từ class abstract thực thi các hàm mà nó “bao bọc” và bản thân nó phải có chứa luôn class core