Design pattern

MỤC LỤC

Lời nói đầu 3

A. Tổng quan về Design pattern 4

I. Vấn đề trong thiết kế phần mềm hướng đối tượng 4

II. Lịch sử design pattern 4

III. Design pattern là gì? 5

B. Hệ thống các mẫu design pattern 6

I. Hệ thống các mẫu 6

1. NhómCreational 6

2. Nhóm Structural 6

3. Nhóm Behavioral. 6

4. Sưu liệu chuẩn của mẫu 6

5. Quy tắc biểu diễn mẫu trong UML 7

II.Nội dung các mẫu Design pattern 8

1. Abstract Factory 8

2. Builder 12

3. Factory Method 13

4. Prototype 15

5. Singleton 16

6. Adapter 18

7. Bridge 19

8. Composite 20

9. Decorator 23

10. Façade 24

11. Flyweight 26

12. Proxy 28

13. Chain of Responsibility 30

14. Command 33
15. Interperter 35
16. Iterator 38
17. Mediator 40
18. Memento 43
19. Observer 45
20. State 46
21. Strategy 46
22. Template Method 47
23. Visitor 48
C. Ứng dụng design pattern trong thực tế phân tích thiết kế phần mềm hướng đối tượng 50
I. Framework và idom 50
II. Kiến trúc Add – Ins 51
D.Các mẫu thiết kế hiện đại 52
I. Gamma Patterns 52
II. Entity Pattern (datasim) 52
III. Concurrent Patterns 52
E. Xây dựng ứng dụng Chess sử dụng Design pattern 53
F. Tài liệu tham khảo 53
I. Sách 53
II. Địa chỉ website 53

Lời nói đầu

Design pattern là một kỹ thuật dành cho lập trình hướng đối tượng. Nó cung cấp cho ta cách tư duy trong từng tình huống của việc lập trình hướng đối tượng, và phân tích thiết kế hệ thống phần mềm.Nó cần thiết cho cả các nhà lập trình và nhà phân tích thiết kế. Đối với những người chuyên về lập trình thì việc nắm vững công cụ lập trình thôi chưa đủ,họ cần phải có một tư duy, một kỹ năng giải quyết các tình huống nhỏ của công việc xây dựng phần mềm mà họ là người thi hành.Việc giải quyết này phải đảm bảo tính ổn định là họ có thể giải quyết được trong mọi tình huống, với thời gian đúng tiến độ, phương pháp giải quyết hợp lý và đặc biệt là phải theo một chuẩn nhất định.Những nhà phân tích thiết kế mức cao, việc nắm vững công cụ lập trình có thể là không cần thiết, nhưng họ cũng cần phải biết được ở những khâu nhỏ nhất chi tiết nhất của thiết kế của họ đưa ra có thể thực hiện được hay không và nếu thực hiện được thì có thể thực hiện như thế nào, và sẽ theo một chuẩn ra sao.

Design pattern được dùng khắp ở mọi nơi, trong các phần mềm hướng đối tượng các hệ thống lớn. Trong các chương trình trò chơi, ... Và cả trong các hệ thống tính toán song song,..

Design pattern thể hiện tính kinh nghiệm của công việc lập trình, xây dựng và thiết kế phần mềm.Có thể chúng ta đã gặp design pattern ở đâu đó, trong các ứng dụng, cũng có thể chúng ta đã từng sử dụng những mẫu tương tự như design pattern để giải quyết những tình huống của mình, nhưng chúng ta không có một khái niệm gì về nó cả.Trong nội dung đồ án môn học này chúng tôi xin trình bày những hiểu biết của mình về design pattern theo hướng tiếp cận mang tính kinh nghiệm. Việc cài dặt các mẫu được trình bày trên một tài liệu đi kèm.

Chúng em xin cảm ơn sự hướng dẫn của thầy Nguyễn Ngọc Bình, đã giúp đỡ chúng em hoàn thành đồ án môn học này.

A.Tổng quan về Design pattern.

I.Vấn đề trong thiết kế phần mềm hướng đối tượng

Người ta nói rằng, việc thiết kế một phần mềm hướng đối tượng là một công việc khó, và việc thiết kế một một phần mềm hướng đối tượng phục vụ cho mục đích dùng lại còn khó hơn. Chúng ta phải tìm ra những đối tượng phù hợp,đại diện cho một lớp các đối tượng. Sau đó thiết kế giao diện và cây kế thừa cho chúng, thiết lập mối quan hệ giữa chúng.Thiết kế của chúng ta phải đảm bảo là giải quyết được các vấn đề hiện tại, có thể tiến hành mở rộng trong tương lai mà tránh phải thiết kế lại phần mềm. Và một tiêu trí quan trọng là phải nhỏ gọn. Việc thiết kế một phần mềm hướng đối tượng phục vụ cho mục đích dùng lại là một công việc khó, phức tạp vì vậy chúng ta không thể mong chờ thiết kế của mình sẽ là đúng, và đảm bảo các tiêu trí trên ngay được. Thực tế là nó cần phải được thử nghiệm sau vài lần và sau đó nó sẽ được sửa chữa lại. Đứng trước một vấn đề, một người phân tích thiết kế tốt có thể đưa ra nhiều phương án giải quyết, anh ta phải duyệt qua tất cả các phương án và rồi chọn ra cho mình một phương án tốt nhất.Phương án tốt nhất này sẽ được anh ta dùng đi dùng lại nhiều lần, và dùng mỗi khi gặp vấn đề tương tự. Mà trong phân tích thiết kế phần mềm hướng đối tượng ta luôn gặp lại những vấn đề tương tự như nhau.

II. Lịch sử design pattern

Ý tưởng dùng mẫu xuất phát từ ngành kiến trúc, Alexander, Ishikawa,Silverstein,Jacobson,Fiksdahl-King và Angel (1977) lần đầu tiên đưa ra ý tưởng dùng các mẫu chuẩn trong thiết kế xây dựng và truyền thông. Họ đã xác định và lập sưu liệu các mẫu có liên quan để có thể dùng để giải quyết các vấn đề thường xảy ra trong thiết kế các cao ốc. Mỗi mẫu này là một cách thiết kế, chúng đã được phát triển hàng trăm năm như là các giải pháp cho các vấn đề mà người ta làm trong lĩnh vực xây dựng thường gặp. Các giải pháp tốt nhất có được ngày hôm nay là qua một quá trình sàng lọc tự nhiên. Mặc dù nghành công nghệ phần mềm không có lịch sử phát triển lâu dài như nghành kiến trúc, xây dựng nhưng Công nghệ phần mềm là một nghành công nghiệp tiên tiến, tiếp thu tất cả những gì tốt đẹp nhất từ các nghành khác. Mẫu được xem là giải pháp tốt để giải quyết vấn đề xây dựng hệ thống phần mềm.

Suốt những năm đầu 1990,thiết kế mẫu được thảo luận ở các hội thảo workshop, sau đó người ta nổ lực để đưa ra danh sách các mẫu và lập sưu liệu về chúng. Những người tham gia bị dồn vào việc cần thiết phải cung cấp một số kiểu cấu trúc ở một mức quan niệm cao hơn đối tượng và lớp để cấu trúc này có thể được dùng để tổ chức các lớp. Đây là kết quả của sự nhận thức đươc rằng việc dùng các kỹ thuật hướng đối tượng độc lập sẽ không mang lại những cải tiến đáng kể đối với chất lượng cũng như hiệu quả của công việc phát triển phần mềm. Mẫu được xem là cách tổ chức việc phát triển hướng đối tượng, cách đóng gói các kinh nghiệm của những người đi trước và rất hiệu quả trong thực hành.

Năm 1994 tại hội nghị PLoP(Pattern Language of Programming Design) đã được tổ chức. Cũng trong năm này quyển sách Design patterns : Elements of Reusable Object Oriented Software (Gamma, Johnson,Helm và Vhissdes,1995) đã được xuất bản đúng vào thời điểm diễn ra hội nghị OOPSLA’94. Đây là một tài liệu còn phôi thai trong việc làm nỗi bật ảnh hưởng của mẫu đối với việc phát triển phần mềm, sự đóng

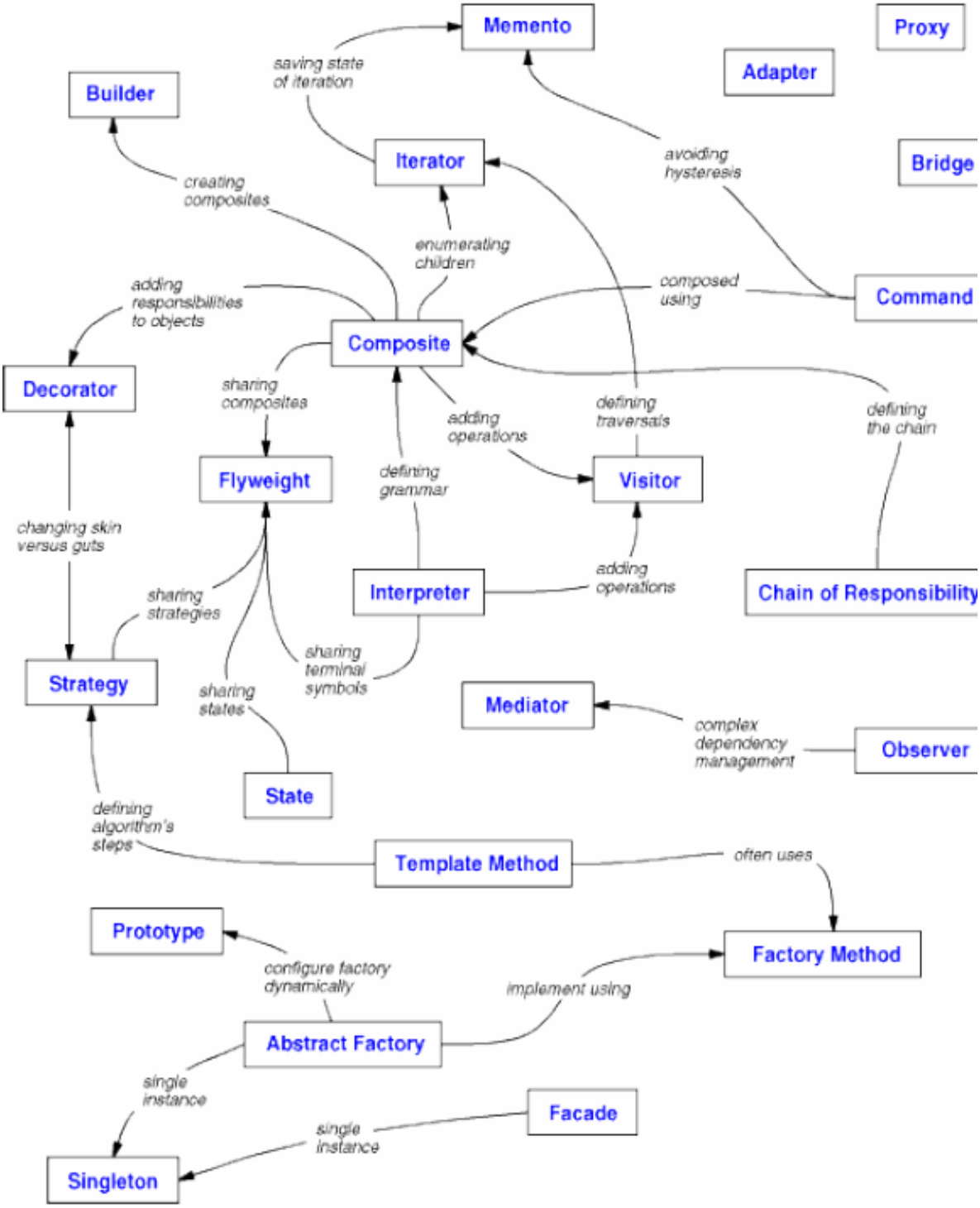
góp của nó là xây dựng các mẫu thành các danh mục (catalogue) với định dạng chuẩn được dùng làm tài liệu cho mỗi mẫu và nổi tiếng với tên Gang of Four (bộ tứ), và các mẫu nó thường được gọi là các mẫu Gang of Four. Còn rất nhiều các cuốn sách khác xuất hiện trong 2 năm sau, và các định dạng chuẩn khác được đưa ra.

Năm 2000 Evitts có tổng kết về cách các mẫu xâm nhập vào thế giới phần mềm (sách của ông lúc bấy giờ chỉ nói về những mẫu có thể được sử dụng trong UML chứ chưa đưa ra khái niệm những mẫu thiết kế một cách tổng quát). Ông công nhận Kent Beck và Ward Cunningham là những người phát triển những mẫu đầu tiên với SmallTalk trong công việc của họ được báo cáo tại hội nghị OOPSLA'87. Có 5 mẫu mà Kent Beck và Ward Cunningham đã tìm ra trong việc kết hợp các người dùng của một hệ thống mà họ đang thiết kế. Năm mẫu này đều được áp dụng để thiết kế giao diện người dùng trong môi trường Windows.

III.Design pattern là gì ?

Design patterns là tập các giải pháp cho cho vấn đề phổ biến trong thiết kế các hệ thống máy tính. Đây là tập các giải pháp đã được công nhận là tài liệu có giá trị, những người phát triển có thể áp dụng giải pháp này để giải quyết các vấn đề tương tự. Giống như với các yêu cầu của thiết kế và phân tích hướng đối tượng (nhằm đạt được khả năng sử dụng các thành phần và thư viện lớp), việc sử dụng các mẫu cũng cần phải đạt được khả năng tái sử dụng các giải pháp chuẩn đối với vấn đề thường xuyên xảy ra.

Christopher Alexander nói rằng :" Mỗi một mẫu mô tả một vấn đề xảy ra lặp đi lặp lại trong môi trường và mô tả cái cốt lõi của giải pháp để cho vấn đề đó.Bằng cách nào đó bạn đã dùng nó cả triệu lần mà không làm giống nhau 2 lần".



Mối quan hệ giữa các Pattern

Design pattern không phải là một phần của UML cốt lõi,nhưng nó lại được sử dụng rộng rãi trong thiết kế hệ thống hướng đối tượng và UML cung cấp các cơ chế biểu diễn mẫu dưới dạng đồ hoạ.

B. Hệ thống các mẫu design pattern.

I. Hệ thống các mẫu

Hệ thống các mẫu design pattern hiện có 23 mẫu được định nghĩa trong cuốn "Design patterns Elements of Reusable Object Oriented Software". Hệ thống các mẫu này có thể nói là đủ và tối ưu cho việc giải quyết hết các vấn đề của bài toán phân tích thiết kế và xây dựng phần mềm trong thời điểm hiện tại.Hệ thống các mẫu design pattern được chia thành 3 nhóm: Creational, nhóm Structural,nhóm behavioral.

1. NhómCreational

Gồm có 5 pattern: AbstractFactory, Abstract Method, Builder, Prototype, và Singleton. Nhóm này liên quan tới việc tạo ra các thể nghiệm (instance) của đối tượng, tách biệt với cách được thực hiện từ ứng dụng. Muốn xem xét thông tin của các mẫu trong nhóm này thì phải dựa vào biểu đồ nào phụ thuộc vào chính mẫu đó, mẫu thiên về hành vi hay cấu trúc.

2. Nhóm Structural

Gồm có 7 mẫu : Adapter, Bridge,Composite,Decorator,Facade,Proxy,và Flyweight.Nhóm này liên quan tới các quan hệ cấu trúc giữa các thể nghiệm, dùng kế thừa,kết tập, tương tác. Để xem thông tin về mẫu này phải dựa vào biểu đồ lớp của mẫu.

3. **Nhóm Behavioral** gồm có 11 mẫu : Interpreter,Template Method,Chain of Responsibility,Command, Iterator,Mediator,Memento,Observer,State,Strategy và Visitor.Nhóm này liên quan đến các quan hệ gán trách nhiệm để cung cấp các chức năng giữa các đối tượng trong hệ thống. Đối với các mẫu thuộc nhóm này ta có thể dựa vào biểu đồ cộng tác và biểu đồ diễn tiến. Biểu đồ cộng tác và biểu đồ diễn tiến sẽ giải thích cho ta cách chuyển giao của các chức năng.

4. Sưu liệu chuẩn của mẫu

Mẫu được phân loại thành 2 nhóm Pattern catalogue (danh mục mẫu) và pattern language (ngôn ngữ mẫu). Một pattern catalogue là một nhóm mẫu có quan hệ với nhau có thể được sử dụng cùng nhau hoặc độc lập. Một pattern language sẽ lập sưu liệu mẫu cho các mẫu làm cùng nhau và có thể được áp dụng để giải quyết các vấn đề trong một lĩnh vực nào đó.Các mẫu được lập sưu liệu bằng cách dùng các template, các template cung cấp các heading bên dưới có chứa chi tiết của mẫu và cách thức nó làm việc cho phép người dùng biết mẫu dó có thích hợp với vấn đề của họ hay không, nếu có thì áp dụng mẫu này để giải quyết vấn đề. Có 4 loại template khác nhau, hai trong số đó thường được sử dụng nhất là Coplien và Gamma.Các heading được liệt kê dưới đây là template của Coplien

- Name – Tên của mẫu, mô tả ý tưởng, giải pháp theo một số cách
- Problem - Vấn đề mà mẫu giúp giải quyết
- Context - Ngữ cảnh ứng dụng của mẫu (kiến trúc hoặc nghiệp vụ) và các yếu tố chính đề mẫu làm việc thành công trong một tình huống nào đó.
- Force – Các ràng buộc hoặc các vấn đề phải được giải quyết bởi mẫu; chúng tạo ra sự mất cân đối, mẫu sẽ giúp ta cân đối.
- Solution - Giải pháp để cân đối các ràng buộc xung đột và làm cho hợp với ngữ cảnh
- Sketch - Bản phác thảo tượng trưng của các ràng buộc và cách giải quyết chúng.
- Resulting context - Ngữ cảnh sau khi thay đổi giải pháp.

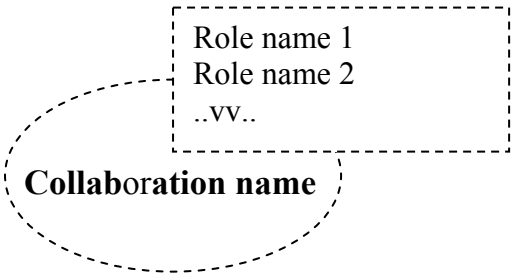
- Rationale – Lý do và động cơ cho mẫu

Sưu liệu có thể gồm mã và các biểu đồ tiêu biểu.Các biểu đồ UML có thể được dùng để minh hoạ cho cách làm việc của từng mẫu.Việc lựa chọn kiểu biểu đồ phụ thuộc vào bản chất của mẫu.

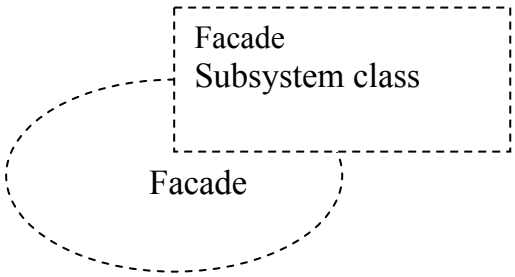
5.Quy tắc biểu diễn mẫu trong UML

Một trong những mục tiêu của UML là hỗ trợ các khái niệm ở cấp cao, như thành phần,cộng tác,framework và mẫu.Việc hỗ trợ này được thực hiện bằng cách cung cấp một cơ chế nhằm định nghĩa rõ ràng ngữ nghĩa của chúng,từ đó việc sử dụng các khái niệm đươc dễ dàng hơn nhằm đạt được khả năng tái sử dụng mà các phương pháp hướng đối tượng yêu cầu. Khía cạnh thuộc cấu trúc của mẫu được biểu diễn trong UML bằng cách dùng các cộng tác mẫu

Cộng tác mẫu được biểu diễn bằng một hình Ellipse nét đứt và một hình chữ nhật nét đứt nằm chồng lên phần cung phía trên bên phải của nó như sau



Ký hiệu cho mẫu cộng tác



Các vai trò liên quan trong mẫu Facade

II.Nội dung các mẫu Design pattern

Nhóm Creational

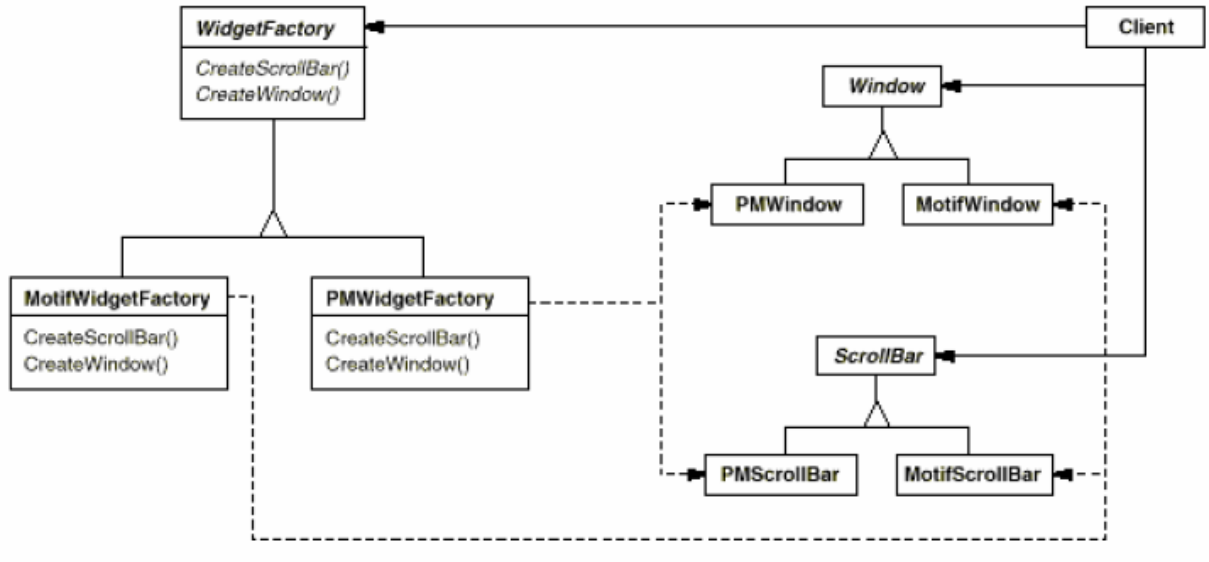
1.Abstract factory:

(tần suất sử dụng : cao trung bình)

a. Vấn đề đặt ra

Chúng ta có thể để ý thấy trong các hệ điều hành giao diện đồ hoạ, một bộ công cụ muốn cung cấp một giao diện người dùng dựa trên chuẩn look - and – feel, chẳng hạn như chương trình trình diễn tài liệu power point.Có rất nhiều kiểu giao diện look-and –feel và cả những hành vi giao diện người dùng khác nhau được thể hiện ở đây như thanh cuộn tài liệu (scroll bar), cửa sổ (window), nút bấm (button), hộp soạn thảo (editbox),...Nếu xem chúng là các đối tượng thì chúng ta thấy chúng có một số đặc điểm và hành vi khá giống nhau về mặt hình thức nhưng lại khác nhau về cách thực hiện. Chẳng hạn đối tượng button và window, editbox có cùng các thuộc tính là chiều rộng, chiều cao,toạ độ,… Có các phương thức là Resize(), SetPosition(),...Tuy nhiên các đối tượng này không thể gộp chung vào một lớp được vì theo nguyên lý xây dựng lớp thì các đối tượng thuộc lớp phải có các phương thức hoạt động giống nhau. Trong khi ở đây tuy rằng các đối tượng có cùng giao diện nhưng cách thực hiện các hành vi tương ứng lại hoàn toàn khác nhau.

Vấn đề đặt ra là phải xây dựng một lớp tổng quát, có thể chứa hết được những điểm chung của các đối tượng này để từ đó có thể dễ dàng sử dụng lại, ta gọi lớp này là WidgetFactory.Các lớp của các đối tượng window, button,editbox kế thừa từ lớp này.Trong thiết kế hướng đối tượng, xây dựng một mô hình các lớp như thế được tối ưu hoá như sau:

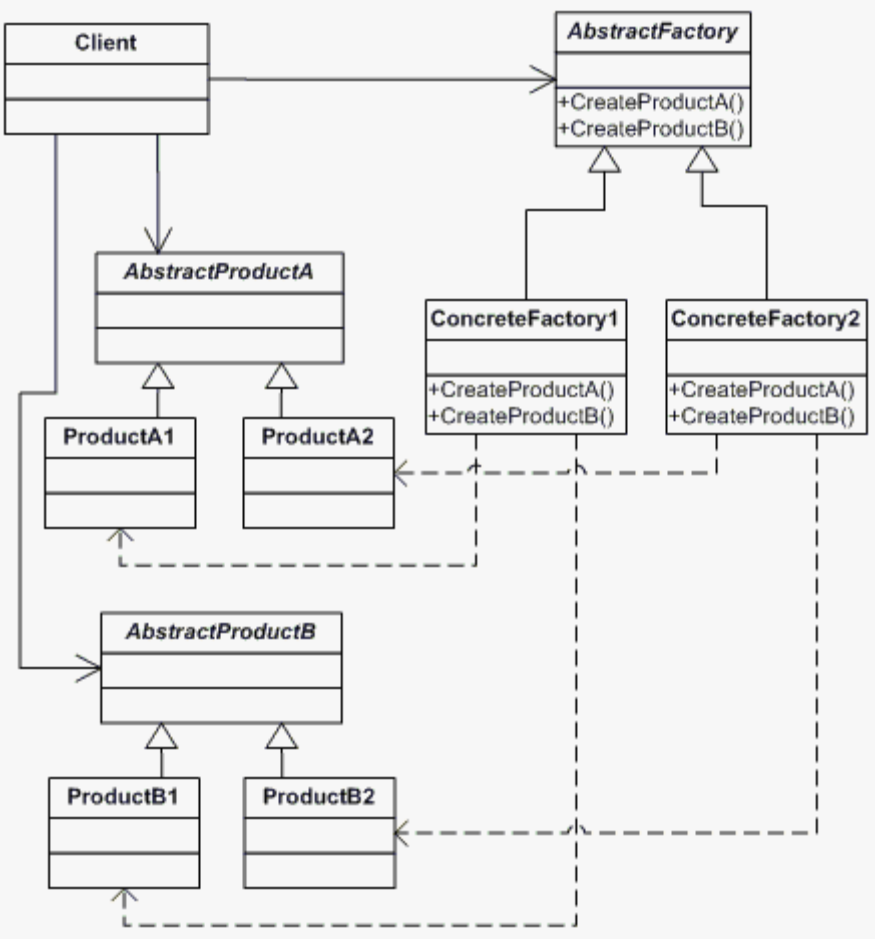


Lớp WidgetFactory có 2 phương thức là CreateScrollBar() và CreateWindow()đây là lớp giao diện trừu tượng tổng quát cho tất cả các MotifWidgetFactory và PMWidgetFactory. Các lớp
MotifWidgeFactory và PMWidgetFactory kế thừa trực tiếp từ lớp WidgetFactory. Trong sơ đồ trên còn có 2 nhóm lớp Window và ScrollBar, chúng đều là các lớp trừu tượng. Từ lớp Window sinh ra các lớp con cụ thể là PMWindow và MotifWindow. Từ lớp ScrollBar sinh ra các lớp con cụ thể là PMScrollBar và MotifScrollBar.Các đối tượng thuộc lớp này được các đối tượng thuộc lớp Factory (MotifWidgetFactory và PMWidgetFactory) gọi trong các hàm tạo đối tượng. Đối tượng trong ứng dụng (đối tượng khách - client) chỉ thông qua lớp giao diện của các đối tượng MotifWidgetFactory và PMWidgetFactory và các đối tượng trừu tượng Window và ScrollBar để làm việc với các đối tượng PMWindow, MotifWindow, PMScrollBar,MotifScrollBar. Điều này có được nhờ cơ chế binding trong các ngôn ngữ hỗ trợ lập trình hướng đối tượng như C++,C#,Java, Small Talk,…Các đối tượng PMWindow, MotifWindow, PMScrollBar,MotifScrollBar được sinh ra vào thời gian chạy chương trình, nên trình ứng dụng (đối tượng thuộc lớp client) chỉ cần giữ một con trỏ trỏ đến đối tượng thuộc lớp WidgetFactory, và thay đổi địa chỉ trỏ đến nó có thể làm việc với tất cả các đối tượng ở trên.Những tình huống thiết kế như thế này thường có cùng một cách giải quyết đã được chứng tỏ là tối ưu. Nó được tổng quát hoá thành một mẫu thiết kế gọi là **AbstractFactory.**

b. Định nghĩa:

Mẫu AbstractFactory là một mẫu thiết kế mà cung cấp cho trình khách một giao diện cho một họ hoặc một tập các đối tượng thuộc các lớp khác nhau nhưng có cùng chung giao diện với nhau mà không phải trực tiếp làm việc với từng lớp con cụ thể.

c. Lược đồ UML



AbstractFactory (ContinentFactory)

Khai báo một giao diện cho các thao tác để tạo ra các dẫn xuất trừu tượng

ConcreteFactory (AfricaFactory, AmericaFactory)

Cài đặt các thao tác để tạo ra các đối tượng dẫn xuất chi tiết

AbstractProduct (Herbivore, Carnivore)

Khai báo một giao diện cho một kiểu đối tượng dẫn xuất

Product (Wildebeest, Lion, Bison, Wolf)

Định nghĩa một đối tượng dẫn xuất được tạo ra bởi một factory cụ thể tương ứng.

Cài đặt giao diện AbstractProduct

Client (AnimalWorld)

Sử dụng giao diện được khai báo bởi các lớp AbstractFactory và AbstractProduct

Cài đặt cho lớp WidgetFactory:

```
class WidgetFactory
{
public:
        virtual Window* CreateWindow();
        virtual ScrollBar* CreateScrollBar();
};
```

Cài đặt cho lớp MotifWidgetFactory và PMWidgeFactory như sau:

```
class MotifWidgetFactory:public WidgetFactory
{
public:
        Window* CreateWindow()
        {
                return new MotifWindow();
        }
        ScrollBar* CreateScrollBar()
        {
                return new MotifScrollBar();
        }
};
class PMWidgetFactory:public WidgetFactory
{
public:
        Window* CreateWindow()
        {
                return new PMWindow();
        }
        ScrollBar* CreateScrollBar()
        {
                return new PMScrollBar();
        }
};
```

Trong đó các lớp đối tượng Window được định nghĩa như sau:

```
class Window
{
        //Các thuộc tính và các phương thức tĩnh và ảo định nghĩa tại đây
};
class MotifWindow:public Window
{
        //Các thuộc tính và các phương thức định nghĩa tại đây
};
class PMWindow:public Window
{
        //Các thuộc tính và các phương thức định nghĩa tại đây
};
```

Các lớp thuộc nhóm ScrollBar.

```
class ScrollBar
{
        //Các thuộc tính và các phương thức tĩnh và ảo định nghĩa tại đây
};
class MotifScrollBar:public ScrollBar
```

```
{
        //Các thuộc tính và các phương thức định nghĩa tại đây
};
class PMScrollBar:public ScrollBar
{
        //Các thuộc tính và các phương thức định nghĩa tại đây
};
```

Để truy cập đến các đối tượng này tại chương trình ứng dụng ta có thể thực hiện như sau:

```
class Client
{
private:
        WidgetFactory* wf;
};
```

d. Mẫu liên quan:

AbstractFactory thường được cài đặt cùng với singleton, FactoryMethod đôi khi còn dùng cả Prototype. Các lớp con cụ thể (concrete class) thường được cài đặt bằng singleton.Bởi singleton có thể tạo ra những đối tượng đồng nhất cho dù chúng ta gọi nó ở đâu trong chương trình.Các mẫu này sẽ được nói kỹ hơn ở các phần sau.

2. Builder

(tần suất sử dụng : trung bình)

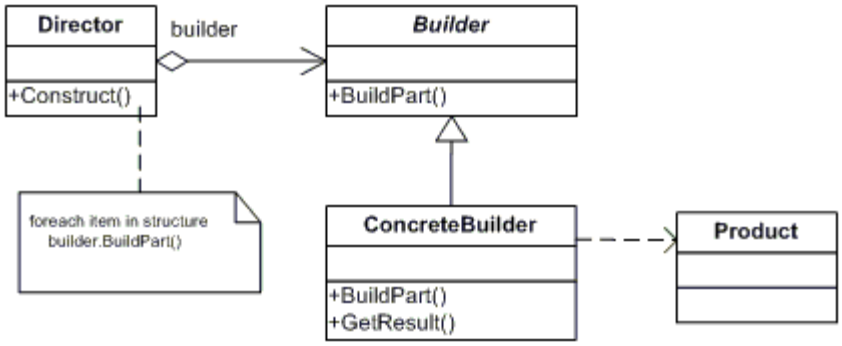
a. Vấn đề đặt ra

Trong những ứng dụng lớn, với nhiều các chức năng phức tạp và mô hình giao diện đồ sộ.Việc khởi tạo ứng dụng thường gặp nhiều khó khăn. Chúng ta không thể dồn tất cả công việc khởi tạo này cho một hàm khởi tạo .Vì như thế sẽ rất khó kiểm soát hết, và hơn nữa không phải lúc nào các thành phần của ứng dụng cũng được khởi tạo một cách đồng bộ. Có thành phần được tạo ra vào lúc dịch chương trình nhưng cũng có thành phần tuỳ theo từng yêu cầu của người dùng, tuỳ vào hoàn cảnh của ứng dụng, mà nó sẽ được tạo ra. Do vậy người ta giao công việc này cho một đối tượng chịu trách nhiêm khởi tạo, và chia việc khởi tạo ứng dụng riêng rẽ, để có thể tiến hành khởi tạo riêng biệt ở các hoàn cảnh khác nhau. Hãy tưởng tượng việc tạo ra đối tượng của ta giống như như việc chúng ta tạo ra chiếc xe đạp. Đầu tiên ta tạo ra khung xe, sau đó tạo ra bánh xe, chúng ta tạo ra budông xe, xích, líp,.. Việc tạo ra các bộ phận này không nhất thiết phải đươc thực hiện một cách đồng thời hay theo một trật tự nào cả, và nó có thể được tạo ra một cách độc lập bởi nhiều người. Nhưng trong một mô hình sản xuất như vậy, bao giờ việc tạo ra chiếc xe cũng được khép kín để tạo ra chiếc xe hoàn chỉnh, đó là nhà máy sản xuất xe đạp.Ta gọi đối tượng nhà máy sản xuất xe đạp này là builder (người xây dựng).

b. Định nghĩa

Builder là mẫu thiết kế hướng đối tượng được tạo ra để chia một công việc khởi tạo phức tạp của một đối tượng ra riêng rẽ từ đó có thể tiến hành khởi tạo đối tượng ở các hoàn cảnh khác nhau.

c. Sơ đồ UML:



Builder (VehicleBuilder)

- Chỉ ra một giao diện trừu tượng cho việc tạo ra các phần của một đối tượng Product.

ConcreteBuilder (MotorCycleBuilder, CarBuilder, ScooterBuilder)

- Xây dựng và lắp ráp các phần của dẫn xuất bằng việc cài đặt bổ sung giao diện Builder.
- Định nghĩa và giữ liên kết đến đại diện mà nó tạo ra.
- Cung cấp một giao diện cho việc gọi dẫn xuất ra.

Director (Shop)

- Xây dựng một đối tượng sử dụng giao diện Builder.

Product (Vehicle)

- Biểu diễn các đối tượng phức tạp.ConcreteBuilder dựng nên các đại diện bên trong của dẫn xuất và định nghĩa quá trình xử lý bằng các thành phần lắp ráp của nó.
- Gộp các lớp mà định nghĩa các bộ phận cấu thành, bao gồm các giao diện cho việc lắp ráp các bộ phận trong kết quả cuối cùng.

d.Các mẫu liên quan

Builder thường được cài đặt cùng với các mẫu như Abstract Factory .Tầm quan trọng của Abstract Factory là tạo ra một dòng các đối tượng dẫn xuất (cả đơn giản và phức tạp).Ngoài ra Builder còn thường được cài đặt kèm với Composite pattern. Composite pattern thường là những gì mà Builder tạo ra.

3.Factory Method

(tần suất sử dụng :cao trung bình)

a. Vấn đề đặt ra

Các Framework thường sử dụng các lớp trừu tượng để định nghĩa và duy trì mối quan hệ giữa các đối tượng.Một framework thường đảm nhiệm việc tạo ra các đối tượng hoàn chỉnh. Việc xây dựng một framework cho ứng dụng mà có thể đại diện cho nhiều đối tượng tài liệu cho người dùng. Có 2 loại lớp trừu tượng chủ chốt trong framework này là lớp ứng dụng và tài liệu.Cả 2 lớp đều là lớp trừu tượng, và trình khách phải xây dựng các dẫn xuất, các lớp con để hiện thực hoá, tạo ra đối tượng phù hợp với yêu cầu của ứng dụng. Chẳng hạn để tạo ra một ứng dụng drawing, chúng ta định nghĩa một lớp DrawingApplication và một lớp DrawingDocument. Lớp ứng dụng

chịu trách nhiệm quản lý tài liệu và chúng ta sẽ tạo ra chúng khi có nhu cầu (chẳng hạn khi người dùng chọn Open hoặc New từ menu).

Bởi vì một lớp Document cá biệt như thế này được tạo ra từ các dẫn xuất của lớp Abstract Document (trong framework) để đưa ra các thể nghiệm cho một ứng dụng Drawing, lớp ứng dụng không thể biết trước được lớp dẫn xuất của Abstract Document nào sẽ được tạo ra để trình bày, mà nó chỉ biết khi nào một đối tượng tài liệu nào nên được tạo chứ không biết loại đối tượng tài liệu nào để tạo. Điều này tạo ra một sự tiến thoái lưỡng nan: framework phải thể nghiệm một lớp, nhưng nó chỉ biết về lớp trừu tượng của nó, mà lớp trừu tượng này lại không thể thể nghiệm trực tiếp được.Nếu ai từng làm việc với giao diện ứng dụng đơn tài liệu và đa tài liệu trong ngôn ngữ Visual C++, chúng ta cũng sẽ bắt gặp một vấn đề tương tự. Đối tượng MainFrame có thể tạo ra một dối tượng view mỗi khi người dùng nhấn chuột vào menu View hay Open, nhưng MainFrame hoàn toàn không hề biết một chút thông tin nào trước đó về View vì nó không chứa bất cứ một thể nghiệm nào của View.

Mẫu Abstract Method đưa ra một giải pháp cho vấn đề này.Nó đóng gói thông tin về lớp dẫn xuất Document nào được tạo và đưa ra ngoài framework.

Lớp dẫn xuất ứng dụng định nghĩa lại một phương thức trừu tượng CreateDocument() trên lớp Application để trả về một đối tượng thuộc lớp dẫn xuất của lớp document.

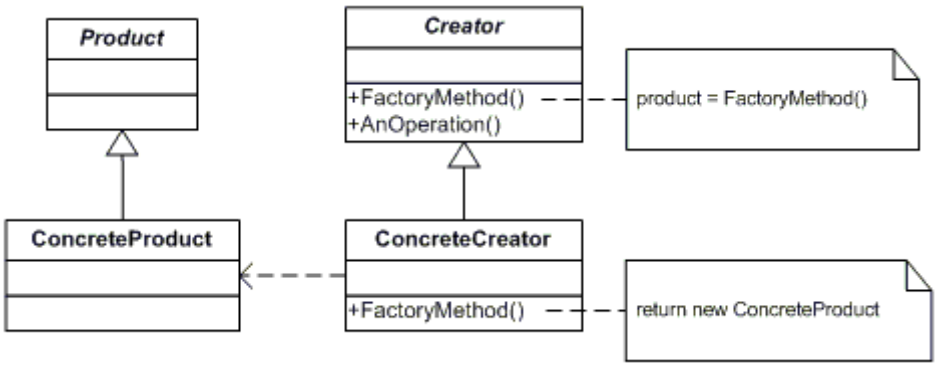
```
Class Application
{
        //Các khai báo khác
Public:
        Document* CreateDocument();
        //Các khai báo khác
};
Document* Application::CreateDocument()
{
        Document* newDocument = new MyDocument();
        Return newDocument;
}
```

Khi một đối tượng thuộc lớp dẫn xuất của Application được tạo ra, nó có thể tạo ra luôn các đối tượng tài liệu đặc biệt cho ứng dụng mà không cần biết về các lớp đó.Chúng ta gọi CreateDocument là một factory method bởi vì nhiệm vụ của nó là sản xuất ra các đối tượng.

b. Định nghĩa Factory Method

Factory Method là một giao diện cho việc tạo ra một đối tượng, nhưng để cho lớp dẫn xuất quyết định lớp nào sẽ được tạo.Factory method để cho một lớp trì hoãn sự thể nghiệm một lớp con.

c. Sơ đồ UML



Product (Page)

- Định nghĩa giao diện của các đối tượng mà Factory Method tạo ra.

ConcreteProduct (SkillsPage, EducationPage, ExperiencePage)

- Cài đặt giao diện Product.

Creator (Document)

- Khai báo Factory Method mà trả về một đối tượng của kiểu Product. Sự kiến tạo này cũng có thể định nghĩa một cài đặt mặc định của Factory Method trả về một đối tượng ConcreteProduct mặc định.
- Có thể gọi Factory Method để tạo ra một đối tượng Product.

ConcreteCreator (Report, Resume)

- Chồng lên Factory Method để trả về một thể nghiệm của một ConcreteProduct.

d. Mẫu liên quan

Abstract Factory thường được cài đặt cùng với Factory Method. Lớp Factory Method thường được gọi là Template Method. Trong ví dụ về ứng dụng Drawing trên,NewDocument là một template method.

Prototype không cần đến một lớp con Creator, tuy nhiên thường đòi hỏi một phương thức để tạo thể nghiệm trên lớp dẫn xuất.

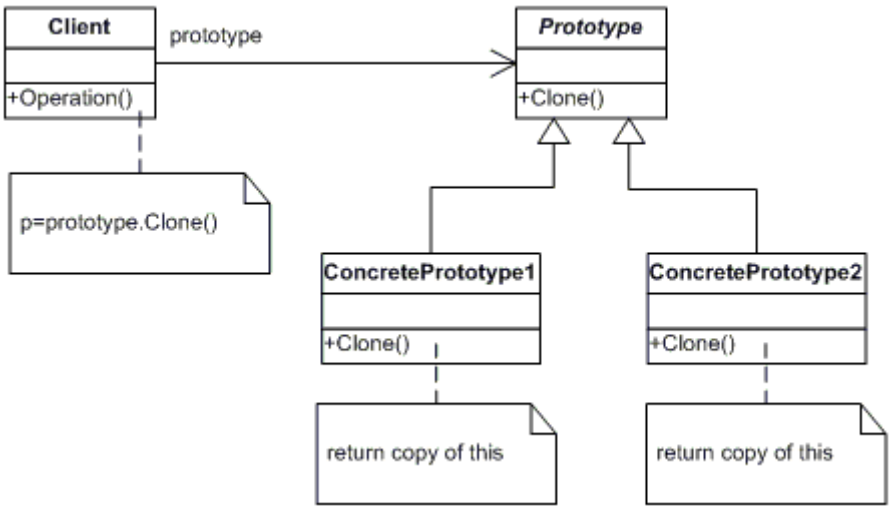
4. Prototype

(tần suất sử dụng : thấp trung bình)

a. Định nghĩa

Prototype là mẫu thiết kế chỉ định ra một đối tượng đặc biệt để khởi tạo, nó sử dụng một thể nghiệm sơ khai rồi sau đó sao chép ra các đối tượng khác từ mẫu đối tượng này.

b.Sơ đồ UML



Prototype (ColorPrototype)

- Khai báo một giao diện cho dòng vô tính của chính nó.

ConcretePrototype (Color)

- Cài đặt một thao tác cho dòng vô tính của chính nó.

Client (ColorManager)

- Tạo ra một đối tượng mới bằng việc yêu cầu một nguyên mẫu từ dòng vô tính của nó

c. Mẫu liên quan

Prototype và Abstract Factory liên quan đến nhau chặt chẽ, có thể đối chọi nhau theo nhiều kiểu.Tuy nhiên chúng cũng có thể kết hợp cùng nhau.Một Abstract Factory có thể chứa một tập các Prototype vô tính và trả về các đối tượng sản xuất.

5. Singleton

(tần suất sử dụng :Cao)

a. Vấn đề đặt ra

Ta hãy xem xét về một đối tượng quản lý tài nguyên trong các ứng dụng. Mỗi ứng dụng có một bộ quản lý tài nguyên, nó cung cấp các điểm truy cập cho các đối tượng khác trong ứng dụng. Các đối tượng (ta gọi là đối tượng khách) có thể thực hiện lấy ra từ bộ quản lý tài nguyên những gì chúng cần và thay đổi giá trị nằm bên trong bộ quản lý tài nguyên đó. Để truy cập vào bộ quản lý tài nguyên đối tượng khách cần phải có một thể nghiệm của bộ quản lý tài nguyên, như vậy trong một ứng dụng sẽ có rất nhiều thể nghiệm của bộ quản lý tài nguyên được tạo ra.

```
class ResourceManager
{
private:
        int x;
public:
        ResourceManager(){x =0;....}
        void SetX(int _x){ x = _x;}
        void GetX(){ return x;}
}
class ClientObject1
```

```
{
public:
        void DoSomething()
        {
        ResourceManager rm;
                printf("x = %d",rm.GetX());
        x = 100;
        }
}
class  ClientObject2
{
public:
        void DoSomething()
        {
        ResourceManager rm;
                printf("x = %d",rm.GetX());
        x = 500;
        }
}
```

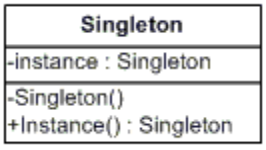
Trong ví dụ trên hàm DoSomething() của ClientObject1 khi truy cập vào đối tượng thuộc lớp ResourceManager sẽ in ra màn hình X = 0; và đặt vào đó x = 100;

Sau đó hàm Dosomething() của ClientObject2 khi truy cập vào đối tượng thuộc lớp ResourceManager sẽ in ra màn hình X = 0 và đặt vào đó x = 500; Rõ ràng là tài nguyên mà các đối tượng khách truy cập vào không thể hiện sự thống nhất lẫn nhau. Điều mà lẽ ra khi giá trị x trong ResourceManager bị ClientObject1 thay đổi thì ClientObject2 phải nhận biết được điều đó và in ra màn hình X=100 . Nhưng theo logic cài đặt ở trên thì khi đối tượng thuộc lớp ClientObject1 truy cập đến ResourceManager nó tạo ra một Instance và đặt thuộc tính x = 100; Đối tượng ClientObject2 truy cập đến ResourceManager nó tạo ra một Instance và đặt thuộc tính x = 500. Hai Instance này hoàn toàn độc lập nhau về vùng nhớ do đó mà tài nguyên x mà nó quản lý cũng là 2 tài nguyên độc lập với nhau.Vấn đề đặt ra là phải tạo ra một bộ quản lý tài nguyên hoàn hảo tạo ra mọi thể nghiệm giống nhau tại nhiều nơi nhiều thời điểm khác nhau trong ứng dụng.Singleton cung cấp cho ta một cách giải quyết vấn đề này.

b.Định nghĩa

Singleton là mẫu thiết kế nhằm đảm bảo chỉ có duy nhất một thể nghiệm và cung cấp điểm truy cập của nó một cách thống nhất toàn cục.

c.Sơ đồ UML



Singleton (LoadBalancer)

- Định nghĩa một thao tác tạo thể nghiệm cho phép đối tượng khách truy nhập đến thể nghiệm đồng nhất của nó như một thao tác của lớp.

- Chịu trách nhiêm cho việc tạo ra và duy trì thể nghiệm đồng nhất của chính nó.

d. Mẫu liên quan

Có rất nhiều mẫu có thể cài đặt bổ sung từ việc sử dụng Singleton, chẳng hạn như Abstract Factory, Builder, và Prototype.

Tổng kết về nhóm Creational Pattern

Có 2 cách thông dụng nhất để tham số hoá một hệ thống bằng các lớp của đối tượng mà nó tạo. Một cách là chia nhỏ nó thành các lớp con để tạo đối tượng tương ứng. Điều này tương đương với việc chúng ta sử dụng mẫu Factory Method. Điểm hạn chế chính của cách tiếp cận này là đòi hỏi phải tạo một lớp mới khi thay đổi lớp dẫn xuất.Giống như kiểu thác nước.Chẳng hạn một dẫn xuất tự tạo ra bằng Abstract Factory, sau đó chúng ta phải đè lên lớp khởi tạo này.

Nhóm Structural

6. Adapter.

(tần suất sử dụng :cao trung bình)

a. Vấn đề đặt ra

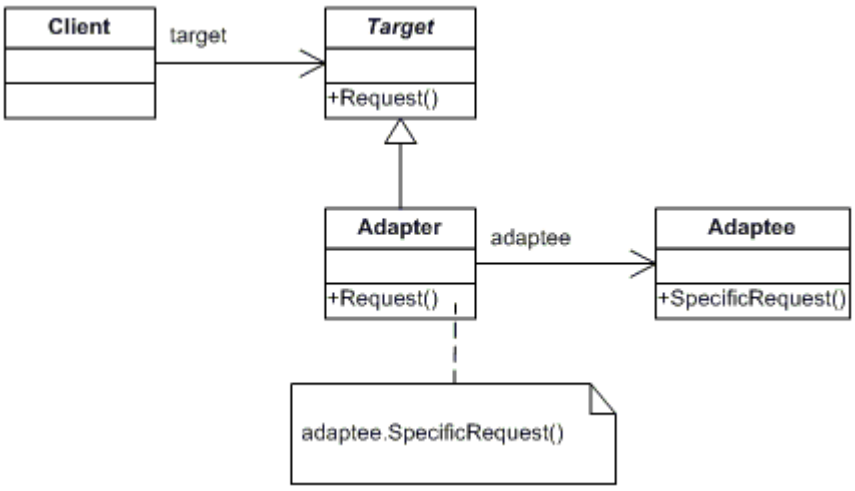
Đôi khi một lớp công cụ được thiết kế cho việc sử dụng lại, lại không thể sử dụng lại chỉ bởi giao diện không thích hợp với miền giao diện đặc biệt mà một ứng dụng yêu cầu.Adapter đưa ra một giải pháp cho vấn đề này.

Trong một trường hợp khác ta muốn sử dụng một lớp đã tồn tại và giao diện của nó không phù hợp với giao diện của một lớp mà ta yêu cầu.Ta muốn tạo ra một lớp có khả năng được dùng lại, lớp đó cho phép kết hợp với các lớp không liên quan hoặc không được dự đoán trước, các lớp đó không nhất thiết phải có giao diện tương thích với nhau.(Chỉ với đối tượng adapter) ta cần sử dụng một vài lớp con đã tồn tại, nhưng để làm cho các giao diện của chúng tương thích với nhau bằng việc phân lớp các giao diện đó là việc làm không thực tế, để làm được điều này ta dùng một đối tượng adapter để biến đổi giao diện lớp cha của nó.

b. Định nghĩa

Adapter là mẫu thiết kế dùng để biến đổi giao diện của một lớp thành một giao diện khác mà clients yêu cầu. Adapter ngăn cản các lớp làm việc cùng nhau đó không thể làm bằng cách nào khác bởi giao diện không tương thích.

c. Sơ đồ UML



Các thành phần tham gia:

- Target:định nghĩa một miền giao diện đặc biệt mà Client sử dụng.
- Client: cộng tác với các đối tượng tương thích với giao diện Target.
- Adaptee: định nghĩa một giao diện đã tồn tại mà cần phải làm biến đổi cho thích hợp.
- Adapter: làm tương thích giao diện của Adaptee với giao diện của Target.

d. Mẫu liên quan

Bridge có một cấu trúc tương tự như một đối tượng của Adapter, nhưng Bridge có một mục đích khác.Nó chia giao diện từ các phần cài đặt của nó ra riêng rẽ để từ đó có thể linh hoạt hơn và độc lập nhau. Sử dụng một Adapter đồng nghĩa với việc thay đổi giao diện của một đối tượng đã tồn tại.

Decorator nâng cấp một đối tượng khác mà không làm thay đổi giao diện của nó. Một Decorator do đó mà trong suốt với ứng dụng hơn là một Adapter. Như một hệ quả Decorator hỗ trợ cơ chế kết tập đệ quy mà điều này không thể thực hiện được đối với các Adapter thuần tuý.

Proxy định nghĩa một đại diện cho một đối tượng khác và không làm thay đổi giao diện của nó.

7. Bridge

(Tần suất sử dụng :trung bình)

a. Vấn đề đặt ra

Khi một lớp trừu tượng (abstraction) có thể có một vài thành phần bổ sung thêm thì cách thông thường phù hợp với chúng là sử dụng kế thừa. Một lớp trừu tượng định nghĩa một giao diện cho trừu tượng đó, và các lớp con cụ thể thực hiện nó theo các cách khác nhau. Nhưng cách tiếp cận này không đủ mềm dẻo. Sự kế thừa ràng buộc một thành phần bổ sung thêm là cố định cho abstraction, điều này làm nó khó thay đổi, mở rộng, và sử dụng lại các abstraction, các thành phần bổ sung một cách độc lập.Trong trường hợp này dùng một mẫu Bridge là thích hợp nhất.Mẫu Bridge thường được ứng dụng khi :

- Ta muốn tránh một ràng buộc cố định giữa một abstraction và một thành phần bổ sung thêm của nó.

- Cả hai, các abstraction và các thành phần cài đặt của chúng nên có khả năng mở rộng bằng việc phân chia lớp. Trong trường hợp này, Bridge pattern cho phép ta kết hợp các abstraction và các thành phần bổ sung thêm khác nhau và mở rộng chúng một cách độc lập.

- Thay đổi trong thành phần được bổ sung thêm của một abstraction mà không ảnh hưởng đối với các client, tức là mã của chúng không nên đem biên dịch lại.

- Ta muốn làm ẩn đi hoàn toàn các thành phần bổ sung thêm của một abstraction khỏi các client.

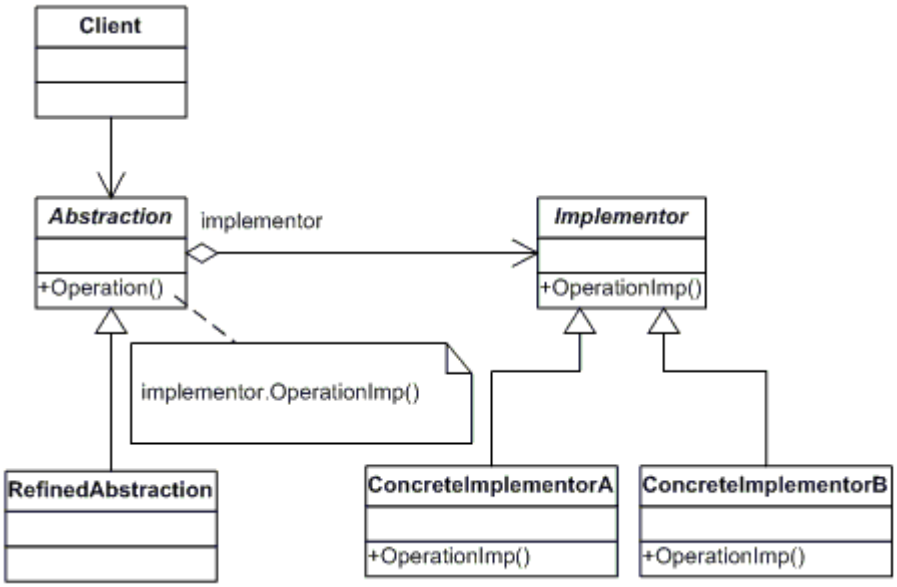
- Ta có một sự phát triển rất nhanh các lớp, hệ thống phân cấp lớp chỉ ra là cần phải tách một đối tượng thành hai phần.

- Ta muốn thành phần bổ sung thêm có mặt trong nhiều đối tượng, và việc này lại được che khỏi client (client không thấy được).

b. Định nghĩa

Bridge là mẫu thiết kế dùng để tách riêng một lớp trừu tượng khỏi thành phần cài đặt của nó để có được hai cái có thể biến đổi độc lập.

c.Sơ đồ UML



Abstraction (BusinessObject)

-Định nghĩa một giao diện trừu tượng

- Duy trì một tham chiếu tới đối tượng của các lớp kế thừa từ nó

RefinedAbstraction (CustomersBusinessObject)

- Mở rộng giao diện bằng cách định nghĩa một đối tượng trừu tượng

Implementor (DataObject)

- Định nghĩa giao diện cho lớp kế thừa.Giao diện này không phải tương ứng chính xác với giao diện trừu tượng. Trong thực tế 2 giao diện này có thể khá là độc lập. Việc kế thừa một cách tuỳ ý các giao diện cũng chỉ cung cấp duy nhất các thao tác nguyên thuỷ và lớp trừu tượng định nghĩa một thao tác mức trên dựa những thao tác nguyên thuỷ này.

ConcreteImplementor (CustomersDataObject)

- Cài đặt giao diện đã được cài đặt và định nghĩa một cài đặt cụ thể.

d. Các mẫu liên quan

Abstract Factory cũng có thể tạo ra và cấu hình một Bridge. Adapter có thể được cơ cấu theo hướng để 2 lớp không có quan hệ gì với nhau có thể làm việc với nhau được. Nó thường ứng dụng cho các hệ thống sau khi đã được thiết kế. Bridge xét ở một khía cạnh khác nó kết thúc một thiết kế để lớp trừu tượng và lớp cài đặt có thể tuỳ biến một cách độc lập.

8. Composite

(tần suất sử dụng :Cao)

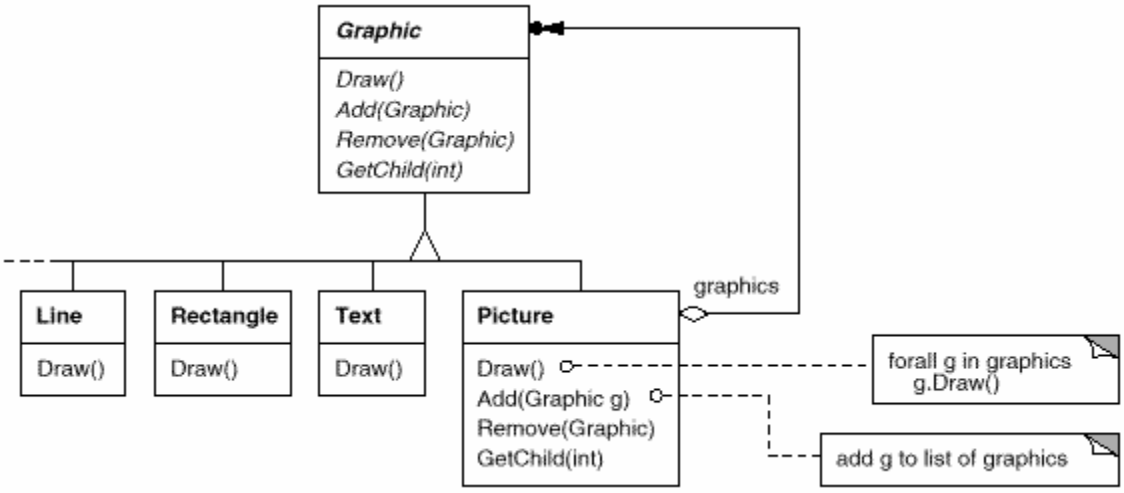
a. Vấn đề đặt ra

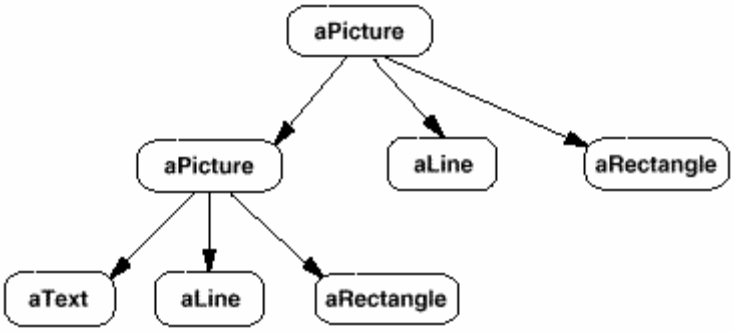
Các ứng dụng đồ họa như bộ soạn thảo hình vẽ và các hệ thống lưu giữ biểu đồ cho phép người sử dụng xây dựng lên các lược đồ phức tạp khác xa với các thành phần cơ bản, đơn giản. Người sử dụng có thể nhóm một số các thành phần để tạo thành các thành phần khác lớn hơn, và các thành phần lớn hơn này lại có thể được nhóm lại để tạo thành các thành phần lớn hơn nữa. Một cài đặt đơn giản có thể xác định các lớp cho các thành phần đồ họa cơ bản như Text và Line, cộng với các lớp khác cho phép hoạt động như các khuôn chứa các thành phần cơ bản đó.

Nhưng có một vấn đề với cách tiếp cận này, đó là, mã sử dụng các lớp đó phải tác động lên các đối tượng nguyên thủy (cơ bản) và các đối tượng bao hàm các thành phần nguyên thủy ấy là khác nhau ngay cả khi hầu hết thời gian người sử dụng tác động lên chúng là như nhau. Có sự phân biệt các đối tượng này làm cho ứng dụng trở nên phức tạp hơn. Composite pattern đề cập đến việc sử dụng các thành phần đệ quy để làm cho các client không tạo ra sự phân biệt trên.

Giải pháp của Composite pattern là một lớp trừu tượng biểu diễn cả các thành phần cơ bản và các lớp chứa chúng. Lớp này cũng xác định các thao tác truy nhập và quản lý các con của nó.

Ví dụ:





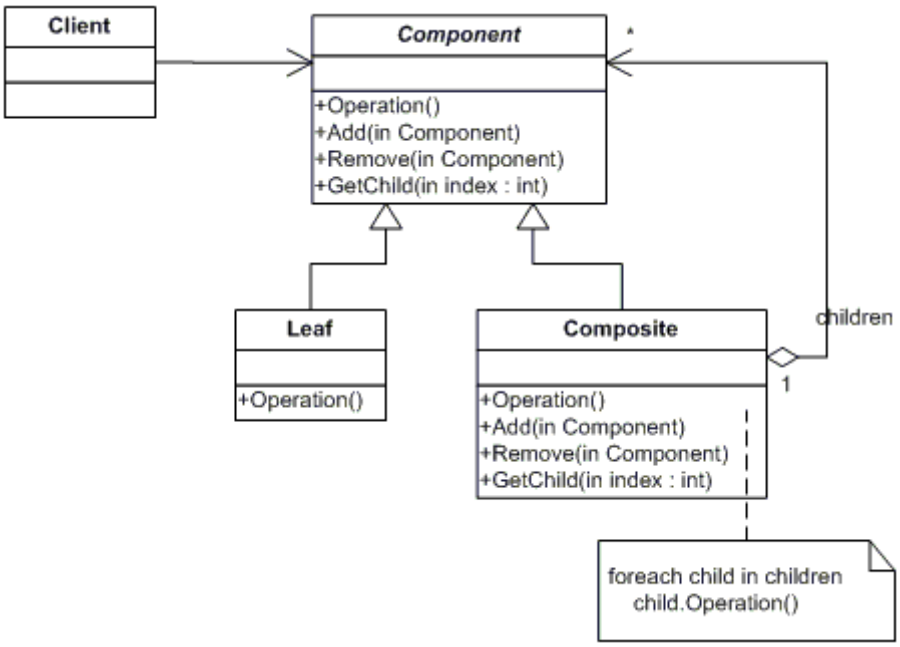
Composite được áp dụng trong các trường hợp sau :

- Ta muốn biểu diễn hệ thống phân lớp bộ phận – toàn bộ của các đối tượng
- Ta muốn các client có khả năng bỏ qua sự khác nhau giữa các thành phần của các đối tượng và các đối tượng riêng lẻ. Các client sẽ "đối xử" với các đối tượng trong cấu trúc composite một cách thống nhất.

b. Định nghĩa

Composite là mẫu thiết kế dùng để tạo ra các đối tượng trong các cấu trúc cây để biểu diễn hệ thống phân lớp: bộ phận – toàn bộ. Composite cho phép các client tác động đến từng đối tượng và các thành phần của đối tượng một cách thống nhất.

c. Biểu đồ UML



Component (DrawingElement)

- Khai báo giao diện cho các đối tượng trong một khối kết tập.
- Cài đặt các phương thức mặc định cho giao diện chung của các lớp một cách phù hợp.
- Khai báo một giao diện cho việc truy cập và quản lý các thành phần con của nó
- Định nghĩa một giao diện cho việc truy cập các đối tượng cha của các thành phần theo một cấu trúc đệ quy và cài đặt nó một cách phù hợp nhất.

Leaf (PrimitiveElement)

- Đại diện cho một đối tượng nút là trong khối kết tập. Một lá là một nút không có con.
- Định nghĩa hành vi cho các đối tượng nguyên thuỷ trong khối kết tập

Composite (CompositeElement)

- Định nghĩa hành vi cho các thành phần mà có con.
- Lưu trữ các thành phần con
- Cài đặt toán tử quan hệ giữa các con trong giao diên của thành phần

Client (CompositeApp)

- Vận dụng các đối tượng trong khối kết tập thông qua giao diện của thành phần

d. Các mẫu liên quan

Một mẫu mà thường dùng làm thành phần liên kết đến đối tượng cha là Chain of Responsibility

Mẫu Decorator cũng thường được sử dụng với Composite.Khi Decorator và Composite cùng được sử dụng cùng nhau, chúng thường sẽ có một lớp cha chung. Vì vậy Decorator sẽ hỗ trợ thành phần giao diện với các phương thức như Add, Remove và GetChild.

Mẫu Flyweight để cho chúng ta chia sẻ thành phần, nhưng chúng sẽ không tham chiếu đến cha của chúng.

Mẫu Iterator có thể dùng để duyệt mẫu Composite.

Mẫu Visitor định vị thao tác và hành vi nào sẽ được phân phối qua các lớp lá và Composite.

9. Decorator

(tần suất sử dụng :Cao trung bình)

a. Định nghĩa

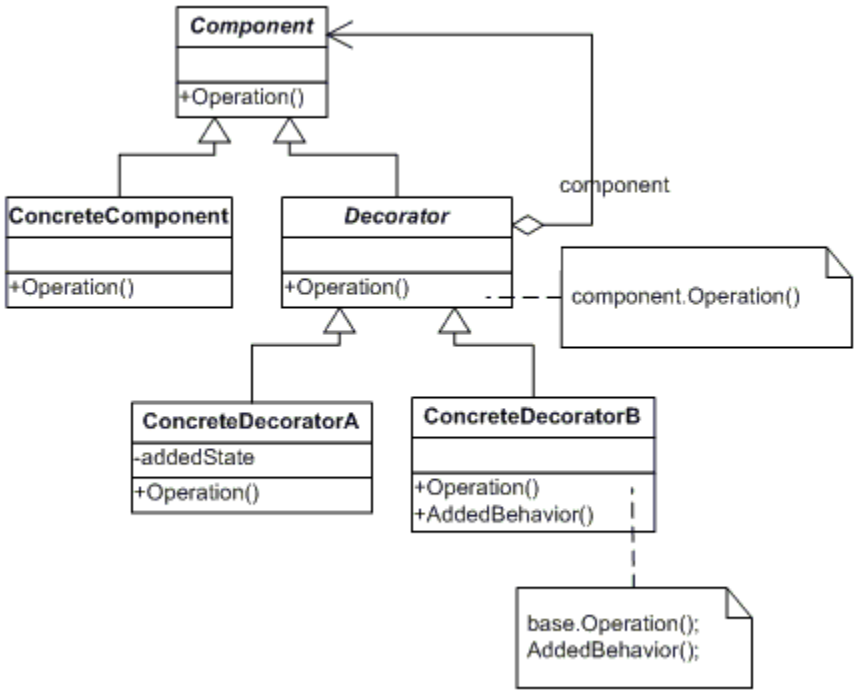
Gắn một vài chức năng bổ sung cho các đối tượng (gán động). Decorator cung cấp một số thay đổi mềm dẻo cho các phân lớp để mở rộng thêm các chức năng.

b. Ứng dụng

Sử dụng Decorator khi:

- Thêm các chức năng bổ sung cho các đối tượng riêng biệt một cách động và trong suốt, nghĩa là không chịu ảnh hưởng (tác động) của các đối tượng khác.
- Cho các chức năng mà các chức năng này có thể được rút lại (hủy bỏ) (nếu không cần nữa).
- Khi sự mở rộng được thực hiện bởi các phân lớp là không thể thực hiện được. Đôi khi một lượng lớn các mở rộng độc lập có thể thực hiện được nhưng lại tạo ra một sự bùng nổ các phân lớp để trợ giúp cho các kết hợp. Hoặc một định nghĩa lớp có thể bị che đi hay nói cách khác nó không có giá trị cho việc phân lớp.

c. Sơ đồ UML



Component (LibraryItem)

- Định nghĩa giao diện cho đối tượng mà có thể có nhiệm vụ thêm nó vào một cách động

ConcreteComponent (Book, Video)

- Định nghĩa một đối tượng để có thể gắn nhiệm vụ thêm thành phần cho nó

Decorator (Decorator)

- Duy trì một tham chiếu tới một đối tượng thành phần và định nghĩa một giao diện mà phù hợp với giao diện của thành phần.

ConcreteDecorator (Borrowable)

- Thêm nhiệm vụ cho thành phần

d. Các mẫu liên quan

Mẫu Decorator khác với Adapter, Decorator chỉ thay đổi nhiệm vụ của đối tượng, không phải là thay đổi giao diện của nó như Adapter.Adapter sẽ mang đến cho đối tượng một giao diện mới hoàn toàn.Decorator cũng có thể coi như một Composite bị thoái hoá với duy nhất một thành phần. Tuy nhiên, một Decorator thêm phần nhiệm phụ, nó là phần đối tượng được kết tập vào.Một Decorator cho phép chúng ta thay đổi bề ngoài của một đối tượng, một strategy cho phép chúng ta thay đổi ruột của đối tượng. Chúng là 2 cách luân phiên nhau để ta thay đổi một đối tượng.

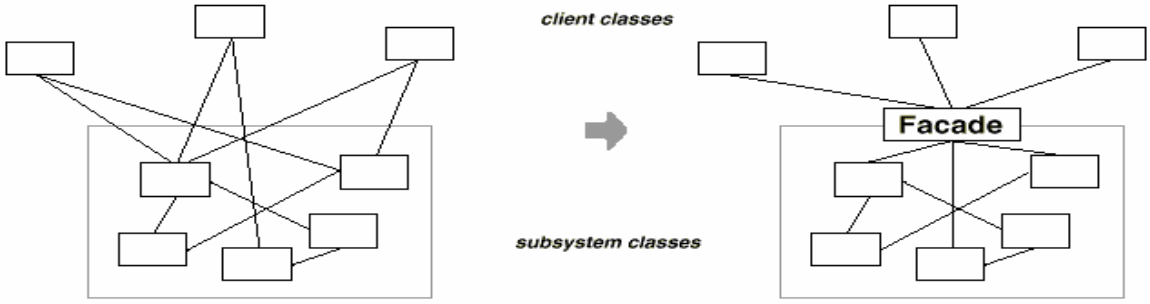
10. Facade

(Tần suất sử dụng :Cao)

a. Vấn đề đặt ra

Khi cấu trúc hóa một hệ thống thành các hệ thống con sẽ giúp làm giảm độ phức tạp của hệ thống. Một mục tiêu thiết kế thông thường là tối thiểu hóa sự giao tiếp và phụ thuộc giữa các hệ thống con. Một cách để đạt được mục tiêu này là đưa ra đối

tượng facade, đối tượng này cung cấp một giao diện đơn giản để dễ dàng tổng quát hóa cho một hệ thống con hơn.

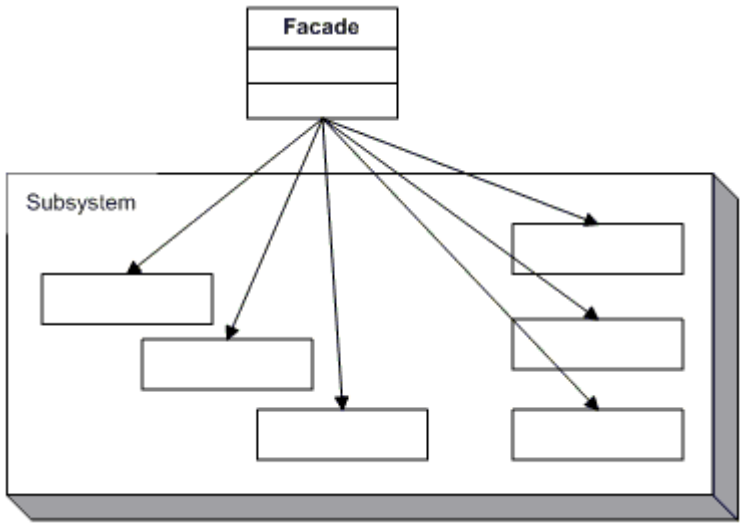


Chúng ta sẽ dùng mẫu Facade để giải quyết vấn đề này

b. Định nghĩa

Mẫu Facade cung cấp một giao diện thống nhất cho một tập các giao diện trong một hệ thống con. Facade định nghĩa một giao diện ở mức cao hơn, giao diện này làm cho hệ thống con được sử dụng dễ dàng hơn.

c. Sơ đồ UML



Facade (MortgageApplication)

- Có thể xem như các lớp hệ thống con mà chịu trách nhiệm đối với các yêu cầu đến nó.
- Trình khác uỷ nhiệm yêu cầu tới một đối tượng của hệ thống con

Subsystem classes (Bank, Credit, Loan)

- Cài đặt chức năng của hệ thống con
- Giữ handle làm việc bằng đối tượng Facade
- Không có một kiến thức gì về Facade và giữ tham chiếu đến nó.

d. Mẫu liên quan

Abstract Factory có thể được sử dụng cùng với Facade để cung cấp một giao diện cho việc tạo hệ thống con một cách độc lập hệ thống con. Abstract Factory cũng có thể được sử dụng như một sự thay thế cho Facade để ẩn các lớp nền đặc biệt.

Mediator tương tự như Facade ở chổ trừu tượng chức năng của một lớp đã tồn tại.Tuy nhiên mục đích của Mediator là trừu tượng một cách chuyên quyền sự giao tiếp giữa đối tượng cộng tác, thường chức năng trung tâm không thuộc về bất kỳ đối tượng cộng tác nào.Một đối tượng cộng tác với Mediator được nhận và giao tiếp với mediator thay vì giao tiếp với nhau một cách trực tiếp. Trong hoàn cảnh nào đó, Facade chỉ đơn thuần là trừu tượng giao diện cho một một đối tượng hệ thống con để làm nó dễ sử dụng hơn, nó không định nghĩa một chức năng mới và lớp hệ thống con không hề biết gì về nó.

Thường thì một đối tượng Facade là đủ. Do đó đối tượng Facade thường là Singleton.

11. Flyweight

(Tần suất sử dụng :thấp trung bình)

a. Vấn đề đặt ra

Một vài ứng dụng có thể được lợi từ việc sử dụng các đối tượng xuyên suốt thiết kế của chúng, nhưng một cài đặt không tốt sẽ cản trở điều đó.Trong tình huống này chúng ta sẽ dùng mẫu thiết kế Flyweight để giải quyết.

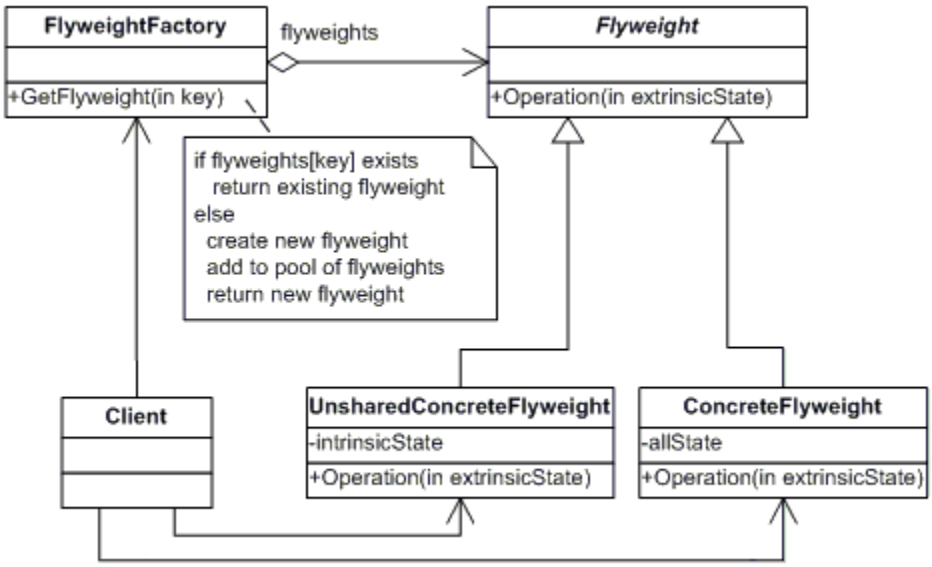
b. Định nghĩa

Mẫu thiết kế Flyweight là mẫu thiết kế được sử dụng việc chia xẻ để trợ giúp một lượng lớn các đối tượng một cách hiệu quả.

Việc sử dụng mẫu này cần chú ý rằng :các hiệu ứng của Flyweight pattern đòi hỏi rất nhiều vào việc nó được sử dụng ở đâu và sử dụng nó như thế nào. Sử dụng Flyweight pattern khi tất cả cá điều sau đây là đúng:

- Một ứng dụng sử dụng một lượng lớn các đối tượng.
- Giá thành lưu trữ rất cao bởi số lượng các đối tượng là rất lớn.
- Hầu hết trạng thái của các đối tượng có thể chịu tác động từ bên ngoài.
- Ứng dụng không yêu cầu đối tượng đồng nhất. Khi các đối tượng flyweight có thể bị phân tách, việc kiểm tra tính đồng nhất sẽ trả về đúng cho các đối tượng được định nghĩa dựa trên các khái niệm khác nhau.

c. Sơ đồ UML



Flyweight (Character)

- Khai báo một giao diện qua Flyweight mà có thể nhận và hành động nằm ngoài trạng thái

ConcreteFlyweight (CharacterA, CharacterB, ..., CharacterZ)

- Cài đặt giao diện Flyweight và thêm phần lưu trữ cho các trạng thái ngoài. -- Một đối tượng Concrete Flyweight phải được chia sẽ. Bất cứ một trạng thái nào nó lưu trữ đều phải là ở bên ngoài, phải độc lập với ngữ cảnh của đối tượng ConcreteFlyweight

UnsharedConcreteFlyweight (not used)

- Không phải tất cả các lớp con đều cần phải chia sẽ. Giao diên Flyweight cho phép chia sẽ, nhưng điều đó là không bắt buộc. Nó là thông dụng cho đối tượng UnsharedConcreteFlyweight để có đối tượng ConcreteFlyweight như đối tượng con ở các mức trong cấu trúc đối tượng Flyweight.

FlyweightFactory (CharacterFactory)

- Tạo và quản lý đối tượng flyweight
- Đảm bảo rằng flyweight được chia sẽ một cách đúng đắn. Khi đối tượng khách yêu cầu một đối tượng flyweight cung cấp một thể nghiệm đã tồn tài hoặc tạo một cái khác, nếu nó chưa có.

Client (FlyweightApp)

- Duy trì một tham chiếu đến Flyweight
- Tính toán hoặc lưu trữ trạng thái ngoài của Flyweight

d.Mẫu liên quan

Mẫu Flyweight thường kết hợp với mẫu Composite để cài đặt cấu trúc phân cấp lôgic trong giới hạn của một sơ đồ vòng xoắn trực tiếp với các lá chia sẻ của nó.

Thường tốt nhất là cài đặt các mẫu State và Strategy giống như là flyweight.

12. Proxy

(Tần suất sử dụng :cao)

a. Vấn đề đặt ra

Lý do để điều khiển truy nhập tới một đối tượng là làm theo toàn bộ quá trình tạo ra và khởi đầu của nó cho tới tận khi thực sự chúng ta cần sử dụng nó.Trong trường hợp này ta nên dùng mẫu thiết kế proxy. Proxy có thể được ứng dụng tại bất cứ nơi nào mà ở đó cần có một sự tham chiếu tới một đối tượng linh hoạt hơn, tinh xảo hơn so với việc sử dụng một con trỏ đơn giản.

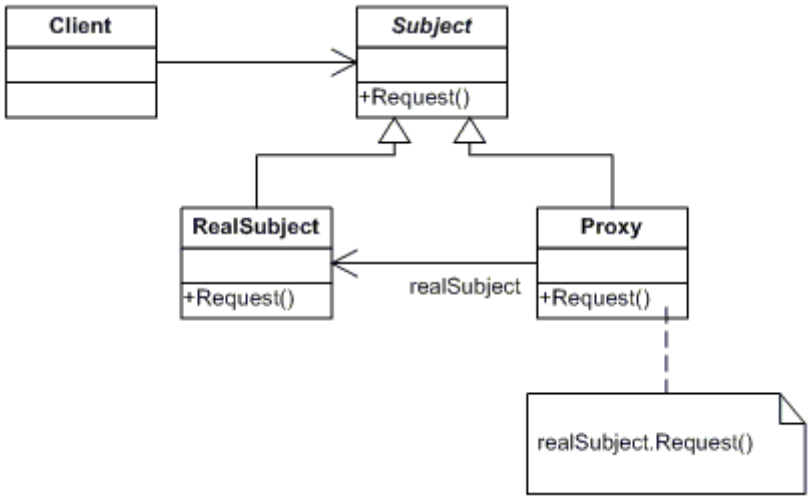
Sau đây là một vài trường hợp thông thường trong đó proxy được vận dụng:

- Một remote proxy cung cấp một biểu diễn (một mẫu) cục bộ cho một đối tượng trong một không gian địa chỉ khác.
- Một virtual proxy tạo ra một đối tượng có chi phí cao theo yêu cầu.
- Một protection proxy điều khiển việc truy nhập đối tượng gốc. Các protection proxy rất hữu ích khi các đối tượng có các quyền truy nhập khác nhau.
- Một smart reference là sự thay thế cho một con trỏ rỗng cho phép thực hiện các chức năng thêm vào khi một đối tượng được truy nhập. Các trường hợp sử dụng điển hình:
  + Đếm số tham chiếu tới đối tượng thực, do đó nó có thể tự động giải phóng khi có không nhiều các tham chiếu.
  + Tải một đối tượng liên tục vào bộ nhớ khi nó được tham chiếu lần đầu tiên.
  + Kiểm tra đối tượng thực đó có được khóa hay không trước khi nó bị truy nhập để đảm bảo không đối tượng nào khác có thể thay đổi nó.

b. Định nghĩa

Cung cấp một đại diện cho một đối tượng khác để điều khiển việc truy nhập nó.

c. Sơ đồ UML



Proxy (MathProxy)

- Duy trì một tham chiếu để cho proxy truy cập vào một đối tượng thực. Proxy có thể tham khả đến một chủ thể nếu như giao diện của RealSubject và Subject là như nhau.
- Cung cấp một giao diện xác định Subject để một proxy có thể thay thế cho đối tượng thực.

- Điều khiển truy cập tới đối tượng thực và có thể chịu trách nhiệm tạo ra và xoá nó đi.
- Trong các nhiệm vụ khác phụ thuộc vào từng loại proxy
    - *Remote proxies* chịu trách nhiệm cho việc mã hoá một yêu cầu và các tham số của nó và để truyền yêu cầu mã hoá cho chủ thể trong không gian địa chỉ khác.
    - *Virtual proxies* có thể thêm phần thông tin đệm về chủ thể thực để chúng có thể trì hoãn truy nhập nó.
    - *Protection proxies* kiểm tra đối tượng gọi đã có yêu cầu quyền truy nhập để thực hiện một yêu cầu.

Subject (IMath)

- Định nghĩa một giao diện chung cho chủ thể thực và Proxy để proxy có thể sử dụng ở mọi nơi mà một RealSubject mong đợi.

RealSubject (Math)

- Định nghĩa đối tượng thực mà proxy đại diện.

d. Mẫu liên quan

Một Adapter cung cấp một giao diện khác với đối tượng mà nó thích nghi. Trong trường hợp này, một Proxy cung cấp cùng một giao diện như vậy giống như một chủ thể.

Mặc dù decorator có thể có cài đặt tương tự như các proxy, decorator có một mục đích khác. Một decorator phụ thêm nhiều nhiệm vụ cho một đối tượng nhưng ngược lại một proxy điều khiển truy cập đến một đối tượng.

Proxy tuỳ biến theo nhiều cấp khác nhau mà chúng có thể sẽ được cài đặt giống như một decorator. Một proxy bảo vệ có thể được cài đặt chính xác như một decorator. Mặt khác một proxy truy cập đối tượng từ xa sẽ không chứa một tham chiếu trực tiếp đến chủ thể thực sự nhưng chỉ duy nhất có một tham chiếu trực tiếp, giống như ID của host và địa chỉ trên host vậy. Một proxy ảo sẽ bắt đầu với một tham chiếu gián tiếp chẳng hạn như tên file nhưng rốt cuộc rồi cũng sẽ đạt được một tham chiếu trực tiếp.

Các mẫu Behavioral

Các mẫu hành vi là các mẫu tập trung vào vấn đề giải quyết các thuật toán và sự phân chia trách nhiệm giữa các đối tượng. Mẫu hành vi không chỉ miêu tả mô hình của các đối tượng mà còn miêu tả mô hình trao đổi thông tin giữa chúng; đặc trưng hoá các luồng điều khiển phức tạp, giúp chúng ta tập trung hơn vào việc xây dựng cách thức liên kết giữa các đối tượng thay vì các luồng điều khiển.

Mẫu hành vi kiểu lớp sử dụng tính kế thừa để phân phối hành vi giữa các lớp. Dưới đây chúng ta sẽ nghiên cứu hai mẫu thuộc loại đó : Temple Method và Interpreter. Trong hai mẫu đó, Temple Method là mẫu đơn giản và thông dụng hơn. Nó định nghĩa trừu tượng từng bước của một thuật toán; mỗi bước sử dụng một hàm trừu tượng hoặc một hàm nguyên thuỷ. Các lớp con của nó làm sáng tỏ thuật toán cụ thể bằng cách cụ

thể hoá các hàm trừu tượng. Mẫu Interpreter diễn tả văn phạm như một hệ thống phân cấp của các lớp và trình phiên dịch như một thủ tục tác động lên các thể hiện của các lớp đó.

Mẫu hành vi kiểu đối tượng lại sử dụng đối tượng thành phần thay vì thừa kế. Một vài mẫu miêu tả cách thức một nhóm các đối tượng ngang hàng hợp tác với nhau để thi hành các tác vụ mà không một đối tượng riêng lẻ nào có thể tự thi hành. Một vấn đề quan trọng được đặt ra ở đây là bằng cách nào các đối tượng ngang hàng đó biết được sự tồn tại của nhau. Cách đơn giản nhất và cũng "dư thừa" nhất là lưu trữ các tham chiếu trực tiếp đến nhau trong các đối tượng ngang hàng. Mẫu Mediator tránh sự thừa thãi này bằng cách xây dựng kết nối trung gian, liên kết gián tiếp các đối tượng ngang hàng.

Mẫu Chain of Responsibility xây dựng mô hình liên kết thậm chí còn "lỏng" hơn. Nó cho phép gửi yêu cầu đến một đối tượng thông qua chuỗi các đối tượng "ứng cử". Mỗi ứng cử viên có khả năng thực hiện yêu cầu tuỳ thuộc vào các điều kiện trong thời gian chạy. Số lượng ứng cử viên là một con số mở và ta có thể lựa chọn những ứng cử viên nào sẽ tham gia vào chuỗi trong thời gian chạy.

Mẫu Observer xây dựng và vận hành một sự phụ thuộc giữa các đối tượng. Một ví dụ cụ thể của mẫu này là mô hình Model/View/Controller của Smalltalk trong đó tất cả các views của model đều được thông báo khi trạng thái của model có sự thay đổi.

Một số mẫu hành vi khác lại quan tâm đến việc đóng gói các hành vi vào một đối tượng và "uỷ thác" các yêu cầu cho nó. Mẫu Strategy đóng gói một thuật toán trong một đối tượng, xây dựng cơ chế khiến cho việc xác định và thay đổi thuật toán mà đối tượng sử dụng trở nên đơn giản. Trong khi đó mẫu Command lại đóng gói một yêu cầu vào một đối tượng; làm cho nó có thể được truyền như một tham số, được lưu trữ trong một history list hoặc thao tác theo những cách thức khác. Mẫu State đóng gói các trạng thái của một đối tượng, làm cho đối tượng có khả năng thay đổi hành vi của mình khi trạng thái thay đổi. Mẫu Visitor đóng gói các hành vi vốn được phân phối trong nhiều lớp và mẫu Iterator trừu tượng hoá cách thức truy cập và duyệt các đối tượng trong một tập hợp.

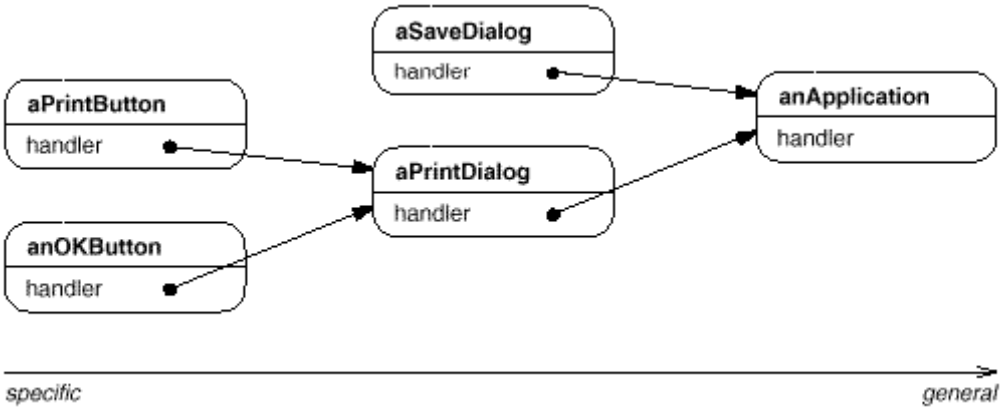
13. Mẫu Chain of Responsibility

a.Vấn đề đặt ra

Xét bài toán xây dựng hệ thống trợ giúp phụ thuộc ngữ cảnh cho một giao diện đồ hoạ; trong đó người sử dụng có thể lấy thông tin trợ giúp về bất cứ thành phần nào của giao diện bằng cách ấn vào nó. Thông tin trợ giúp cung cấp phụ thuộc vào thành phần giao diện được chọn và ngữ cảnh của nó. Lấy ví dụ một nút trong một hộp thoại sẽ có thông tin trợ giúp khác với một nút tương tự trong cửa sổ chính. Ngoài ra nếu không có thông tin trợ giúp nào được cung cấp cho thành phần được chọn của giao diện, hệ thống trợ giúp sẽ hiển thị thông tin trợ giúp tổng quát hơn về ngữ cảnh, lấy ví dụ thông tin về hộp thoại.

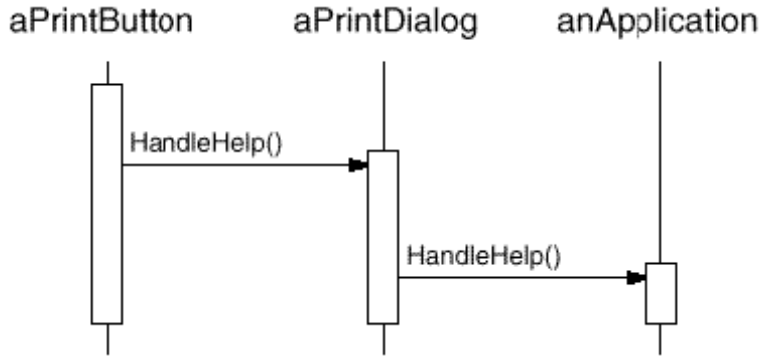
Bài toán trên dẫn đến một hành động rất tự nhiên đó là tổ chức các thông tin trợ giúp theo mức độ tổng quát của chúng, từ cụ thể nhất đến tổng quát nhất. Ngoài ra chúng ta cũng dễ dàng nhận thấy rằng yêu cầu trợ giúp sẽ được xử lý bởi một trong số các đối tượng giao diện người dùng phụ thuộc vào ngữ cảnh sử dụng và tính chi tiết của thông tin trợ giúp đươc cung cấp.

Vấn đề đặt ra ở đây là đối tượng khởi xướng yêu cầu không hề biết yêu cầu đó sẽ được xử lý bởi đối tượng cụ thể nào. Vì thế chúng ta cần có cơ chế tách rời chúng, và mẫu Chain of Responsibility xây dựng mô hình để thực hiện điều này :



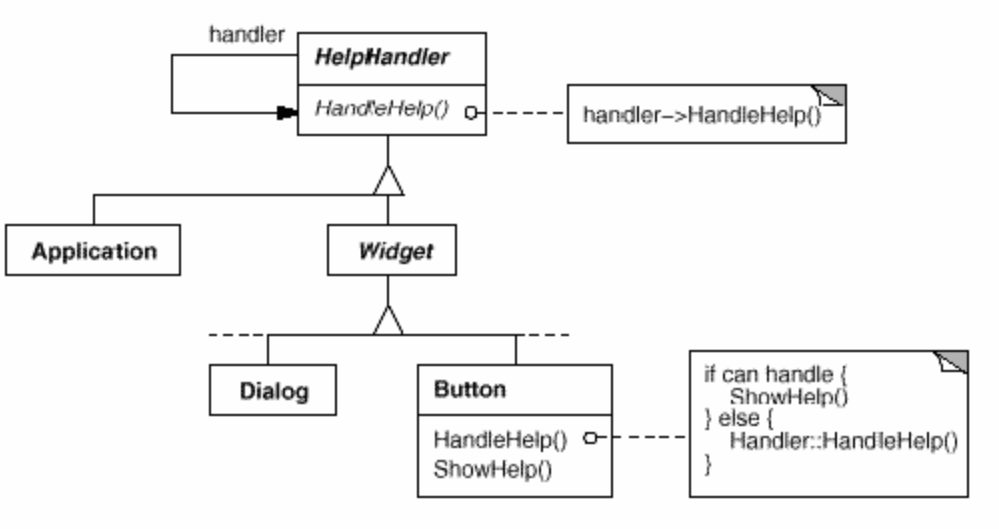
Theo mô hình này, đối tượng đầu tiên trong dây chuyền sẽ nhận được yêu cầu có thể xử lý yêu cầu đó hoặc chuyển nó cho đối tượng kế tiếp; điều tương tự cũng xảy ra với đối tượng này. Bằng cách này, đối tượng khởi xướng yêu cầu không cần biết yêu cầu sẽ được xử lý bởi đối tượng nào, nó chỉ biết rằng yêu cầu đó sẽ đươc xử lý một cách "hợp lý" nhất.

Giả sử người sử dụng yêu cầu trợ giúp cho một nút có tiêu đề "Print". Nút đó được đặt trong hộp thoại PrintDialog. Đến lượt mình, hộp thoại đó lại có khả năng xác định lớp Application chứa nó. Sơ đồ tương tác sau sẽ cho thấy cách thức yêu cầu trợ giúp đó đươc truyền đi dọc theo dây chuyền :



Trong trường hợp này, cả hai đối tượng aPrintButton và aPrintDialog đều không xử lý yêu cầu trợ giúp; vì thế yêu cầu này đươc chuyển tới cho aApplication để xử lý hoặc bỏ qua không làm gì.

Để có khả năng chuyền yêu cầu dọc theo dây chuyền và để đảm bảo tính "ẩn" của các đối tượng có thể nhận yêu cầu (với đối tượng khởi xướng yêu cầu), mỗi đối tượng trên dây chuyền đều có chung một giao diện trong việc xử lý yêu cầu và chuyển yêu cầu cho đối tượng kế tiếp. Lấy ví dụ, hệ thống trợ giúp có thể định nghĩa lớp HelpHandler với phương thức HandleHelp tương ứng. Lớp HelpHandler có thể đươc lấy làm lớp cha của các lớp mà ta muốn cung cấp khả năng xử lý yêu cầu trợ giúp :

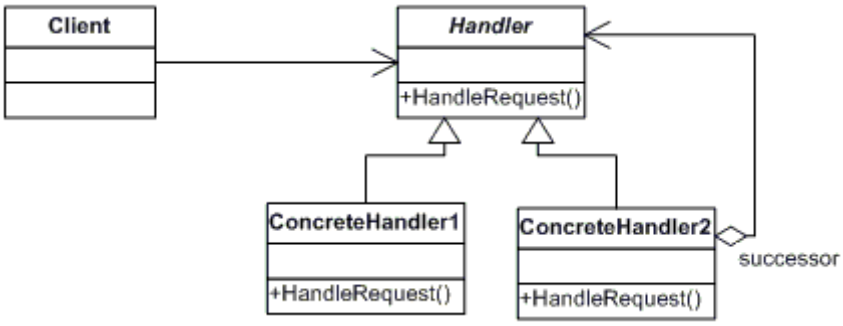


Các lớp Button, Dialog và Application sử dụng các phương thức HandleHelp để xử lý yêu cầu trợ giúp trong khi phương thức này mặc định chỉ chuyển tiếp yêu cầu trợ giúp cho nút kế tiếp. Khi đó các lớp con có thể định nghĩa lại phương thức này để cung cấp thông tin trợ giúp cụ thể hoặc chỉ sự dụng phương thức mặc định để chuyển tiếp yêu cầu theo dây chuyền.

b. Định nghĩa mẫu

Mẫu Chain of Responsiblity dùng để tránh sự liên kết trực tiếp giữa đối tượng gửi yêu cầu và đối tượng nhận yêu cầu khi yêu cầu có thể đươc xử lý bởi hơn một đối tượng. Móc nối các đối tượng nhận yêu cầu thành một chuỗi và gửi yêu cầu theo chuỗi đó cho đến khi có một đối tượng xử lý nó.

c. Sơ đồ UML



Handler (Approver)

- Định nghĩa một giao diện cho việc nắm giữ các yêu cầu.
- Cài đặt liên kết tới các Successor

ConcreteHandler (Director, VicePresident, President)

- Nắm giữ các yêu cầu mà nó đáp ứng
- Có thể truy nhập nó
- Nếu ConcreteHandle có thể nắm giữ các yêu cầu, nó sẽ làm như vậy, bằng không nó se gửi các yêu cầu tới các succcesor

Client (ChainApp)

- Khởi tạo yêu cầu tới đối tượng ConcreteHandler trên chuối đáp ứng

d. Khả năng ứng dụng của mẫu

Mẫu Chain of Responsibility có các khả năng và hạn chế sau :

- Giảm kết nối. Thay vì một đối tượng có khả năng xử lý yêu cầu chứa tham chiếu đến tất cả các đối tượng khác, nó chỉ cần một tham chiếu đến đối tượng tiếp theo.
- Tăng tính linh hoạt và phân chia trách nhiệm cho các đối tượng. Có khả năng thay đổi dây chuyền trong thời gian chạy.
- Không đảm bảo có đối tượng xử lý yêu cầu.

Chúng ta sử dụng mẫu Chain of Responsibility trong các trường hợp sau :

- Ccó lớn hơn một đối tượng có khả thực xử lý một yêu cầu trong khi đối tượng cụ thể nào xử lý yêu cầu đó lại phụ thuộc vào ngữ cảnh sử dụng.
- Muốn gửi yêu cầu đến một trong số vài đối tượng nhưng không xác định đối tượng cụ thể nào sẽ xử lý yêu cầu đó.
- Tập các đối tượng có khả năng xử lý yêu cầu là một tập biến đổi.

e. Các mẫu liên quan

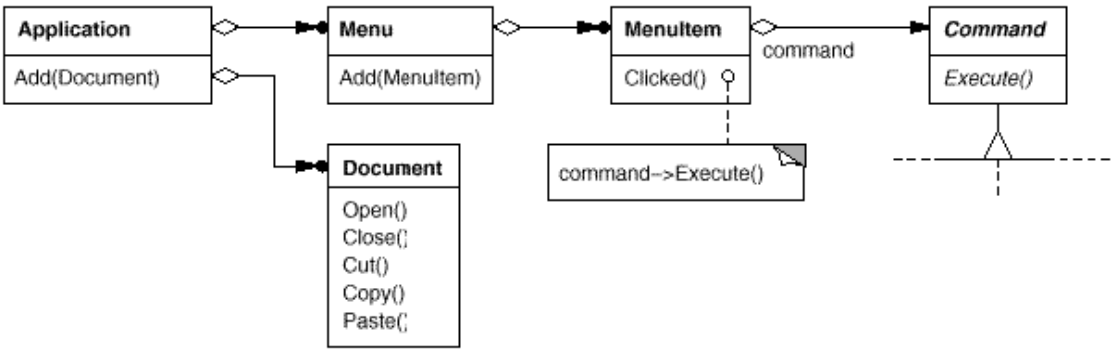
Chain of Responsibility thường được kết hợp trong mối quan hệ với composite.Có một thành phần cha có thể hành động như là successor của nó.

14.Command

a.Vấn đề đặt ra

Đôi khi chúng ta gặp tình huống trong đó cần phải gửi yêu cầu đến một đối tượng mà không biết cụ thể về đối tượng nhận yêu cầu cũng như phương thức xử lý yêu cầu. Lấy ví dụ về bộ công cụ giao diện người sử dụng cho phép xây dựng các menu. Rõ ràng, nó không thể xử lý trực tiếp yêu cầu trên các đối tượng menu vì cách thức xử lý phụ thuộc vào từng ứng dụng cụ thể.

Mẫu Command cho phép bộ công cụ như trên gửi yêu cầu đến các đối tượng chưa xác định bằng cách biến chính yêu cầu thành một đối tượng. Khái niệm quan trọng nhất của mẫu này là lớp trừu tượng Command có chức năng định nghĩa giao diện cho việc thi hành các yêu cầu. Trong trường hợp đơn giản nhất, ta chỉ cần định nghĩa một thủ tục ảo Execute trên giao diện đó. Các lớp con cụ thể của Command sẽ xác định cặp đối tượng nhận yêu cầu-các thao tác bằng cách lưu trữ một tham chiếu đến đối tượng nhận yêu cầu và định nghĩa lại thủ tục Execute để gọi các thủ tục xử lý.

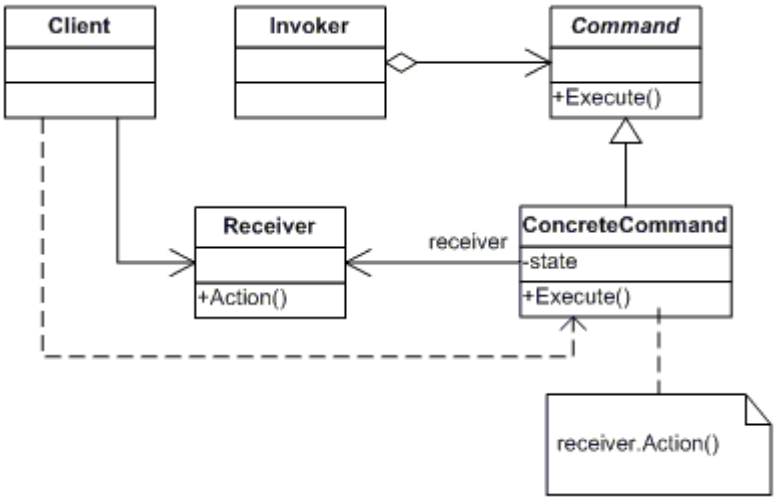


Theo cách này, chương trình sẽ có nhiệm vụ tạo ra các menu, menuitem và gán mỗi menuitem với một đối tượng thuộc lớp con cụ thể của Command. Khi người sử dụng chọn một menuitem, nó sẽ gọi hàm command->Execute() mà không cần con trỏ command trỏ đến loại lớp con cụ thể nào của lớp Command. Hàm Execute() sẽ có nhiệm vụ xử lý yêu cầu.

b.Định nghĩa mẫu

Mẫu Command đóng gói yêu cầu như là một đối tượng, làm cho nó có thể được truyền như một tham số, được lưu trữ trong một history list hoặc thao tác theo những cách thức khác nhau.

c.Sơ đồ UML



Command (Command)

- Khai báo một giao diện cho việc thực thi một thao tác

ConcreteCommand (CalculatorCommand)

- Định nghĩa một liên kết giữa một đối tượng Receiver và một hành động
- Cài đặt Execute bằng cách gọi một thao tác tương ứng trên Receiver

Client (CommandApp)

- Tạo ra một đối tượng ConcreteCommand và thiết lập đối tượng nhận của nó

Invoker (User)

- Đề nghị command để thực hiệu yêu cầu

Receiver (Calculator)

- Biết cách để thực hiện các thao tác kết hợp việc thực hiện các yêu cầu

d. Ứng dụng :

Dùng mẫu Command khi :

- Tham số hoá các đối tượng theo thủ tục mà chúng thi hành, như đối tượng MenuItem ở trên.
- Xác định, xếp hàng và thi hành các yêu cầu tại những thời điểm khác nhau.
- Cho phép quay ngược. Thủ tục Execute của lớp Command có thể lưu lại trạng thái để cho phép quay ngược các biến đổi mà nó gây ra. Khi đó lớp Command trừu tượng cần định nghĩa thêm hàm Unexecute để đảo ngược các biến đổi. Các commands đã được thi hành sẽ được lưu trong một danh sách, từ đó cho phép undo và redo không giới hạn mức.
- Cần hỗ trợ ghi lại các commands đã đươc thi hành để thi hành lại trong trường hợp hệ thống gặp sự cố.
- Thiết kế một hệ thống với các thủ tục bậc cao đươc xây dựng dựa trên các thủ tục nguyên thuỷ. Cấu trúc này thường gặp trong các hệ thống thông tin hỗ trợ "phiên giao dịch". Một phiên giao dịch là một tập hợp các sự thay đổi lên dữ liệu. Mẫu Command cung cấp cách thức mô tả phiên giao dịch. Nó có giao diện chung cho phép khởi xướng các phiên làm việc với cùng một cách thức và cũng cho phép dễ dàng mở rộng hệ thống với các phiên giao dịch mới.

e. Các mẫu liên quan

Một Composite có thể được sử dụng để cài đặt các MacroCommands.

Một Memmento có thể lưu lại các trạng thái để Command yêu cầu phục hồi lại các hiệu ứng của nó.

Một command phải được sao lưu trước khi nó được thay thế bằng các hành động trước đó như là một Prototype.

15. Interperter

a.Vấn đề đặt ra

Nếu một dạng bài toán có tần suất xuất hiện tương đối lớn, người ta thường biểu diễn các thể hiện cụ thể của nó bằng các câu trong một ngôn ngữ đơn giản. Sau đó ta có thể xây dựng một trình biên dịch để giải quyết bài toán bằng cách biên dịch các câu.

Lấy ví dụ, tìm kiếm các xâu thoả mãn một mẫu cho trước là một bài toán thường gặp và các "biểu diễn thông thường" tạo thành ngôn ngữ dùng để diễn tả các mẫu của xâu. Thay vì xây dựng từng thuật toán riêng biệt để tương ứng mỗi mẫu với một tập các xâu, người ta xây dựng thuật toán tổng quát có thể phiên dịch các "biểu diễn thông thường" thành tập các xâu tương ứng.

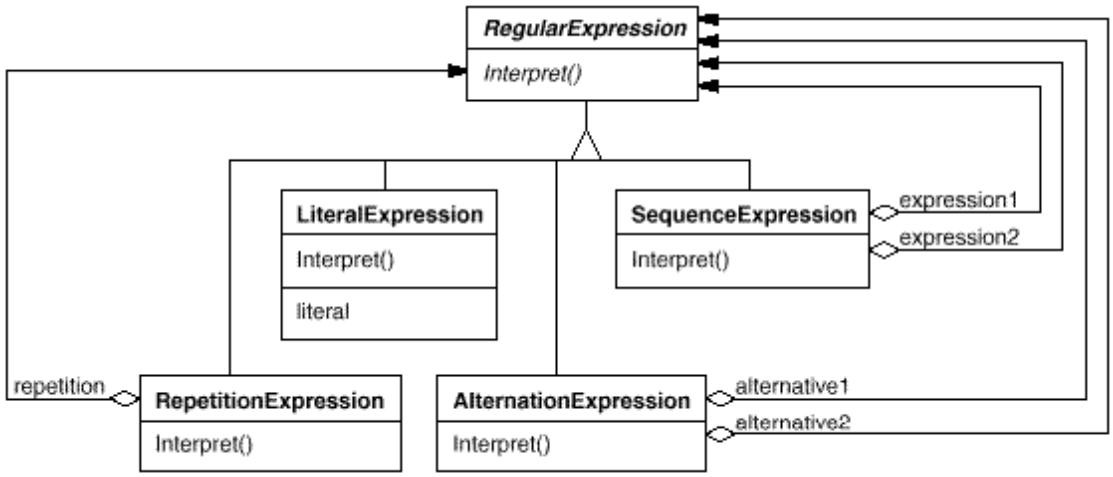
Mẫu Interpreter miêu tả cách thức xây dựng cấu trúc ngữ pháp cho những ngôn ngữ đơn giản, cách thức biểu diễn câu trong ngôn ngữ và cách thúc phiên dịch các câu đó. Trong ví dụ cụ thể này, nó miêu tả cách thức xây dựng cấu trúc ngữ pháp cho các biểu diễn thông thường, cách thức xây dựng biểu diễn thông thường và cách thúc phiên dịch các biểu diễn thông thường đó.

Giả sử cấy trúc ngữ pháp sau xác định các biểu diễn thông thường :

```
expression ::= literal | alternation | sequence |
repetition |
'(' expression ')'
alternation ::= expression '|' expression
```

```
sequence ::= expression '&' expression
repetition ::= expression '*'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' |
... }*
```

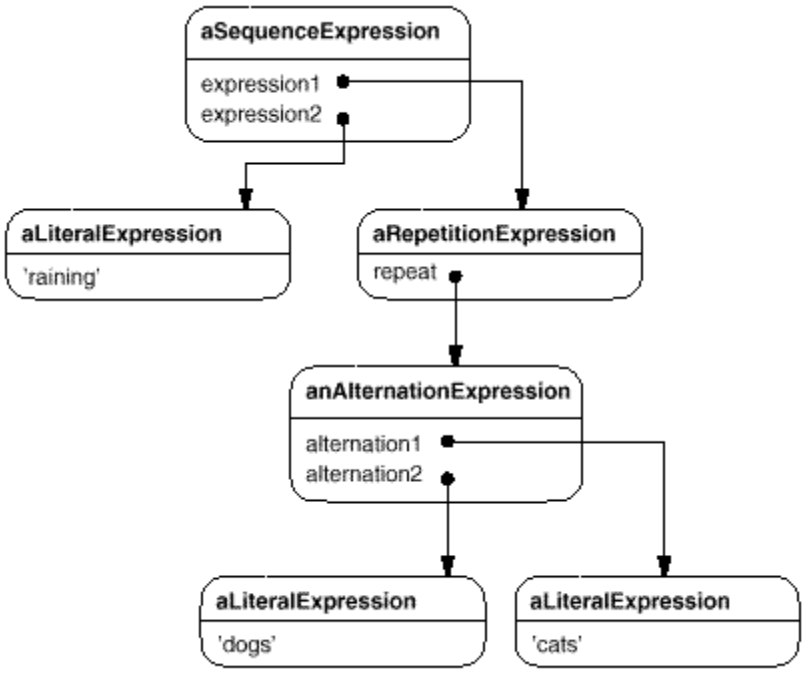
Mẫu Interpreter sử dụng các lớp để diễn tả các quy luật ngữ pháp. Khi đó cấu trúc ngữ pháp trên được diễn tả bởi năm lớp : lớp trừu tượng RegularExpression và bốn lớp con của nó : LiteralExpression, AlternationExpression, SequenceExpression và RepetitionExpression trong đó ba lớp cuối có các biến để chứa các biểu thức con.



Mỗi biểu diễn thông thường xác định bởi cấu trúc ngữ pháp trên đều được miêu tả bởi một cây cú pháp có thành phần là các đối tượng của các lớp trên. Lấy ví dụ biểu diễn

```
raining & (dogs | cats) *
```

được miêu tả bởi cây sau :



Chúng ta có thể tạo ra trình biên dịch cho các biểu diễn trên bằng cách định nghĩa thủ tục Interpret trên từng lớp con của RegularExpression. Hàm này nhận tham số

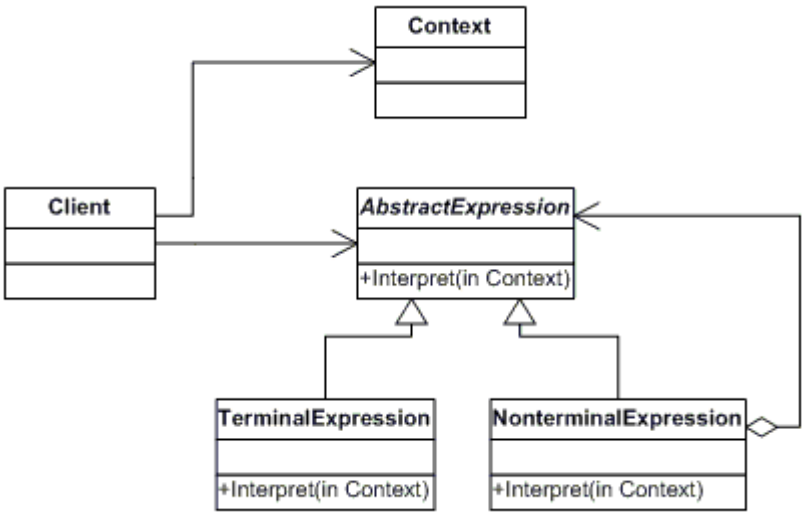
đầu vào là ngữ cảnh phiên dịch biểu diễn bao gồm xâu đầu vào và thông tin về lượng xâu đã được ghép. Nó sẽ tiếp tục ghép phần tiếp theo của xâu dựa trên ngữ cảnh đó :
VD :

- LiteralExpression sẽ kiểm tra xem đầu vào có trùng với bảng chữ cái nó định nghĩa không.
- AlternationExpression sẽ kiểm tra xem thông tin đầu vào có phải là một trong các lựa chọn của nó không.
- ...

b.Định nghĩa

Interpreter đưa ra một ngôn ngữ, xây dựng cách diễn đạt ngôn ngữ đó cùng với một trình phiên dịch sử dụng cách diễn tả trên để phiên dịch các câu.

c.Sơ đồ UML



AbstractExpression (Expression)

- Khai báo một giao diện cho việc thực hiện một thao tác

TerminalExpression (ThousandExpression, HundredExpression, TenExpression, OneExpression)

- Cài đặt một thao tác thông dịch liên kết với những ký pháp đầu cuối
- Một thể nghiệm được yêu cầu cho mọi ký pháp đầu cuối trong câu

NonterminalExpression (not used)

- Một lớp như vậy được yêu cầu cho các luật R::=R1R2..Rn trong dãy cú pháp
- Duy trì một biến thể nghiệm của kiểu AbstractExpression cho mỗi một ký pháp R1 đến Rn
- Cài đặt một phương thức thông dịch cho các ký pháp không phải đầu cuối.
- Thông dịch tự gọi đệ quy cho các biến đại diện từ R1 đến Rn

Context (Context)

- Chứa thông tin toàn cục cho việc thông dịch

Client (InterpreterApp)

Tạo (hay mang lại) một cây cú pháp trừu tượng đại diện cho một câu đặc thù trong ngữ pháp đã được định nghĩa.Cây cú pháp trừu tượng này xuất phát từ thể nghiệm của các lớp NonterminalExpression và TerminalExpression
Yêu cầu một thao tác thông dịch.

d.Ứng dụng :

Sử dụng mẫu Interpreter khi cần phiên dịch một ngôn ngữ mà ta có thể miêu tả các câu bằng cầu trúc cây cú pháp. Mẫu này hoạt động hiệu quả nhất khi :

- Cấu trúc ngữ pháp đơn giản. Với các cấu trúc ngữ pháp phức tạp, cấu trúc lớp của ngữ pháp trở nên quá lớn và khó kiểm soát, việc tạo ra các cây cú pháp sẽ tốn thời gian và bộ nhớ.
- Hiệu quả không phải là yếu tố quan trọng nhất. Các cách thức biên dịch hiệu quả nhất thường không áp dụng trực tiếp mẫu Interpreter mà phải biến đổi các biểu diễn thành các dạng khác trước.
-

e. Các mẫu liên quan

Cây cú pháp trừu tượng là một thể nghiệm trong mẫu Composite.

Flyweight chỉ ra cách chia sẻ ký pháp đầu cuối trong phạm vi của cây cú pháp trừu tượng.

Interpreter thường sử dụng một Iterator để duyệt cấu trúc.

Visitor có thể được sử dụng để duy trì hành vi trên mỗi nút trong cây cú pháp trừu tượng của lớp.

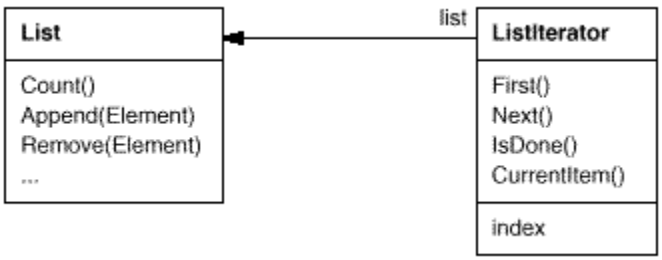
16.Iterator

a. Vấn đề đặt ra

Một đối tượng tập hợp như là một danh sách cũng cung cấp cho ta các phương thức truy cập các thành phần của nó. Tuy nhiên đôi lúc chúng ta cần duyệt các thành phần của danh sách theo những cách thức và tiêu chí khác nhau. Chúng ta không nên làm phồng giao diện của danh sách List với các phương thức cho các cách thức duyệt.

Mẫu Iterator cho phép chúng ta duyệt danh sách dễ dàng bằng cách tách rời chức năng truy cập và duyệt ra khỏi danh sách và đặt vào đối tượng iterator. Lớp Iterator sẽ định nghĩa một giao diện để truy cập các thành phần của danh sách, đồng thời quản lý cách thức duyệt danh sách hiện thời.

Ví dụ :



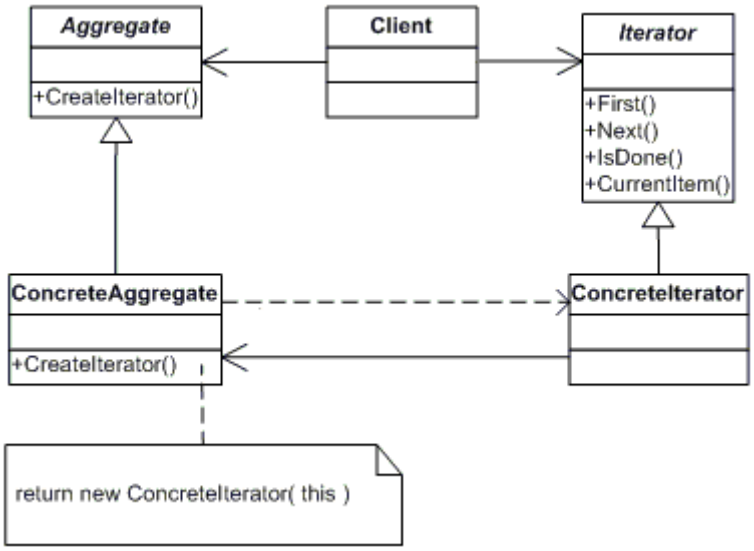
Mỗi đối tượng thuộc ListIterator quản lý một đối tượng List. Phương thức CurrentItem trả về đối tượng hiện thời, First chuyển đến đối tượng đầu tiên còn Next chuyển sang đối tượng tiếp theo; IsDone kiểm tra xem quá trình duyệt đã hoàn tất chưa.

Việc tách biệt giữa cơ chế duyệt và đối tượng List cho phép chúng ta xây dựng các đối tượng iterator với các tiêu khác nhau. Lấy ví dụ, hàm FilteringListIterator chỉ cung cấp quyền truy cập đến các thành phần thoả mãn điều kiện lọc.

b. Định nghĩa

Mẫu Iterator cung cấp khả năng truy cập và duyệt các thành phần của một tập hợp không cần quan tâm đến cách thức biểu diễn bên trong.

c. Sơ đồ UML



Iterator (AbstractIterator)

Định nghĩa một giao diện cho việc truy cập và duyệt các phần tử

ConcreteIterator (Iterator)

Cài đặt giao diện Iterator

Giữ dấu vết của vị trí hiện tại trong tập duyệt

Aggregate (AbstractCollection)

Định nghĩa một giao diện để tạo một đối tượng Iterator

ConcreteAggregate (Collection)

Cài đặt giao diện tạo Iterator để trả về một thể nghiệm đúng của ConcreteIterator.

d. Ứng dụng

- Hỗ trợ nhiều phương án duyệt một tập hợp.
- Đơn giản hoá giao diện tập hợp.
- Cho phép có lớn hơn một cách thức duyệt đối với một tập hợp tại một thời điểm.

Ứng dụng trong các trường hợp sau :

- Truy cập các thành phần của một tập hợp mà không cần quan tâm đến cách thức biểu diễn bên trong.
- Hỗ trợ nhiều phương án duyệt của các đối tượng tập hợp.
- Cung cấp giao diện chung cho việc duyệt các cấu trúc tập hợp.

e. Các mẫu liên quan

Iterator thường được sử dụng để duyệt một cấu trúc đệ quy như Composite.

Đa hình một Iterator dựa trên FactoryMethod để tạo thể nghiệm cho các lớp con tương ứng của Iterator.

Memento thường được sử dụng cùng với Iterator.Một Iterator có thể sử dụng một Memento để nắm bắt trạng thái của một Iterator. Iterator lưu trữ các memento ở bên trong.

17. Mediator

a. Vấn đề đặt ra

Cách thiết kế hướng đối tượng khuyến khích việc phân bố các ứng xử giữa các đối tượng. Việc phân chia đó có thể dẫn đến cấu trúc trong đó tồn tại rất nhiều liên kết giữa các đối tượng mà trong trường hợp xấu nhất là tất cả các đối tượng đều liên kết trực tiếp với nhau.

Lấy ví dụ về một hộp thoại sử dụng một cửa sổ để biểu diễn các widget như các nút, menu và entry field :

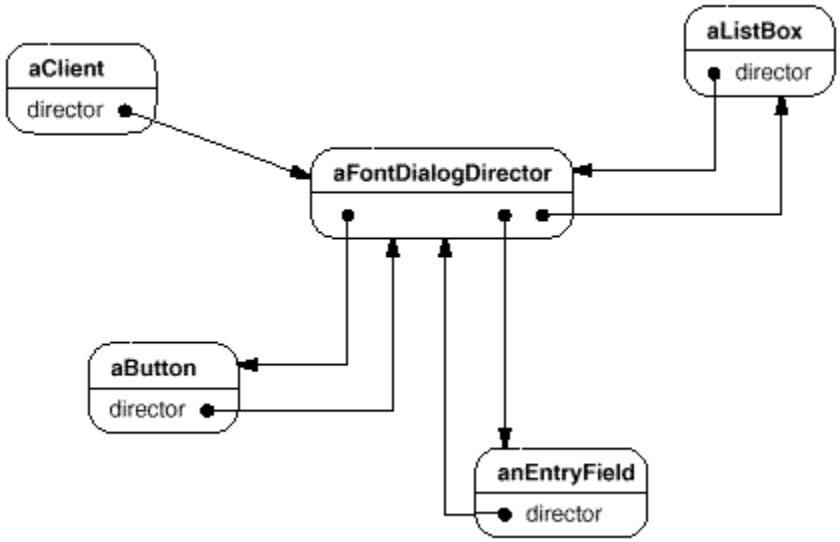
Thường các widgets trong cùng một cửa sổ sẽ có sự phụ thuộc lẫn nhau. Lấy ví dụ một nút bị disable khi một entry field rỗng, việc thay đổi chọn lựa trong listbox dẫn đến thay đổi giá trị trong entry field ...

Mỗi dialog sẽ có một ràng buộc riêng biệt giữa các widgets của nó vì thế cho dù nếu hai dialog cùng hiển thị các widgets như nhau nhưng chúng vẫn không thể sử dụng lại các lớp widgets đã xây dựng.

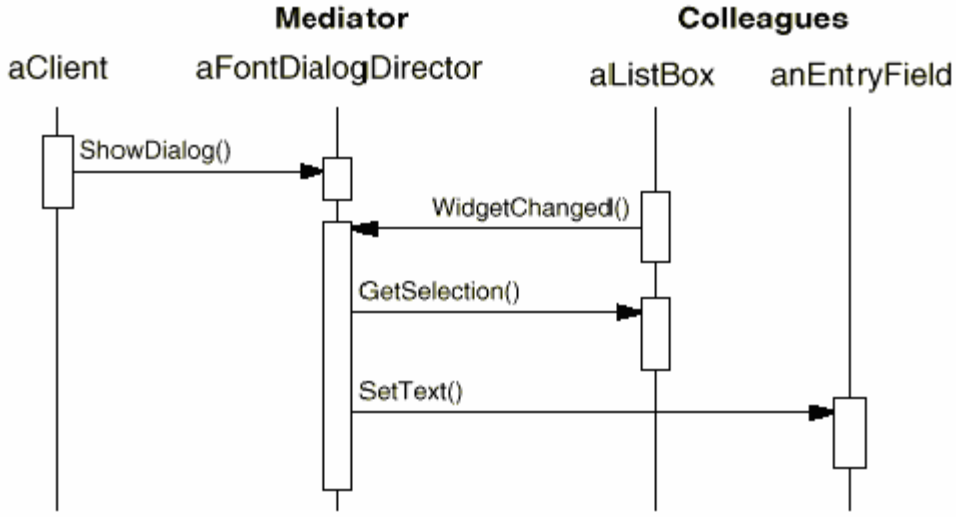
Ta có thể tránh vấn đề này bằng cách xây dựng đối tương trung gian mediator có nhiệm vụ điều khiển và xác định tương tác giữa một nhóm các đối tượng. Khi đó các đối tượng trong nhóm không cần liên kết trực tiếp với nhau mà chỉ cần liên kết với đối tượng mediator.

Ví dụ về việc thay đổi chọn lựa trong listbox dẫn đến thay đổi giá trị trong entry field sử dụng mediator FontDialogDirector :

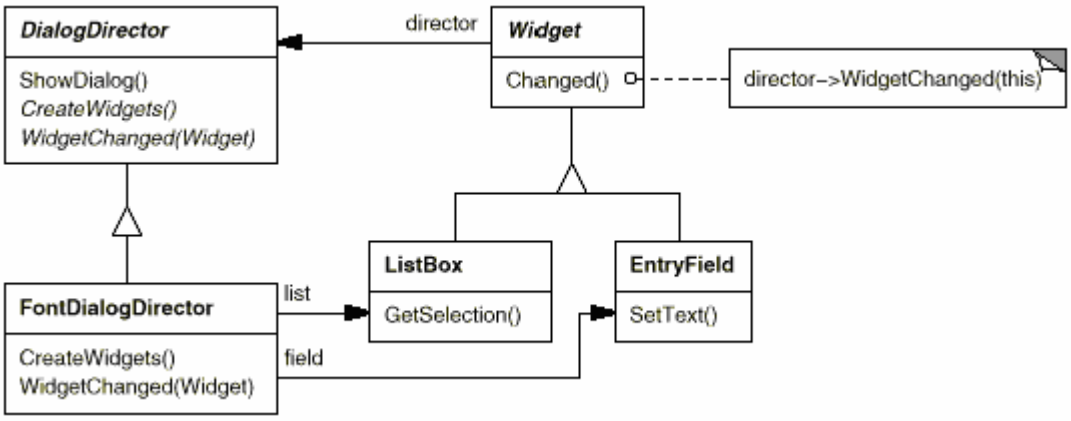
Biểu đồ đối tượng :



Biểu đồ diễn tiến :



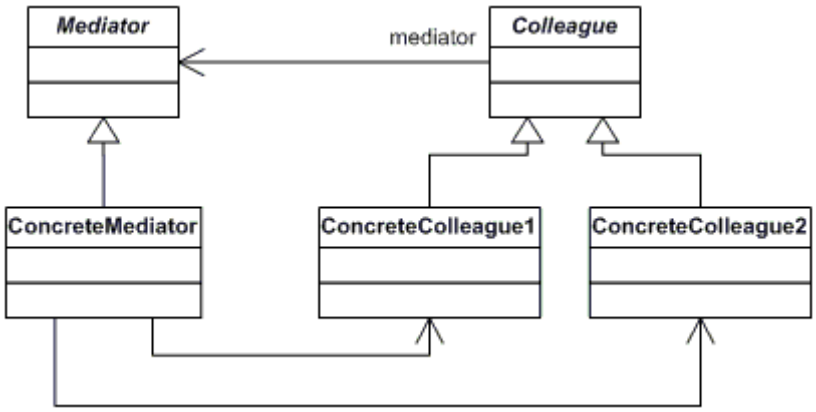
Biểu đồ lớp :



b.Định nghĩa

Mediator dùng để đóng gói cách thức tương tác của một tập hợp các đối tượng. Giảm bớt liên kết và cho phép thay đổi cách thức tương tác giữa các đối tượng.

c.Biểu đồ UML



Mediator (IChatroom)

- Định nghĩa một giao diện cho việc giao tiếp với đối tượng cộng tác.

ConcreteMediator (Chatroom)

- Xây dựng các hành vi tương tác giữa các đối tượng colleague
- Xác định các đối tượng colleague

Colleague classes (Participant)

- Mỗi lớp Colleague đều xác định đối tượng Mediator tương ứng
- Mỗi đối tượng colleague trao đổi với đối tượng mediator khi muốn trao đổi với colleague khác.

d.Ứng dụng

Mẫu Mediator có những ưu và nhược điểm sau :

- Giảm việc chia lớp con
- Giảm liên kết giữa các colleagues.
- Đơn giản hoá giao thức giữa đối tượng bằng cách thay các tương tác nhiều - nhiều bằng tương tác 1 - nhiều giữa nó và các colleagues.

- Trừu tượng hoá cách thức tương tác hợp tác giữa các đối tượng. Tạo ra sự tách biệt rõ ràng hơn giữa các tương tác ngoài và các đặc tính trong của đối tượng.
- Điều khiển trung tâm: thay sự phức tạp trong tương tác bằng sự phức tạp tại mediator.

Ứng dụng mediator vào trong các trường hợp sau :

- Một nhóm các đối tượng trao đổi thông tin một cách rõ ràng nhưng khá phức tạp dẫn đến hệ thống các kết nối không có cấu trúc và khó hiểu.
- Việc sử dụng lại một đối tượng gặp khó khăn vì nó liên kết với quá nhiều đối tượng khác.
- Cho phép tuỳ biến một ứng xử được phân tán trong vài lớp mà không phải phân nhiều lớp con.

e. Các mẫu liên quan

Facade khác với Mediator ở chỗ nó trừu tượng một hệ thống con của các đối tượng để cung cấp một giao diện tiện ích hơn. Giao thức của nó theo một hướng duy nhất đó là, các đối tượng Facade tạo ra các yêu cầu của các lớp hệ thống con nhưng không có chiều ngược lại. Ngược lại Mediator cho phép các hành vi kết hợp mà các đối tượng cộng tác không thể cung cấp và giao thức này là không đa hướng.

Các cộng tác có thể đi giao tiếp với Mediator bằng cách sử dụng mẫu Observer.

18. Memento

a. Vấn đề đặt ra

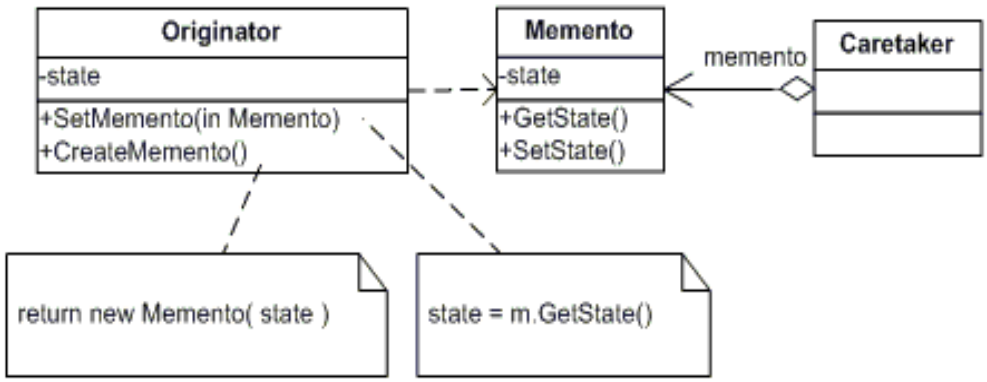
Đôi lúc, việc lưu lại trạng thái của một đối tượng là cần thiết. Ví dụ khi xây dựng cơ chế checkpoints và undo để phục hồi hệ thống khỏi trạng thái lỗi. Tuy nhiên các đối tượng thường phải che dấu một vài hoặc tất cả các thông tin trạng thái của mình, làm cho chúng không thể được truy cập từ ngoài.

Chúng ta có thể giải quyết vấn đề này với mẫu Memento. memento là đối tượng có chức năng lưu lại trạng thái nội tại của một đối tượng khác gọi là memento's originator. Cơ chế undo sẽ yêu cầu một memento từ originator khi nó cần khôi phục lại trạng thái của originator. Cũng chỉ originator mới có quyền truy xuất và lưu trữ thông tin vào memento vì nó trong suốt đối với các đối tượng còn lại.

b. Định nghĩa

Memento là mẫu thiết kế có thể lưu lại trạng thái của một đối tượng để khôi phục lại sau này mà không vi phạm nguyên tắc đóng gói.

c. Sơ đồ UML



Memento

- Lưu trữ trạng thái của đối tượng Originator.
- Bảo vệ, chống truy cập từ các đối tượng khác originator.

Originator

- Tạo memento chứa hình chụp trạng thái của mình
- Sử dụng memento để khôi phục về trạng thái cũ.

Caretaker

- Có trách nhiệm quản lý các memento.
- Không thay đổi hoặc truy xuất nội dung của memento.

d.Ứng dụng

Memento có những đặc điểm

- Đảm bảo ranh giới của sự đóng gói.
- Đơn giản Originator. Trong các thiết kế hỗ trợ khôi phục trạng thái khác, Originator có chức năng lưu trữ các phiên bản trạng thái của mình và do đó phải thi hành các chức năng quản lý lưu trữ.
- Sử dụng memento có thể gây ra chi phí lớn nếu Originator có nhiều thông tin trạng thái cần lưu trữ hoặc nếu việc ghi lại và khôi phục trạng thái diễn ra với tần suất lớn.
- Việc đảm bảo chỉ originator mới có quyền truy cập memento là tương đối khó xây dựng ở một số ngôn ngữ lập trình.
- Chi phí ẩn của việc lưu trữ memento : caretaker có nhiệm vụ quản lý cũng như xoá bỏ các memento nó yêu cầu tạo ra. Tuy nhiên nó không được biết kích thước của memento là bao nhiêu và do đó có thể tốn nhiều không gian bộ nhớ khi lưu trữ memento.

Ứng dụng khi

- Cần lưu lại trạng thái nội bộ của một đối tượng để có thể khôi phục về trạn thái đó sau này.
- Xây dựng giao diện trực tiếp để truy xuất thông tin trạng thái sẽ phơi bầy cấu trúc và phá hỏng tính đóng gói của đối tượng.

e. Các mẫu liên quan

Các Command có thể sử các memento để duy trì trạng thái cho các thao tác có khả năng phục hồi được.

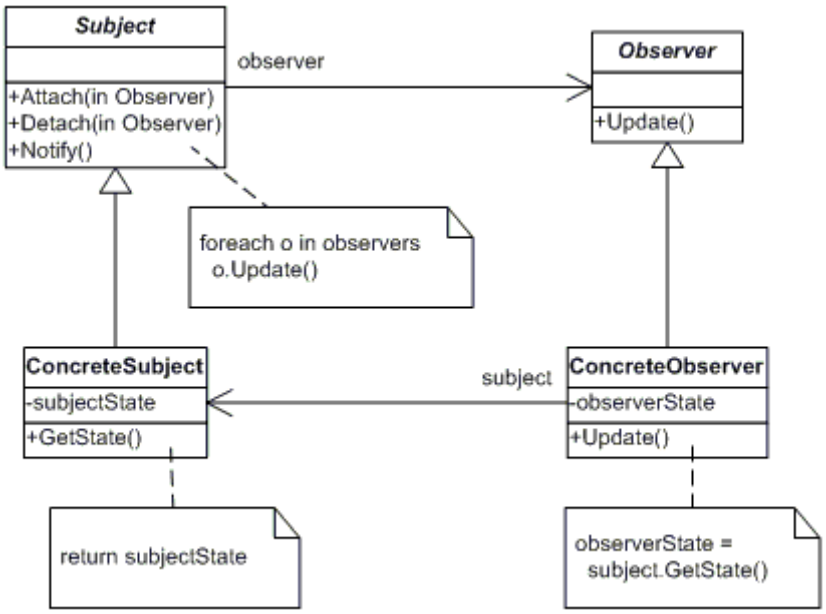
Các Memento có thể được sử dụng cho vòng lặp sớm hơn.

19. Observer

a.Định nghĩa

Observer định nghĩa một phụ thuộc 1- nhiều giữa các đối tượng để khi một đối tượng thay đổi trạng thái thì tất cả các phục thuộc của nó được nhận biết và cập nhật tự động

bSơ đồ UML



Subject (Stock)

- Hiểu về các Observer của nó. Một số lượng bất kỳ Observer có thể theo dấu một chủ thể nào đó.
- Cung cấp một giao diện cho việc gắn và tách các đối tượng Observer

ConcreteSubject (IBM)

- Lưu trữ trạng thái của ConcreteObserver cần quan tâm.
- Gửi tín hiệu đến các observer của nó khi trạng thái của nó đã thay đổi.

Observer (IInvestor)

- Định nghĩa một giao diện cập nhật cho các đối tượng mà sẽ nhận tín hiệu của sự thay đổi tại chủ thể.

ConcreteObserver (Investor)

- Duy trì một tham chiếu tới một đối tượng ConcreteSubject.
- Lưu trữ các trạng thái cố định.
- Cài đặt giao diện cập nhật của Observer đẻ giữ các trạng thái cố định của nó.

c. Mẫu liên quan

Bằng việc đóng gói các ngữ nghĩa cập nhật phức tạp, ChangeManager hoạt động như một Mediator giữa các chủ thể và các Observer.

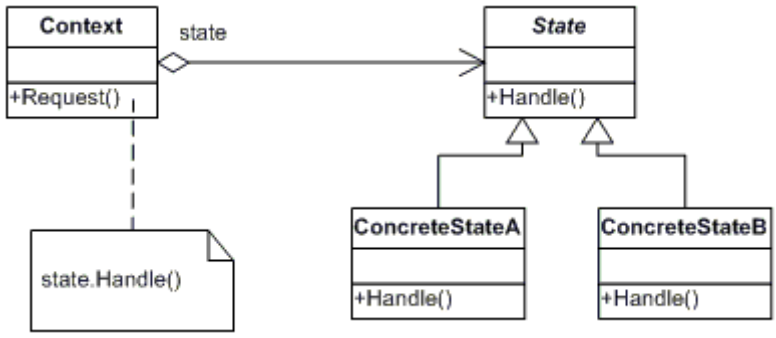
ChangeManager có thể sử dụng mẫu Singleton để cho việc truy nhập nó là đồng nhất và tổng thể.

20. Sate

a. Định nghĩa

Observer là mẫu thiết kế cho phép một đối tượng thay đổi các hành vi của nó khi các trạng thái bên trong của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi các lớp của nó.

b. Sơ đồ UML



Context (Account)

- Định nghĩa giao diện mà đối tượng khách quan tâm
- Duy trì một thể nghiệm của một lớp ConcreteState mà định nghĩa trạng thái hiện tại

State (State)

- Định nghĩa một giao diện cho việc đóng gói hành vi kết hợp với trạng thái đặc biệt của Context.

Concrete State (RedState, SilverState, GoldState)

- Mỗi lớp con cài đặt một hành vi kết hợp với một trạng thái của Context.

c. Các mẫu liên quan

Mẫu Flyweight giải thích khi nào các đối tượng State có thể được phân tách và được phân tách như thế nào.

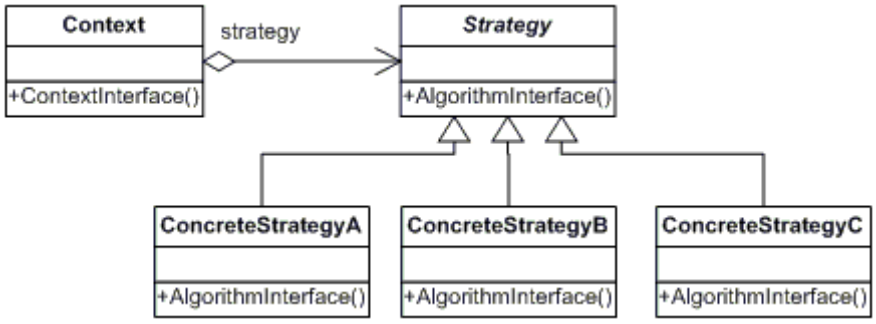
Các đối tượng State thường là các Singleton.

21.Strategy

a. Định nghĩa

Strategy là mẫu thiết kế dùng để định nghĩa một họ các thuật toán, đóng gói mỗi thuật toán đó và làm cho chúng có khả năng thay đổi dễ dàng.Strategy cho phép giả thuật tùy biến một cách độc lập tại các Client sử dụng nó.

b.Sơ đồ UML



Strategy (SortStrategy)

- Khai báo một giao diện thông thường tới tất cả các giải thuật được hỗ trợ. Context sử dụng giao diện này để gọi các giải thuật được định nghĩa bởi một ConcreteStrategy.

ConcreteStrategy (QuickSort, ShellSort, MergeSort)

- Cài đặt giải thuật sử dụng giao diện Strategy

Context (SortedList)

- Được cấu hình với một đối tượng ConcreteStrategy
- Duy trì một tham chiếu tới một đối tượng Stategy
- Có thể định nghĩa một giao diện để cho Strategy truy nhập dữ liệu của nó.

c.Các mẫu liên quan

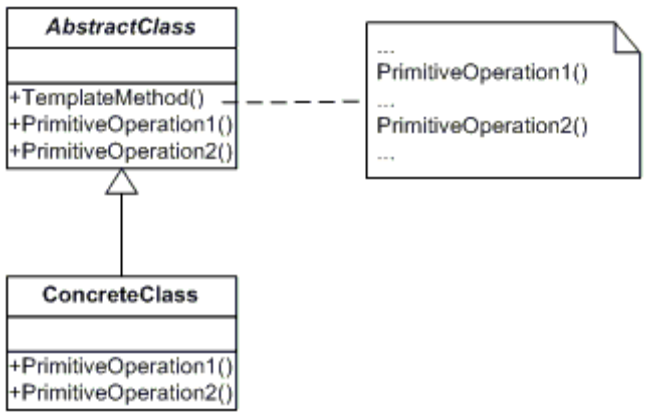
Các đối tượng Strategy thường tạo ra các Flyweight tốt hơn.

22.Template Method

a. Định nghĩa

Template Method là mẫu xác định sườn của một giải thuật trong một thao tác, theo một số bước của các phân lớp. Template Method cho phép các lớp con xác định lại chắc chắn một số bước của một giải thuật bên ngoài cấu trúc của giải thuật đó.

b. Sơ đồ UML



AbstractClass:

- Định nghĩa các primitive operation (thao tác nguyên thủy) trừu tượng, các thao tác này định nghĩa các lợp con cụ thể để thực hiện các bước của một giải thuật.

- Cài đặt một template method định nghĩa sườn của một giải thuật. Template method này gọi các thao tác nguyên thủy cũng như các thao tác được định nghĩa trong AbstractClass hoặc một số các đối tượng khác.

ConcreteClass:

- Thực hiện các thao tác nguyên thủy để thực hiện các bước đã chỉ ra trong các lớp con của giải thuật

c.Vận dụng

Template Method nên sử dụng trong các trường hợp:

- Thực hiện các phần cố định của một giải thuật khi đặt nó vào các lớp con để thực hiện hành vi có thể thay đổi.
- Khi các lớp hành vi thông thường nên được phân tách và khoanh vùng trong một lớp thông thường để tránh sự giống nhau về mã.
- Điều khiển mở rộng các lớp con. Ta có thể định nghĩa một template method, template method này gọi các thao tác "hook" tại các điểm đặc biệt, bằng cách đó cho phép các mở rộng chỉ tại các điểm đó.

d.Các mẫu liên quan

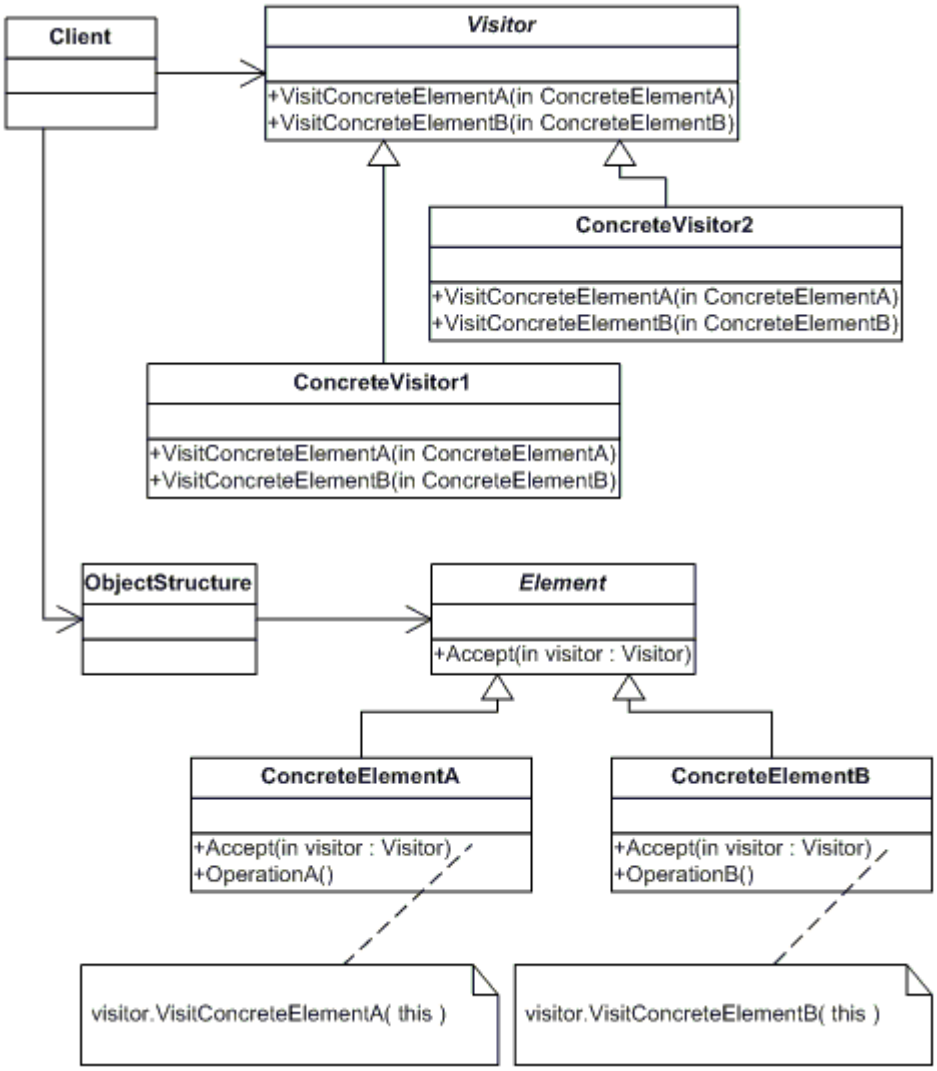
Các template Method sử dụng sự kế thừa để thay đổi các bộ phận của một giải thuật . Các Strategy sử dụng sự ủy nhiệm để thay đổi hoàn toàn một thuật toán.

23.Visitor

a. Định nghĩa

Visitor là mẫu thiết kế xác định sườn của một giải thuật trong một thao tác, theo một số bước của các phân lớp. Template Method cho phép các lớp con xác định lại chắc chắn một số bước của một giải thuật bên ngoài cấu trúc của giải thuật đó.

b. Sơ đồ UML



- Visitor:
  + Đưa ra một thao tác Visit cho mỗi lớp của ConcreteElement trong cấu trúc đối tượng. Tên và dấu hiệu của các thao tác này nhận dạng lớp gửi yêu cầu Visit tới visitor, nó cho phép visitor quyết định lớp cụ thể nào của thành phần được thăm. Sau đó visitor có thể truy nhập thành phần trực tiếp thông qua giao diện đặc biệt của nó.
- ConcreteVisitor:
  + Thực hiện mỗi thao tác được đưa ra bởi Visitor. Mỗi thao tác thực hiện một phần của giải thuật định nghĩa cho lớp phù hợp của đối tượng trong cấu trúc. ConcreteVisitor cung cấp ngữ cảnh cho giải thuật và lưu trữ trạng thái cục bộ của nó.
- Element:
  + Định nghĩa một thao tác Accept, thao tác này mang một visitor như là một đối số.
- ConcreteElement:
  + Thực hiện một thao tác Accept, thao tác này mang một visitor như là một đối số.
- ObjectStructure:
  + Có thể đếm các thành phần của nó.

+ Có thể cung cấp một giao diện mức cao cho phép visitor thăm các thành phần của nó.
+ Có thể là một composite hoặc một sưu tập như danh sách hay tập hợp.

c. Vận dụng:

Sử dụng Visitor pattern khi:

- Một cấu trúc đối tượng chứa đựng nhiều lớp của các đối tượng với các giao diện khác nhau, và ta muốn thực hiện các thao tác trên các đối tượng này thì đòi hỏi các lớp cụ thể của chúng.
- Nhiều thao tác khác nhau và không có mối liên hệ nào cần được thực hiện trên các đối tượng trong một cấu trúc đối tượng, và ta muốn tránh "làm hỏng" các lớp của chúng khi thực hiện các thao tác đó. Visitor cho phép ta giữ các thao tác có mối liên hệ với nhau bằng cách định nghĩa chúng trong một lớp. Khi một cấu trúc đối tượng được chia sẻ bởi nhiều ứng dụng, sử dụng Visitor để đặt các thao tác này vào trong các ứng dụng cần chúng.
- Các lớp định nghĩa các cấu trúc đối tượng hiếm khi thay đổi, nhưng ta muốn định nghĩa các thao tác mới trên các cấu trúc. Thay đổi các lớp cấu trúc yêu cầu định nghĩa lại giao diện cho tất cả các visitor.

d. Mẫu liên quan

Các Visitor có thể được sử dụng để cung cấp một thao tác trên một cấu trúc đối tượng được định nghĩa bởi mẫu Composite. Visitor có thể được cung cấp để làm thông dịch.

C. Ứng dụng design pattern trong thực tế phân tích thiết kế phần mềm hướng đối tượng

Ứng dụng của design pattern trong thực tế phân tích thiết kế phần mềm hướng đối tượng là rất lớn. Hầu như cứ ở đâu có phần mềm hướng đối tượng thì ở đó có design pattern. Design pattern được vận dụng linh hoạt và dưới nhiều hình thức khác nhau.Trong nội dung đồ án môn học này chúng tôi xin trình bày một vài ứng dụng điển hình của Design pattern.

I.Framework và idom

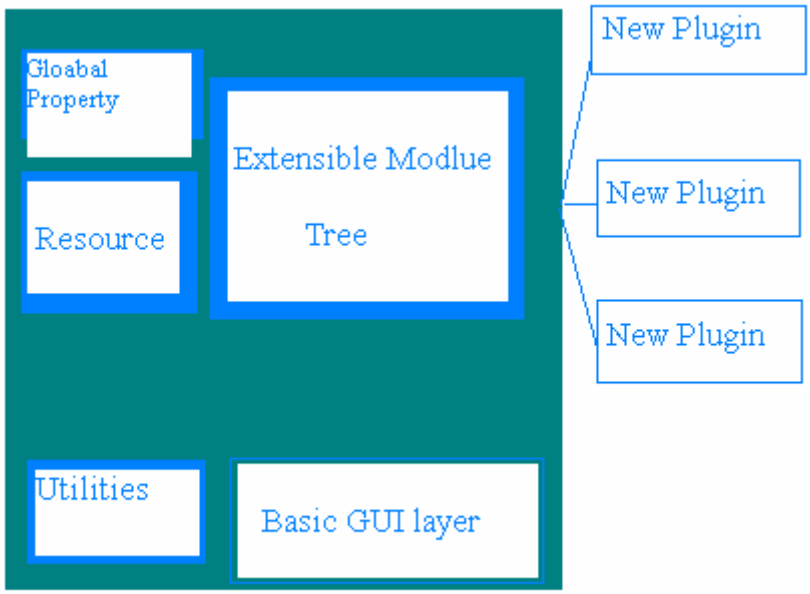
Cả Framework và idom có liên quan đến mẫu, nhưng giữa chúng có sự khác nhau rõ ràng.

Framework thì tổng quát hơn và có thể áp dụng cho một lĩnh vực cụ thể.Ví dụ Framework tài chính sẽ chứa các lớp về tài chính trong các mối quan hệ được xác định bởi các mẫu thiết kế, từ framework này có thể phát triển để tạo ra các ứng dụng về tài chính.Framework là một tập các lớp được đóng gói để có thể dùng lại cho các ứng dụng cụ thể. Ứng dụng thực hiện Customize các lớp này như kế thừa, dẫn xuất để tạo ra các lớp mới phục vụ cho mục đích của ứng dụng.Framework phải có được những đặc điểm là : nhỏ gọn,và đầy đủ dễ customize, tính khái quát cao,... Tập các lớp trong framework được cài đặt và thiết lập các mối quan hệ theo các mẫu design pattern.

Idom là một tập các chỉ dẫn về cách cài đặt các khía cạnh của một hệ thống phần mềm viết bằng một ngôn ngữ cụ thể.Coplien (1992) lần đầu tiên đã xuất bản một tập các idom cho việc dùng ngôn ngữ C++.Các idom này ghi lại các kinh nghiệm của các lập trình viên chuyên nghiệp C++, để từ đó các lập trình viên không chuyên có thể giải quyết các vấn đề thường gặp khi viết chương trình bằng C++.

II.Kiến trúc Add – Ins

Đây là một mô hình ứng dụng cho phép tạo ra một giao diện ghép nối các môđun ứng dụng một cách dễ dàng. Ứng dụng gồm có nhân ứng dụng (core) và các môđun ghép nối là các gói DLL. Cấu hình của ứng dụng được lưu vào các file định dạng XML.



Global property thường là các mẫu thực thể (datasim) có thể cấu hình các thành phần được.

Resource : thường là các lớp singleton quản lý tài nguyên tập trung bao gồm

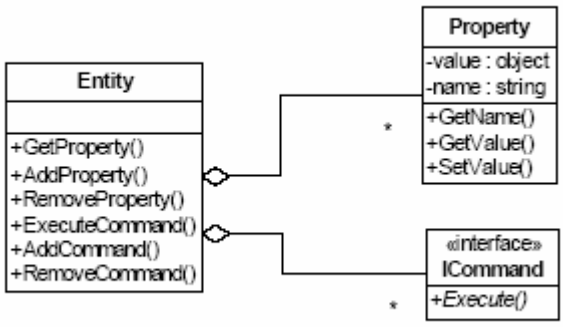
- Icon Resource
- Error Message Resource
- Language Resource

Chúng được gộp vào một đối tượng quản lý là SingletonResourceManager cung cấp mọi điểm truy cập đồng nhất trong đối tượng.

Basic GUI layer : cung cấp các giao diện đồ hoạ cơ bản.Thường sử dụng các mẫu Abstract Factory, Abstract Method Proxy, Facade và Memento (kết hợp với XML).

Extensible Module Tree : Đây là phần quan trọng của nhân ứng dụng. Nó cung cấp các giao diện ghép nối với các mô đun bên ngoài. Các lớp trong phần này thường được cài đặt dưới dạng các Entity patterns (mẫu thực thể), hay còn gọi là các Codon.

Mỗi codon gồm có :ID (name - chỉ duy nhất một tên cho một codon), Label(nhãn có thể trùng nhau) và Class (đây là mã thực thi của codon đó). Class này thường là các Command patterns.



Cấu trúc một Codon

D.Các mẫu thiết kế hiện đại

I.Gamma Patterns

II.Entity Pattern (datasim) : Mẫu thực thể

Mẫu thực thể là :

- Một lớp động
- Không có nhiều các thành phần thuộc tính và phương thức cố định
- Các thành phần có thể cấu hình được
- Nó là thế hệ nối tiếp của mẫu Gamma patterns.

Đặc điểm của mẫu thực thể :

- Rất phức tạp
- Có thể thao tác được bằng các công cụ khác
- Sử dụng XML và các hệ quản trị cơ sở dữ liệu để cấu hình các thành phần
- Nó là chất liệu để tạo ra giao diện
- Rất cần thiết cho các môi trường động

Những vấn đề chuyên sâu của mẫu này nội dung của đồ án môn học xin không đưa ra ở đây.Thông tin về mẫu này có thể tham khảo tại trang web www.datasim.com.

III.Concurrent Patterns :

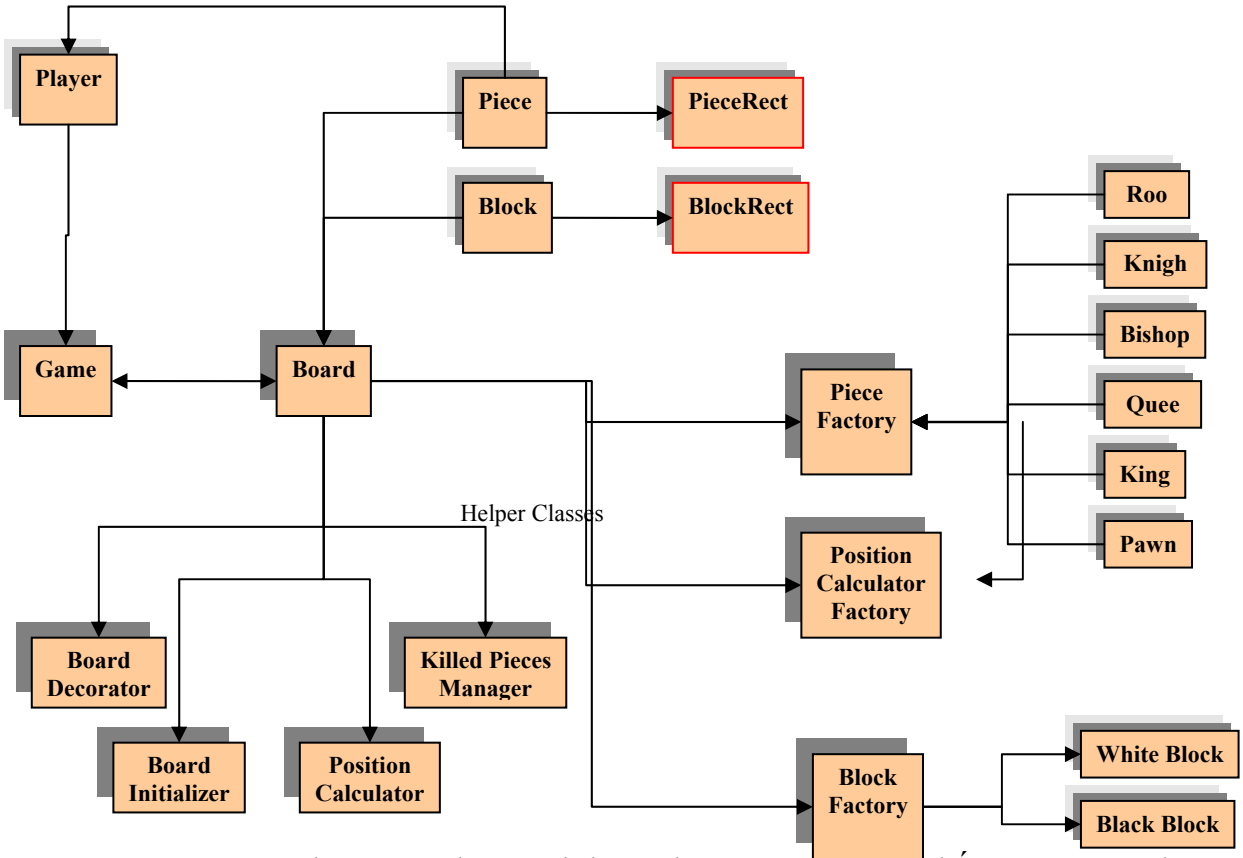
- Đây là mẫu thiết kế cấu thành nên các hệ thống thời gian thực.
- Các đối tượng được phân tán
- Tích hợp với tiến trình của mẫu thực thể.

Mẫu này thuộc nhóm gồm có các mẫu :

- Service Access/Configuration
- Event Handling
- Synchronisation
- Concurrency

E. Xây dựng ứng dụng Chess sử dụng Design pattern

Sơ đồ các lớp của ứng dụng



- Trong ứng dụng này chúng tôi đã sử dụng các mẫu thiết kế Decorator, Abstract Factory ,Factory Method để giải quyết vấn đề.

F. Tài liệu tham khảo

I. Sách

1.*Design patterns Elements of Reusable Object Oriented Software*

2. *The design patterns SmallTalk Companion*

3. *Analysis Patterns: Reusable Object Models*

4. *Concurrent Programming in Java™: Design Principles and Patterns*

5. *Pattern Languages of Program Design*

6. *Pattern Languages of Program Design 2*

7. *Pattern Languages of Program Design 3*

8. *ThinkInPatterns*

II. Địa chỉ website

http://www.dofactory.com/Patterns

http://patterndigest.com