

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA CÔNG NGHỆ PHẦN MỀM



Báo cáo môn học
DESIGN PATTERN

GVHD: Thầy Phạm Thi Vương

Sinh viên thực hiện:

- | | |
|------------------------|----------|
| 1. Văn Vũ Tuấn | 12520487 |
| 2. Phạm Ngọc Linh | 12520228 |
| 3. Huỳnh Đức Đăng Khoa | 12520204 |

Thành phố Hồ Chí Minh – 12/12016

[illegible]

NỘI DUNG

Nội dung	3
1. Tổng quan	6
1.1. Giới thiệu đề tài.....	6
1.2. Phạm vi báo cáo	6
2. Giới thiệu về Mẫu thiết kế	7
2.1. Khái niệm Mẫu thiết kế:.....	7
2.2. Lịch sử ra đời	7
2.3. Phân loại.....	8
2.3.1. Mẫu kiến tạo	8
2.3.2. Mẫu cấu trúc	9
2.3.3. Mẫu hành vi.....	10
2.4. Lợi ích khi sử dụng mẫu.....	12
2.5. Cấu trúc trình bày mẫu	13
2.6. Một số nguyên tắc	14
3. Các mẫu Thiết kế Gang of Four	15
3.1. Mẫu Singleton	15
3.1.1. Tên, phân loại, bí danh	15
3.1.2. Mục đích, ý định.....	15
3.1.3. Động lực sử dụng	15
3.1.4. Khả năng ứng dụng	15
3.1.5. Cấu trúc	15
3.1.6. Các thành viên	16
3.1.7. Sự cộng tác	16
3.1.8. Các hệ quả mang lại	16
3.1.9. Các chú ý liên quan đến cài đặt	17
3.1.10. Các ví dụ hệ thống thực tế	18
3.1.11. Các mẫu liên quan	19
3.1.12. Mã nguồn minh họa.....	19
3.2. Mẫu Façade	21
3.2.1. Tên, phân loại, bí danh.....	21
3.2.2. Mục đích, ý định	21
3.2.3. Động lực sử dụng.....	21
3.2.4. Khả năng ứng dụng.....	22
3.2.5. Cấu trúc.....	23
3.2.6. Các thành viên	23

3.2.7	Sự cộng tác	23
3.2.8	Các hệ quả mang lại.....	24
3.2.9	Các chú ý liên quan đến cài đặt	24
3.2.10	Các ví dụ về hệ thống thực tế	25
3.2.11	Các mẫu liên quan	25
3.2.12	Mã nguồn minh họa.....	26
3.3	Mẫu Proxy.....	29
3.3.1	Tên, phân loại, bí danh.....	29
3.3.2	Mục đích, ý định	29
3.3.3	Động lực sử dụng.....	30
3.3.4	Khả năng ứng dụng.....	31
3.3.5	Cấu trúc.....	31
3.3.6	Các thành viên	32
3.3.7	Sự cộng tác	32
3.3.8	Các hệ quả mang lại.....	33
3.3.9	Các chú ý liên quan đến cài đặt	33
3.3.10	Các ví dụ về hệ thống thực tế	33
3.3.11	Các mẫu liên quan	33
3.3.12	Mã nguồn minh họa.....	34
3.4	Mẫu Iterator.....	36
3.4.1	Tên, phân loại, bí danh.....	36
3.4.2	Mục đích, ý định	36
3.4.3	Động lực sử dụng.....	36
3.4.4	Khả năng ứng dụng.....	37
3.4.5	Cấu trúc.....	37
3.4.6	Các thành viên	38
3.4.7	Sự cộng tác	38
3.4.8	Các hệ quả mang lại.....	38
3.4.9	Các chú ý liên quan đến cài đặt	39
3.4.10	Các ví dụ về hệ thống thực tế	39
3.4.11	Các mẫu liên quan	40
3.4.12	Mã nguồn minh họa.....	40
3.5	Mẫu Observer.....	44
3.5.1	Tên, phân loại, bí danh.....	44
3.5.2	Mục đích, ý định	45
3.5.3	Động lực sử dụng.....	45
3.5.4	Khả năng ứng dụng.....	46

3.5.5	Cấu trúc.....	47
3.5.6	Các thành viên	47
3.5.7	Sự cộng tác	48
3.5.8	Các hệ quả mang lại.....	48
3.5.9	Các chú ý liên quan đến cài đặt	49
3.5.10	Các ví dụ về hệ thống thực tế	49
3.5.11	Các mẫu liên quan	50
3.5.12	Mã nguồn minh họa.....	50
4.	Phụ lục	53
4.1.	Phân loại.....	53
4.2.	Tần suất sử dụng các mẫu	53
4.3.	Độ khó.....	54
5.	Tài liệu tham khảo	55

1. Tổng quan

1.1. Giới thiệu đề tài

Mẫu thiết kế là một kỹ thuật giành cho lập trình hướng đối tượng. Nó cung cấp cho chúng ta các giải pháp đã được chứng thực từ những người đi trước trong các tình huống mà chúng ta có thể gặp phải trong quá trình phát triển phần mềm. Các lập trình viên và các nhà phân tích đều cần có kiến thức về mẫu thiết kế để nâng cao khả năng tư duy và kỹ năng giải quyết vấn đề.

Chúng ta có thể dễ dàng nhận ra một lập trình viên có kinh nghiệm hay không thông qua kiến thức của anh ta về lập trình hướng đối tượng và mẫu thiết kế. Thật vậy, một lập trình viên có kinh nghiệm nhất định sẽ nhận biết được khi nào nên và không nên sử dụng các mẫu thiết kế trong các tình huống mà anh ta gặp phải trong quá trình lập trình.

Các nội dung trong báo cáo được trình bày dựa trên các kiến thức mà nhóm thu thập và tìm hiểu về các mẫu thiết kế và các ứng dụng của chúng trong quá trình học tập tại trường. Nội dung của báo cáo bao gồm các phần chính:

- Phần 1: Tổng quan về đề tài
- Phần 2: Tổng quan về các mẫu thiết kế
- Phần 3: Trình bày về 5 mẫu thiết kế được nhóm chọn ra từ 23 mẫu.
- Phần 4: Phụ lục về các mẫu thiết kế
- Phần 5: Thông tin về tài liệu tham khảo.

1.2. Phạm vi báo cáo

Trong phạm vi báo cáo này nhóm xin trình bày về 5 mẫu thiết kế Gang of Four có tần suất sử dụng tương đối cao, được áp dụng trong nhiều lĩnh vực khác nhau, bao gồm: Façade, Iterator, Observe, Proxy và Singleton.

2. Giới thiệu về Mẫu thiết kế

2.1. Khái niệm Mẫu thiết kế:

Trong quá trình thiết kế phần mềm, chúng ta có thể gặp phải một vài vấn đề khó giải quyết, hoặc cách giải quyết của chúng không phù hợp, hoặc giải quyết vấn đề không trọn vẹn, chúng ta có thể nghĩ đến các mẫu thiết kế phần mềm. Nói chung, mẫu thiết kế là một giải pháp chung được các lập trình viên và các nhà phát triển đi trước tìm ra và tổng hợp lại, chúng thường được áp dụng trong việc giải quyết một số vấn đề thường xảy ra trong quá trình chúng ta thiết kế phần mềm.

Nói một cách khác, ta có thể coi một mẫu thiết kế như một khuôn mẫu mô tả cách thức giải quyết một vấn đề mà ta có thể sử dụng trong nhiều tình huống khác nhau.

2.2. Lịch sử ra đời

Khái niệm mẫu thiết kế mà chúng ta đang dùng có nguồn gốc trong lĩnh vực kiến trúc xây dựng. Một kỹ sư người Áo là Christopher Alexander (sinh năm 1936) đã nhận thấy rằng có nhiều thiết kế được sử dụng nhiều lần trong quá trình thiết kế các công trình kiến trúc. Ông đã xuất bản hai quyển sách có tầm ảnh hưởng lớn là *A Pattern Language: Towns, Buildings, Construction* (xuất bản năm 1977) và *The Timeless Way of Building* (xuất bản năm 1979). Trong các quyển sách này, ông đã mô tả các mẫu thiết kế kiến trúc như sau:

“Mỗi mẫu mô tả một vấn đề mà chúng xảy ra lặp đi lặp lại trong môi trường của chúng tôi. Đồng thời mô tả giải pháp cốt lõi để giải quyết vấn đề đó, theo một cách mà bạn có thể sử dụng giải pháp này hàng triệu lần mà không cần phải giải quyết một vấn đề giống nhau hai lần”.

Gần hai thập kỷ sau, các chuyên gia trong lĩnh vực phần mềm bắt đầu kết hợp các nguyên lý của Alexander và việc tạo ra các tài liệu hướng dẫn đầu tiên về mẫu thiết kế cho các nhà phát triển. Vào năm 1994, hội nghị đầu tiên về các mẫu thiết kế được tổ chức. Tên của hội nghị là Pattern Languages of Program Design (PLoP), tạm dịch là *“Các ngôn ngữ mẫu của thiết kế chương trình”*. Không lâu sau đó (1995), cuốn sách có tầm ảnh hưởng lớn nhất đến các mẫu thiết kế phần mềm là *Design Patterns: Elements*

of *Reusable Object-Oriented Software* đã được xuất bản, nó giới thiệu 23 mẫu thiết kế đầu tiên, do bốn người Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides đồng tác giả. Các tác giả sau này thường được nhắc đến với biệt danh *Gang of Four* hay GoF. Kể từ đó, càng ngày càng có nhiều mẫu thiết kế được ghi chép và phân loại. Tuy nhiên, 23 mẫu GoF đầu tiên được biết đến và được ứng dụng rộng rãi nhất.

2.3. Phân loại

Các mẫu thiết kế được chia thành 3 loại chính: Mẫu kiến tạo, mẫu kiến trúc và mẫu hành vi.

2.3.1. Mẫu kiến tạo

Mẫu kiến tạo (Creational Pattern) trừu tượng hóa quá trình khởi tạo. Chúng giúp chúng ta tạo nên một hệ thống độc lập với cách mà các đối tượng của hệ thống được tạo ra, kết hợp và thể hiện. Các mẫu kiến tạo có thể được chia thành 2 nhóm nhỏ hơn là các mẫu Kiến tạo Lớp (class-creation) và các mẫu Kiến tạo Đối tượng (object-creation). Trong khi mẫu Kiến tạo Lớp sử dụng kế thừa để thay đổi lớp cần được khởi tạo, mẫu Kiến tạo Đối tượng sẽ ủy nhiệm quá trình khởi tạo đến một đối tượng khác.

Mẫu kiến tạo có vai trò quan trọng bởi khi hệ thống phát triển phụ thuộc theo thời gian. Khi đó hệ thống cần được phát triển dựa vào sự kết hợp các đối tượng nhiều hơn sự kế thừa các lớp.

Khi điều đó xảy ra, việc hard-coding các tập cố định các hành vi sẽ chuyển thành việc định nghĩa một tập nhỏ hơn các hành vi cơ bản để chúng có thể được kết hợp thành bất kỳ các thành phần phức tạp nào khác nhau. Khi đó, việc tạo ra một đối tượng với các hành vi cụ thể yêu cầu nhiều công sức hơn so với việc chỉ đơn giản là khởi tạo một lớp.

Có hai chủ đề quan trọng trong các mẫu khởi tạo. Đầu tiên, chúng đóng gói tất cả các thông tin về các lớp con cụ thể mà hệ thống sẽ sử dụng. Thứ hai, chúng ẩn đi cách mà các thể hiện của lớp con được hệ thống sử dụng sẽ được tạo ra và kết hợp với nhau. Tất cả những gì mà các hệ thống sử dụng các đối tượng đó biết được là giao diện của chúng đã được định nghĩa bởi các lớp ảo.

Mẫu kiến tạo cung cấp cho chúng ta sự linh hoạt vô cùng lớn, trong việc đối tượng nào sẽ được tạo ra, cái gì sẽ tạo ra chúng, làm thế nào để chúng tạo ra, và khi nào. Mẫu kiến tạo cho phép chúng ta cấu hình một hệ thống với các đối tượng “sản phẩm” thuộc

đủ các thể loại cấu trúc và hành vi khác nhau. Việc cấu hình có thể tĩnh (được quy định tại thời điểm biên dịch – compile-time) hay động (trong khi chương trình đang trong thời gian chạy – run-time).

Các mẫu kiến tạo của GoF bao gồm 5 mẫu, được giới thiệu sơ lược sau đây:

MẪU	MÔ TẢ NGẮN GỌN
ABSTRACT FACTORY	Cung cấp một interface cho việc tạo lập các đối tượng (có liên hệ với nhau) mà không cần qui định lớp hay xác định lớp cụ thể (concrete) khi tạo mỗi đối tượng.
FACTORY METHOD	Định nghĩa Interface để sinh ra đối tượng nhưng để cho lớp con quyết định lớp nào được dùng để sinh ra đối tượng Factory method cho phép một lớp chuyển quá trình khởi tạo đối tượng cho lớp con
SINGLETON	Đảm bảo 1 class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến nó.
PROTOTYPE	Qui định loại của các đối tượng cần tạo bằng cách dùng một đối tượng mẫu, tạo mới nhờ vào sao chép đối tượng mẫu này.
BUILDER	Tách rời việc xây dựng (construction) một đối tượng phức tạp khỏi biểu diễn của nó sao cho cùng một tiến trình xây dựng có thể tạo được các biểu diễn khác nhau

2.3.2. Mẫu cấu trúc

Các mẫu cấu trúc có liên quan đến cách thức cấu tạo nên các lớp và các đối tượng để chúng có thể tạo thành các cấu trúc lớn và phức tạp hơn bằng cách sử dụng tính kế thừa để tạo ra các giao diện hoặc các triển khai.

Thay vì tạo ra các giao diện và các triển khai, mẫu cấu trúc mô tả các cách để chúng ta có thể phối hợp các đối tượng để nhận ra các chức năng mới. Sự linh hoạt được thêm

vào của việc kết hợp các đối tượng đến từ khả năng thay đổi sự kết hợp đó tại thời điểm chương trình đang chạy – run-time, một điều bất khả thi khi chúng ta kết hợp tĩnh các lớp.

Các mẫu cấu trúc của GoF bao gồm 7 mẫu, được giới thiệu sơ lược sau đây:

MẪU	MÔ TẢ NGẮN GỌN
ADAPTER	Do vấn đề tương thích, thay đổi interface của một lớp thành một interface khác phù hợp với yêu cầu từ phía sử dụng lớp.
BRIDGE	Tách rời sự trừu tượng của một vấn đề khỏi việc cài đặt; mục đích để cả hai bộ phận (trừu tượng và cài đặt) có thể thay đổi độc lập với nhau.
COMPOSITE	Tổ chức các đối tượng theo cấu trúc phân cấp dạng cây; Tất cả các đối tượng trong cấu trúc được thao tác theo một cách duy nhất như nhau. Tạo quan hệ thứ bậc bao gộp giữa các đối tượng.
DECORATOR	Thêm các trách nhiệm vào một đối tượng một cách tự động.
FACADE	Một lớp duy nhất đại diện toàn bộ cho một hệ thống con.
FLYWEIGHT	Một thể hiện nhỏ vừa đủ để sử dụng trong việc chia sẻ.
PROXY	Cung cấp một đối tượng đại diện cho một đối tượng khác để hỗ trợ hoặc kiểm soát quá trình truy xuất đối tượng đó.

2.3.3. Mẫu hành vi

Các mẫu hình vi có liên quan đến các giải thuật và việc phân công trách nhiệm giữa các đối tượng. Các mẫu hành vi mô tả không chỉ mô hình của các đối tượng hay các lớp mà còn mô hình sự tương tác giữa chúng. Chúng thể hiện các luồng điều khiển phức tạp mà chúng ta khó theo dõi trong quá trình chạy thành trở nên đơn giản hơn. Khi đó, chúng chuyển sự chú ý của chúng ta ra khỏi các dòng điều khiển rườm rà phức tạp, và ta chỉ cần tập trung vào cách mà các đối tượng liên kết với nhau.

Có ba cách mà các mẫu hành vi quản lý các đối tượng của mình và cách các đối tượng đó tương tác với nhau.

Cách thứ nhất, chúng sử dụng các tính kế thừa để phân phối các hành vi giữa các lớp. Có hai mẫu sử dụng cách này là Template Method và Interpreter. Cách thứ hai, chúng sử dụng sự kết hợp các đối tượng thay vì kế thừa. Một số mẫu mô tả làm cách nào một nhóm các đối tượng hợp tác với nhau để thực hiện một thao tác mà không đối tượng nào có thể tự mình thực hiện được, ví dụ như Mediator quản lý cách thức các đối tượng giao tiếp với nhau, Chain of Responsibility làm các đối tượng trong tập hợp liên kết lỏng lẻo với nhau, Observer định nghĩa và quản lý sự phụ thuộc giữa các đối tượng.

Cách thứ ba, các mẫu hành vi đóng gói các hành vi trong một đối tượng và ủy nhiệm các yêu cầu đến đối tượng đó. Mẫu Strategy là một ví dụ cụ thể, nó đóng gói một giải thuật vào một đối tượng, khi đó chúng ta dễ dàng lựa chọn và thay đổi giải thuật của một đối tượng khác sử dụng mẫu này.

Các mẫu hành vi của GoF bao gồm 11 mẫu, được giới thiệu sơ lược sau đây:

MẪU	MÔ TẢ NGẮN GỌN
CHAIN OF RESP	Các đối tượng nhận thông điệp được kết nối thành một chuỗi và thông điệp được chuyển dọc theo chuỗi này đến khi gặp được đối tượng xử lý nó. Tránh việc gắn kết cứng giữa phần tử gửi request với phần tử nhận và xử lý request bằng cách cho phép nhiều hơn một đối tượng có cơ hội xử lý request.
COMMAND	Mỗi yêu cầu (thực hiện một thao tác nào đó) được bao bọc thành một đối tượng. Các yêu cầu sẽ được lưu trữ và gửi đi như các đối tượng.
INTERPRETER	Hỗ trợ việc định nghĩa biểu diễn văn phạm và bộ thông dịch cho một ngôn ngữ.
ITERATOR	Truy xuất các phần tử của đối tượng dạng tập hợp tuần tự (list, array, ...) mà không phụ thuộc vào biểu diễn bên trong của các phần tử.

MEDIATOR	Định nghĩa một đối tượng để bao bọc việc giao tiếp giữa một số đối tượng với nhau.
MEMENTO	Lưu giữ và phục hồi trạng thái bên trong của một đối tượng mà vẫn không vi phạm việc tính đóng gói của đối tượng đó.
OBSERVER	Định nghĩa sự phụ thuộc một-nhiều giữa các đối tượng sao cho khi một đối tượng thay đổi trạng thái thì tất cả các đối tượng phụ thuộc nó cũng thay đổi theo.
STATE	Thay đổi hành vi của một đối tượng khi trạng thái bên trong của nó thay đổi.
STRATEGY	Bao bọc một họ các thuật toán bằng các lớp đối tượng để thuật toán có thể thay đổi độc lập đối với chương trình sử dụng thuật toán. Cung cấp một họ giải thuật cho phép client chọn lựa linh động một giải thuật cụ thể khi sử dụng
TEMPLATE METHOD	Định nghĩa phần khung của một thuật toán, tức là một thuật toán tổng quát gọi đến một số phương thức chưa được cài đặt trong lớp cơ sở; việc cài đặt các phương thức được ủy nhiệm cho các lớp kế thừa.
VISITOR	Cho phép định nghĩa thêm phép toán mới tác động lên các phần tử của một cấu trúc đối tượng mà không cần thay đổi các lớp định nghĩa cấu trúc đó.

2.4. Lợi ích khi sử dụng mẫu

a. Mẫu thiết kế tạo điều kiện cho việc tái sử dụng

Một trong những vấn đề gây nhiều tranh cãi nhất trong phát triển phần mềm là làm thế nào để kết hợp và tái sử dụng các thành phần có sẵn một cách thích hợp. Nhiều người trong số các bên liên quan (stakeholder) tham gia phát triển phần mềm nhận ra sự cần thiết và cơ hội để tái sử dụng mã, nhưng thử thách sẽ ngày càng lớn hơn để có

thể tái sử dụng trên quy mô lớn. Các mẫu thiết kế cung cấp cho chúng ta một cách để tái sử dụng các giải pháp thiết kế qua nhiều ứng dụng khác nhau.

b. Mẫu thiết kế làm quá trình thiết kế đơn giản hơn

Trong giai đoạn phân tích thiết kế phần mềm, nếu chúng ta nhận ra rằng một vấn đề có thể được giải quyết bằng cách áp dụng mẫu thiết kế thì giải pháp này sẽ có chất lượng cao hơn so với việc tìm một giải pháp giải quyết thông thường. Tuy nhiên, chúng ta cần có khả năng nhận diện khi nào sẽ sử dụng mẫu thiết kế nào và làm thế nào để áp dụng mẫu thiết kế đó vào trong ngữ cảnh sử dụng một cách phù hợp. Việc lạm dụng mẫu thiết kế mà không suy xét cẩn thận có thể dẫn đến các vấn đề nghiêm trọng thiết kế và hiện thực hóa phần mềm.

2.5. Cấu trúc trình bày mẫu

Các mẫu thiết kế trong báo cáo sẽ được trình bày theo cấu trúc dưới đây:

MỤC TRÌNH BÀY	NỘI DUNG TRÌNH BÀY
TÊN, PHÂN LOẠI, BÍ DANH	Mô tả ngắn gọn về mẫu
MỤC ĐÍCH, Ý ĐỊNH	Mô tả mẫu này làm được những gì
ĐỘNG LỰC SỬ DỤNG	Nêu một ví dụ về một vấn đề cần giải quyết có thể sử dụng mẫu này.
KHẢ NĂNG ỨNG DỤNG	Liệt kê một số tình huống cụ thể trong thiết kế phần mềm có thể áp dụng mẫu này
CẤU TRÚC MẪU	Mô tả mẫu bằng một sơ đồ UML bao gồm các lớp và đối tượng
CÁC THÀNH VIÊN	Trình bày ý nghĩa của các lớp/đối tượng tham gia vào mẫu thiết kế và trách nhiệm của chúng
SỰ CỘNG TÁC	Mô tả các thức các thành viên của mẫu tương tác với nhau như thế nào để thực hiện trách nhiệm của chúng

CÁC HỆ QUẢ MANG LẠI	Trình bày về ưu điểm khi sử dụng mẫu, các nhược điểm sẽ dẫn tới khi sử dụng mẫu.
CÁC CHÚ Ý KHI CÀI ĐẶT	Các chú ý đặc biệt khi cài đặt mẫu
VÍ DỤ THỰC TẾ	Nêu ra những ví dụ thực tế về các hệ thống (đã pt triển và đang chạy) có sử dụng mẫu này
CÁC MẪU LIÊN QUAN	Những mẫu nào có liên hệ đến mẫu này, những điểm quan trọng cần phân biệt, mẫu này có thể phối hợp với những mẫu nào.
MÃ NGUỒN MINH HỌA	Trình bày một ví dụ demo cụ thể về mẫu

2.6. Một số nguyên tắc

- Bên sử dụng (client) không bao giờ gọi trực tiếp các lớp hiện thực, thay vào đó, chúng gọi các lớp ảo hoặc các interface (sự trừu tượng hóa của các lớp đó).
- Các thay đổi trong tương lai không được ảnh hưởng đến hệ thống hiện tại.

3. Các mẫu Thiết kế Gang of Four

3.1. Mẫu Singleton

3.1.1. Tên, phân loại, bí danh

- Tên: Singleton Pattern.
- Bí danh: Không có
- Phân loại: Mẫu kiến tạo

3.1.2. Mục đích, ý định

Đảm bảo rằng mỗi lớp chỉ có một thể hiện duy nhất và cung cấp khả năng truy xuất toàn cục đến thể hiện này.

3.1.3. Động lực sử dụng

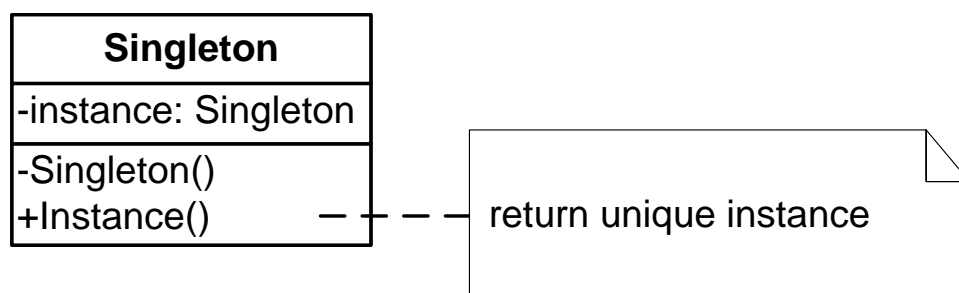
Khi ta cần giới hạn việc tạo ra các thể hiện từ một lớp. Đặc biệt là các đối tượng tốn nhiều tài nguyên hệ thống khi được tạo ra.

3.1.4. Khả năng ứng dụng

Ta có thể sử dụng mẫu Singleton khi:

- Một lớp có đúng một thể hiện, và thể hiện này có thể truy cập từ phía client từ một điểm truy cập đã biết trước.
- Khi thể hiện duy nhất nên được mở rộng bởi việc tạo lớp con, và các client có thể sử dụng một thể hiện đã được mở rộng mà không cần phải sửa đổi mã của chúng.

3.1.5. Cấu trúc



3.1.6. Các thành viên

Chỉ có một lớp Singleton tham gia vào mẫu. Lớp này phải định nghĩa hàm khởi tạo là private để ngăn việc khởi tạo thể hiện từ lớp này từ bên ngoài, đồng thời bắt buộc phải có hàm *getInstance* để trả một tham chiếu đến đối tượng sẽ được trả về có kiểu của lớp đó.

3.1.7. Sự cộng tác

Lớp nào muốn thể hiện mẫu Singleton cần định nghĩa một biến thành viên có kiểu là chính nó, đồng thời biến này cần được đặt modifier là static private.

Hàm khởi tạo của lớp Singleton được định nghĩa thành protected hoặc private để người dùng không thể tạo thể hiện của lớp trực tiếp từ bên ngoài.

Phương thức *getInstance* () dùng để khởi tạo đối tượng duy nhất, có modifier là public static. Client chỉ dùng phương thức *getInstance* () này để nhận được một đối tượng lớp Singleton.

3.1.8. Các hệ quả mang lại

Các lợi ích mà mẫu Singleton mang lại cho chúng ta:

a. *Kiểm soát việc truy cập đến thể hiện duy nhất*

Bởi vì lớp Singleton đóng gói đối tượng duy nhất của nó, nó có thể kiểm soát chặt chẽ việc làm thế nào và khi nào client truy cập tới nó.

b. *Giảm bớt namespace*

Mẫu Singleton là một sự cải tiến cho các biến toàn cục (global variable). Chúng ta sẽ tránh được việc làm cho namespace rối tinh lên với các biến toàn cục lưu trữ các đối tượng duy nhất mà trước đây chúng ta cần phải suy nghĩ cách đặt tên cho chúng.

c. *Cho phép việc tinh chế các hành động và sự biểu diễn của một lớp*

Một lớp Singleton có thể tạo ra lớp con, và việc cấu hình ứng dụng để sử dụng một thể hiện của lớp con tạo ra khá là dễ dàng. Ta có thể cấu hình ứng dụng với thể hiện của lớp mà chúng ta muốn tại thời điểm run-time.

d. Cho phép tạo một số lượng thể hiện nhất định

Mẫu Singleton cho phép ta tạo ra một số lượng nhất định các thể hiện mà không cần bắt buộc phải là một. Hơn thế nữa, ta cũng có thể dùng cách tiếp cận tương tự để điều khiển số lượng thể hiện có thể được tạo ra. Khi đó, chỉ có hành động thực hiện việc cho phép truy cập đến các thể hiện của lớp Singleton là cần được thay đổi.

3.1.9. Các chú ý liên quan đến cài đặt

Có nhiều cách để khởi tạo đối tượng sẽ được trả về trong C#:

a. Sử dụng Lazy initialization và property method:

Là cách thường dùng nhất.

- Cách tiếp cận này sẽ làm trì hoãn quá trình khởi tạo đối tượng cho đến khi nào client thực sự cần đến nó.
- Code minh họa:

```
class LazyInitializedSingleton
{
    private static LazyInitializedSingleton _instance;

    1 reference
    protected LazyInitializedSingleton()
    {
    }

    1 reference
    public static LazyInitializedSingleton Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new LazyInitializedSingleton();
            }
            return _instance;
        }
    }

    0 references
    public void Foo()
    {
        Console.WriteLine("This instance is fooing!");
    }
}
```

b. Sử dụng Static Initialization – Khởi tạo tĩnh:

- Cách tiếp cận ở trên không phù hợp khi đối tượng được tạo ra cần được sử dụng chung, khi đó if (_instance == null) sẽ gây ra lỗi khi lập trình đa luồng.
- Trong cách tiếp cận này, thể hiện _instance sẽ được tạo ra bất cứ khi nào một thành viên của lớp StaticInitializedSingleton được tham chiếu đến. Khi đó chúng ta để cho CLR khởi tạo đối tượng. Lớp tương ứng sẽ được đánh dấu là sealed để ngăn việc kế thừa, đồng thời biến _instance sẽ được đánh dấu là readonly, chỉ được gán giá trị lúc khởi tạo.
- Code minh họa:

```
sealed class StaticInitializedSingleton
{
    private static readonly StaticInitializedSingleton _instance
        = new StaticInitializedSingleton();

    1reference
    private StaticInitializedSingleton ()
    {
        Console.WriteLine("Initialize new instance using static initialization!");
    }

    // get object by using property
    0references
    public static StaticInitializedSingleton Instance
    {
        get
        {
            return _instance;
        }
    }

    0references
    public void Foo()
    {
        Console.WriteLine("This instance is fooing!");
    }
}
```

Trong nhiều trường hợp, cách tiếp cận này phù hợp hơn cách đầu tiên.

3.1.10. Các ví dụ hệ thống thực tế

Các hệ thống trong C# sử dụng mẫu Singleton

- c. Win32: GetProcessHeap
- d. MFC: AfxGetApp, AfxGetMainWnd
- e. .NET: System.AppDomain.CurrentDomain

3.1.11 Các mẫu liên quan

Abstract Factory: thường là Singleton để trả về các đối tượng factory duy nhất

Builder: dùng để xây dựng một đối tượng phức tạp, trong đó Singleton được dùng để tạo một đối tượng truy cập tổng quát (Director).

Prototype: dùng để sao chép một đối tượng, hoặc tạo ra một đối tượng khác từ Prototype của nó, trong đó Singleton được dùng để chắc chắn chỉ có một Prototype.

3.1.12 Mã nguồn minh họa

- Giới thiệu bài toán:

Giả sử chúng ta phát triển một ứng dụng sử dụng CSDL. Ta có thể kết nối đến CSDL thông qua một lớp DbConnection đã được các thành viên tham gia phát triển tạo ra. Vấn đề đặt ra là nếu mọi người đều có thể tạo ra các đối tượng DbConnection thì sẽ tốn rất nhiều chi phí bộ nhớ và làm giảm hiệu suất chương trình.

Ta sẽ áp dụng mẫu Singleton để chỉnh sửa lại lớp DbConnection này.

- Sơ đồ lớp:

- **Code:**
- **Lớp DbConnection – (Singleton):**

```
sealed class DbConnection
{
    // Static members are 'eagerly initialized', that is,
    // immediately when class is loaded for the first time.
    // .NET guarantees thread safety for static initialization
    private static readonly DbConnection _connection = new DbConnection();

    // Note: constructor is 'private'
    1 reference
    private DbConnection() { }

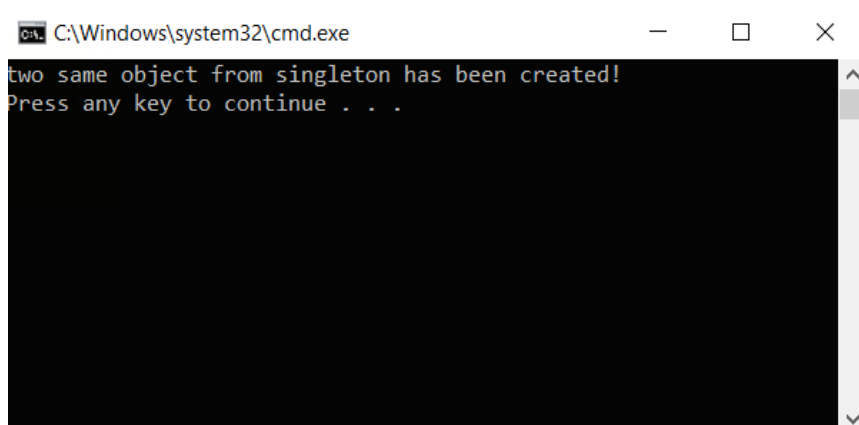
    // use property method instead of getInstance() method
    2 references
    public static DbConnection Connection
    {
        get
        {
            return _connection;
        }
    }
}
```

- **Lớp Program – (Client):**

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        DbConnection connection1 = DbConnection.GetInstance();
        DbConnection connection2 = DbConnection.GetInstance();

        if (connection1.Equals(connection2))
        {
            System.Console
                .WriteLine("two same object from singleton has been created!");
        }
    }
}
```

- **Kết quả khi chạy chương trình**



3.2 Mẫu Façade

3.2.1 Tên, phân loại, bí danh

- Tên đầy đủ: Façade Pattern
- Phân loại: Mẫu kiến trúc.
- Bí danh: Không có

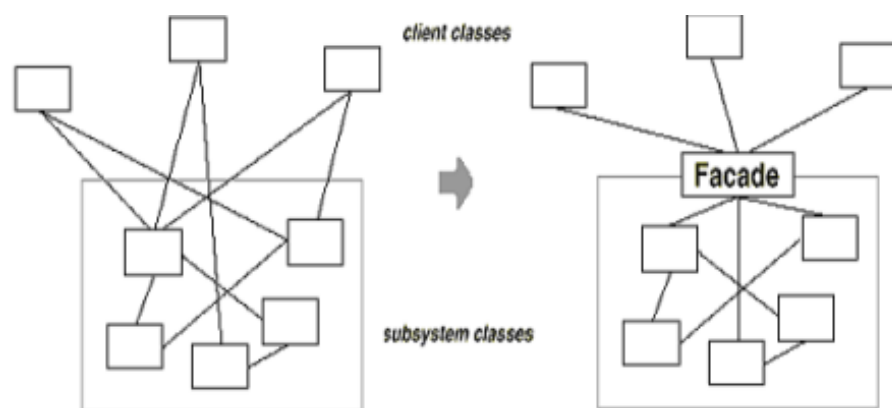
3.2.2 Mục đích, ý định

Façade Pattern là pattern cung cấp một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem).

Façade Pattern định nghĩa một giao diện ở một cấp độ cao hơn để giúp cho người dùng có thể dễ dàng sử dụng hệ thống con này vì chỉ cần giao tiếp với một giao diện chung duy nhất.

3.2.3 Động lực sử dụng

Cấu trúc một hệ thống phần mềm thành các mô đun (hệ thống con) nhỏ hơn làm giảm tính phức tạp vốn có. Khi đó một mục tiêu thiết kế chung được đặt ra là làm giảm đến mức nhỏ nhất việc giao tiếp và phụ thuộc giữa các mô đun đó. Một cách giúp chúng ta đạt được mục tiêu này là giới thiệu một đối tượng façade cung cấp một giao diện đơn giản và duy nhất thay cho một giao diện gồm nhiều các chức năng chung chung của một mô đun.



Giả sử ta có một môi trường lập trình cho phép các ứng dụng truy cập vào mô đun biên dịch (compiler) của nó. Mô đun này chứa các lớp chẳng hạn như Scanner, Parser, ProgramNode, BytecodeStream, và ProgramNodeBuilder. Một vài ứng dụng đặc biệt

có thể cần sử dụng các lớp bên trên này một cách trực tiếp để nâng cao hiệu suất làm việc. Nhưng hầu hết các client sử dụng mô đun compiler này không cần quan tâm chi tiết đến việc chuyển đổi dữ liệu (parsing) do lớp Parser đảm nhiệm, hay công việc tạo ra code (code generation) của ProgramNodeBuilder; chúng chỉ muốn biên dịch một vài đoạn mã mà thôi. Đối với các client này, các giao diện mạnh mẽ nhưng ở mức thấp của mô đun compiler chỉ làm phức tạp thêm các tác vụ của chúng.

Để cung cấp một giao diện mức cao hơn để ngăn các client khỏi các lớp có giao diện ở mức thấp. Modun compiler cung cấp một lớp Compiler, lớp này định nghĩa một giao diện duy nhất cho các chức năng của mô đun compiler. Lớp Compiler hoạt động như một façade: nó cho phép các client sử dụng một giao diện đơn giản để có thể sử dụng các chức năng của mô đun compiler. Nó kết dính các lớp chịu trách nhiệm cho các chức năng chính trong mô đun mà không làm ẩn hoàn toàn chúng đi. Khi đó, façade của compiler làm cho cuộc sống của các lập trình viên trở nên dễ dàng hơn mà không ẩn đi các chức năng mức thấp mạnh mẽ mà họ cần dùng.

3.2.4 Khả năng ứng dụng

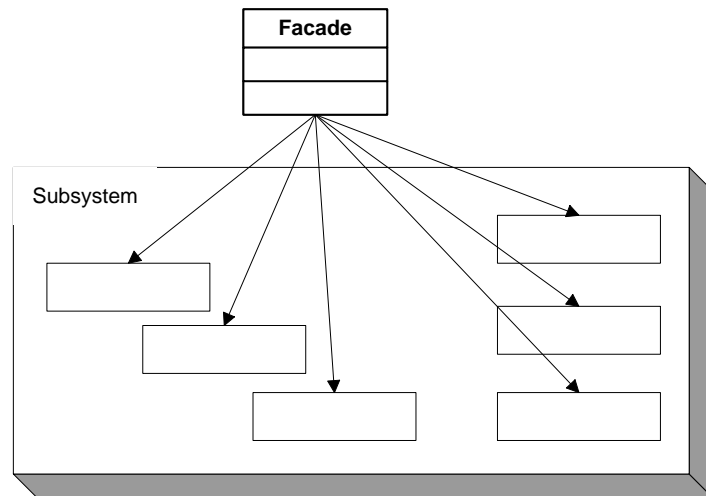
Khi ta muốn cung cấp một giao diện đơn giản hơn cho một hệ thống con phức tạp:

- Các hệ thống con càng phát triển theo thời gian thì chúng càng trở nên phức tạp
- Nhiều mẫu khi được áp dụng thường để lại kết quả là có nhiều và rất nhiều những lớp nhỏ hơn.
- Cung cấp một góc nhìn mặc định đơn giản hơn cho một hệ thống con và đủ dùng cho hầu hết các client. Chỉ có các client cần nhiều khả năng chỉnh sửa (customizability) hơn mới cần phải nhìn vượt ra khỏi façade.

Có quá nhiều phụ thuộc giữa các client và sự hiện thực các class của một trừu tượng nào đó.

- Façade giúp decouple hệ thống con ra khỏi client và các hệ thống con khác, nâng cấp khả năng độc lập và tính di động (portability) của hệ thống con.

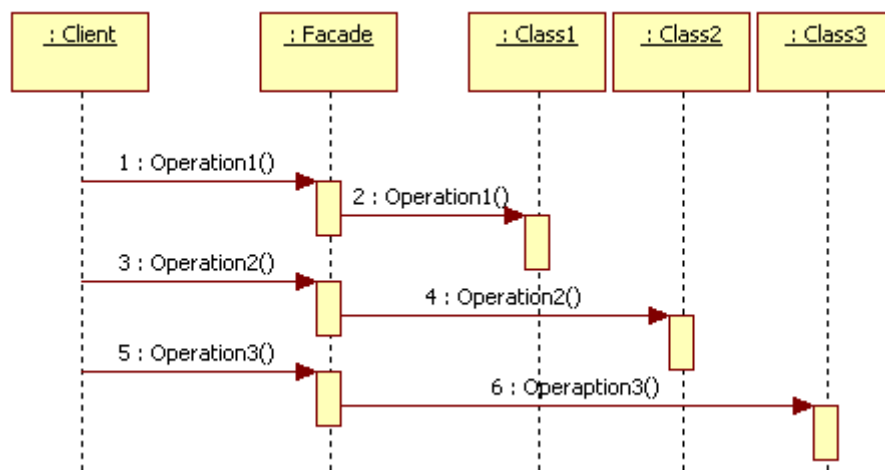
3.2.5 Cấu trúc



3.2.6 Các thành viên

- **Façade**
 - o Biết các lớp của hệ thống con nào chịu trách nhiệm cho một request.
 - o Ủy quyền các request của client đến các đối tượng của hệ thống con thích hợp sẽ chịu trách nhiệm xử lý request đó.
- **Subsystem classes:**
 - o Thực thi chức năng của hệ thống con.
 - o Xử lý công việc được gán bởi đối tượng Façade.
 - o Không cần biết gì về façade và không giữ tham chiếu đến nó

3.2.7 Sự cộng tác



3.2.8 Các hệ quả mang lại

Bảo vệ client khỏi các thành phần của hệ thống con, bằng cách đó nó làm giảm số lượng đối tượng mà client cần giải quyết và làm cho hệ thống con trở nên dễ sử dụng hơn.

Làm giảm sự liên kết giữa hệ thống con và client sử dụng nó:

- Chúng ta có thể sửa đổi các lớp trong hệ thống con mà không làm ảnh hưởng đến các client sử dụng nó.
- Giúp tách lớp một hệ thống và sự phụ thuộc giữa các đối tượng. Chúng có thể loại bỏ sự phức tạp hay phụ thuộc vòng tròn (circular dependencies). Đây là một hệ quả rất quan trọng bởi vì client và hệ thống con cần được hiện thực một cách độc lập.
- Không hề ngăn cản ứng dụng sử dụng các lớp của hệ thống con nếu chúng cần. Do đó ta có thể thoải mái lựa chọn giữa việc dùng façade (giao diện sử dụng trở nên đơn giản hơn) hoặc không dùng façade.

3.2.9 Các chú ý liên quan đến cài đặt

Giả sử các trường hợp sau có thể xảy ra khi hiện thực mẫu Façade:

1. *Làm giảm sự liên kết client-subsystem:*

Ta có thể làm cho sự liên kết (coupling) giữa các client và hệ thống con hơn nữa bằng cách khai báo lớp Façade là lớp ảo, các lớp Façade thực sự kế thừa từ lớp ảo này sẽ đại diện cho nhiều cách thể hiện khác nhau của cùng một hệ thống con. Sự liên kết trừu tượng này sẽ làm các client không biết chính xác được hiện thực nào của lớp con mà chúng đang sử dụng.

Một giải pháp thay thế khác cho việc tạo lớp con là cấu hình một đối tượng Façade với các đối tượng hệ thống con khác nhau. Để tùy chỉnh façade, ta chỉ đơn giản thay đổi một hay nhiều đối tượng hệ thống con của nó.

2. Các lớp của hệ thống con public hoặc private

Một hệ thống con khá giống với một lớp. Cả hai đều có một giao diện, và cả hai đều đóng gói một thứ bên trong chúng – các lớp thì đóng gói các trạng thái và các hoạt động, còn một hệ thống con thì đóng gói các lớp bên trong chúng. Cũng giống như việc nghĩ về các giao diện public hoặc private của một lớp, ta cũng có thể nghĩ về một hệ thống con với các giao diện public hoặc private.

Giao diện public của một hệ thống phụ bao gồm tất cả các lớp mà client có thể sử dụng, còn giao diện private chỉ được dành riêng cho những bộ phận có trách nhiệm mở rộng hệ thống con. Dĩ nhiên, lớp Façade cũng là một bộ phận của giao diện public, nhưng nó không phải là phần duy nhất, các lớp khác của hệ thống con cũng có thể thuộc giao diện public. Ví dụ, các lớp Parser và Scanner trong hệ thống con của trình biên dịch cũng là các thành phần thuộc về giao diện public của trình biên dịch.

Làm cho các lớp con của hệ thống con đôi khi cũng tốt, có một vài ngôn ngữ hỗ trợ việc này. Cả C++ và Smalltalk mặc định có một vài namespace toàn cục cho phép chúng ta sử dụng.

3.2.10 Các ví dụ về hệ thống thực tế

Các API mà các bên thứ 3 cung cấp cho chúng ta có thể được coi là một ứng dụng của mẫu Façade. Khi đó chúng ta không quan tâm đến hệ thống phức tạp như thế nào nhưng ta vẫn có thể sử dụng các chức năng của hệ thống đó thông qua một façade bao gồm các API mà nó cung cấp.

3.2.11 Các mẫu liên quan

Abstract Factory:

- Có thể được sử dụng cùng với Facade để cung cấp một giao diện cho việc tạo ra các đối tượng hệ thống con một cách độc lập với nhau. Abstract Factory cũng có thể được sử dụng như một giải pháp thay thế cho Facade nhằm ẩn đi các lớp phụ thuộc nền tảng (platform-specific).

Mediator:

- Tương tự như Façade, tuy nhiên mục đích chính của Mediator là trừu tượng hóa các giao tiếp truyền thông tùy ý giữa các đối tượng colleague. Giao tiếp của các đối tượng colleague phải thông qua Mediator.
- Tuy nhiên Façade chỉ đơn giản là trừu tượng hóa giao diện của một đối tượng hệ thống con và làm cho nó dễ xài hơn mà thôi. Façade cũng không định nghĩa các tính năng mới, và các lớp của hệ thống con cũng không biết gì về nó.

Singleton:

- Thông thường chỉ cần một đối tượng Façade là đủ. Do đó các đối tượng Façade thường là các Singleton.

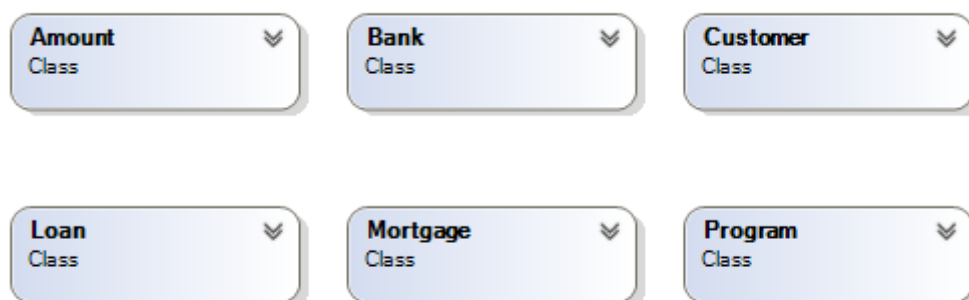
3.2.12 Mã nguồn minh họa

- Giới thiệu bài toán

Một hệ thống ngân hàng muốn cho khách hàng vay thế chấp thì cần phải thực hiện 3 thao tác chính: kiểm tra thông tin khách hàng, kiểm tra khoản vay của khách hàng có phù hợp hay không, kiểm tra xem khách hàng có đang nợ khoản tiền nào hay không? Nếu cả 3 điều kiện trên thỏa mãn thì mới cho phép khách hàng vay tiền.

Hiện tại hệ thống đang có 3 mô đun riêng rẽ thực hiện việc kiểm tra này, và hiện tại muốn kiểm tra một khách hàng, chúng ta cần phải trải qua 3 bước. Chúng ta có thể rút gọn thao tác này bằng cách sử dụng mẫu façade, sử dụng một hàm duy nhất để thực hiện việc kiểm tra.

- Sơ đồ lớp:



- **Code mẫu**

○ **Modul kiểm tra khoản nợ**

```
class Loan
{
    public bool HasNoBadLoans(Customer c)
    {
        Console.WriteLine("Check loans for " + c.Name);
        return true;
    }
}
```

○ **Modul kiểm tra khoản vay**

```
class Amount
{
    public bool isValid(int amount)
    {
        if (amount < 100000)
        {
            Console.WriteLine("valid amount of money");
            return true;
        }
        else return false;
    }
}
```

○ **Modul kiểm tra thông tin khách hàng**

```
class Bank
{
    public bool isValidCustomer(Customer c)
    {
        Console.WriteLine("Check bank for " + c.Name);
        return true;
    }
}
```

- **Lớp Façade cho thao tác kiểm tra khách hàng:**

```
class Mortgage
{
    private Bank _bank = new Bank();
    private Loan _loan = new Loan();
    private Amount _amount = new Amount();

    public bool IsEligible(Customer cust, int amount)
    {
        Console.WriteLine("{0} applies for {1:C} loan\n",
            cust.Name, amount);

        bool eligible = true;

        if (!_bank.isValidCustomer(cust))
        {
            eligible = false;
        }
        else if (!_loan.HasNoBadLoans(cust))
        {
            eligible = false;
        }
        else if (!_amount.isValid(amount))
        {
            eligible = false;
        }

        return eligible;
    }
}
```

- **Sử dụng chương trình:**

```
class Program
{
    static void Main(string[] args)
    {
        // Facade
        Mortgage mortgage = new Mortgage();

        // Evaluate mortgage eligibility for customer
        Customer customer = new Customer("Nguyen Van A");
        bool eligible = mortgage.IsEligible(customer, 125000);

        Console.WriteLine("\n" + customer.Name +
            " has been " + (eligible ? "Approved" : "Rejected"));

        // Wait for user
        Console.ReadKey();
    }
}
```

○ Kết quả khi chạy

```
file:///E:/Workspace/Visual Studio 2015/IteratorDPDemo/FacadeDemo/bin/Debug/FacadeDemo.EXE
Nguyen Van A applies for $5,000.00 loan
Check bank for Nguyen Van A
Check loans for Nguyen Van A
valid amount of money
Nguyen Van A has been Approved
```

3.3 Mẫu Proxy

3.3.1 Tên, phân loại, bí danh

- Tên chính thức: Proxy Pattern
- Phân loại: Structural Pattern
- Tên khác: Không có

3.3.2 Mục đích, ý định

Cung cấp một đối tượng đại diện cho một đối tượng khác để hỗ trợ hoặc kiểm soát quá trình truy xuất đối tượng đó.

Các mẫu Proxy được sử dụng khi bạn cần để biểu diễn một đối tượng phức tạp hay tốn thời gian để tạo ra bằng một thứ đơn giản hơn.

Nếu việc tạo ra một đối tượng mới mất nhiều thời gian hay tài nguyên máy tính, Proxy cho phép chúng ta trì hoãn quá trình tạo đối tượng cho đến khi chúng ta cần đối tượng thực sự.

Một mẫu Proxy thường có chung các phương thức giống như đối tượng mà nó đại diện, và một khi đối tượng được nạp vào bộ nhớ, các lời gọi hàm đến đối tượng thực sự này sẽ được chuyển tiếp qua Proxy

3.3.3 Động lực sử dụng

Một trong những lý do ta cần kiểm soát việc truy cập đến một đối tượng là để trì hoãn việc khởi tạo đầy đủ một đối tượng cho đến khi ta thực sự cần đến nó. Chẳng hạn như một chương trình chỉnh sửa tài liệu có thể nhúng các đối tượng đồ họa vào bên trong một file tài liệu. Các đối tượng đồ họa này có thể là một file hình ảnh. Nếu dung lượng của chúng khá lớn thì chương trình sẽ tiêu tốn một khoảng thời gian tương đối lâu để hiển thị chúng một cách đầy đủ tại vị trí chúng được chèn vào trên file tài liệu. Tuy nhiên việc mở file tài liệu cần phải diễn ra một cách nhanh chóng vì đa số người dùng không thích phải chờ đợi quá lâu. Do đó có một giải pháp cho vấn đề này là chúng ta sẽ tránh việc tạo ra các đối tượng đồ họa đó tại thời điểm file văn bản đó được mở ra. Tuy nhiên điều đó lại không thực sự cần thiết, bởi vì không phải tất cả các đối tượng đồ họa đều cần phải hiển thị trên tài liệu tại cùng một thời điểm.

Các ràng buộc nói trên có thể gợi ý chúng ta việc tạo ra các đối tượng theo *nhu cầu*, một hình ảnh trở nên nhìn thấy được (visible) khi chúng ta thực sự cần nó trở nên như thế. Nhưng chúng ta sẽ đặt cái gì vào vị trí của file ảnh đó khi chúng đang ẩn đi (invisible)? Và làm cách nào ta có thể giấu đi một sự thật rằng một hình ảnh được tạo ra theo nhu cầu như vậy sẽ không làm phức tạp hóa cách chúng ta hiện thực chương trình chỉnh sửa văn bản?

Một vài ví dụ khác về các tình huống ta sẽ sử dụng mẫu Proxy:

- Kết quả của một tính toán cần nhiều thời gian để hoàn thành, và bạn cần hiển thị các kết quả ngay lập tức trong khi quá trình tính toán vẫn đang tiếp tục.
- Đối tượng nằm ở trên một máy remote, và tải nó thông qua mạng có thể mất nhiều thời gian, đặc biệt trong những thời điểm mạng lag.
- Quyền truy cập đối tượng bị hạn chế, và proxy có thể xác nhận quyền truy cập của người dùng.

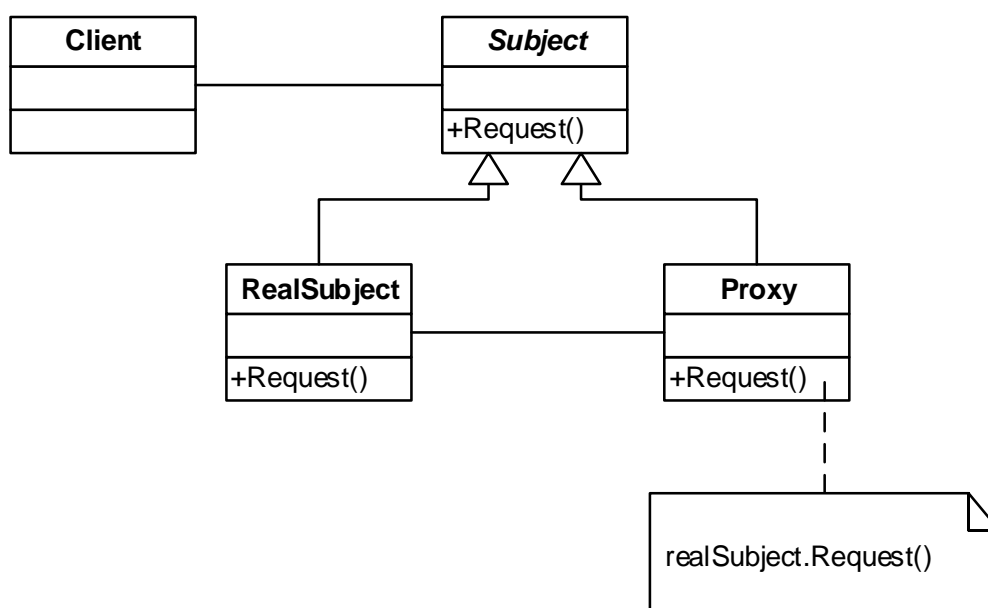
- Giả sử trong trường hợp chương trình cần nạp và hiển thị một file ảnh có dung lượng lớn. Khi chương trình khởi động, cần có một vài dấu hiệu cho người dùng thấy rằng bức ảnh đó sẽ được hiển thị trên màn hình và nằm đúng vị trí, nhưng thực sự thì việc hiển thị hình ảnh sẽ bị trì hoãn lại cho đến khi nào việc nạp bức ảnh hoàn tất.
- Trong các trình duyệt web hay xử lý văn bản, khi chúng cần đặt các nội dung text trước khi hiển thị các hình ảnh nhằm tăng tốc độ xử lý, khi đó các nội dung text sẽ nằm xung quanh các bức ảnh thậm chí trước khi cả tấm ảnh đó được hiển thị lên.

3.3.4 Khả năng ứng dụng

Các tình huống thường thấy khi cần áp dụng một mẫu proxy:

- Proxy ảo: Giữ chỗ cho một đối tượng mà đối tượng đó tốn tài nguyên khi được tạo ra. Đối tượng đó được tạo ra chỉ khi có yêu cầu từ client
- Proxy bảo vệ: Điều khiển truy cập đối một đối tượng RealObject nào đó. Việc truy cập đến RealObject sẽ được thông qua ProxyObject
- Liên kết thông minh: bổ sung hành động cho đối tượng được truy cập như đếm số lượng tham chiếu, tải đối tượng vào bộ nhớ khi nó được tham chiếu lần đầu tiên, kiểm tra đối tượng có đang được truy cập bởi một đối tượng khác hay không

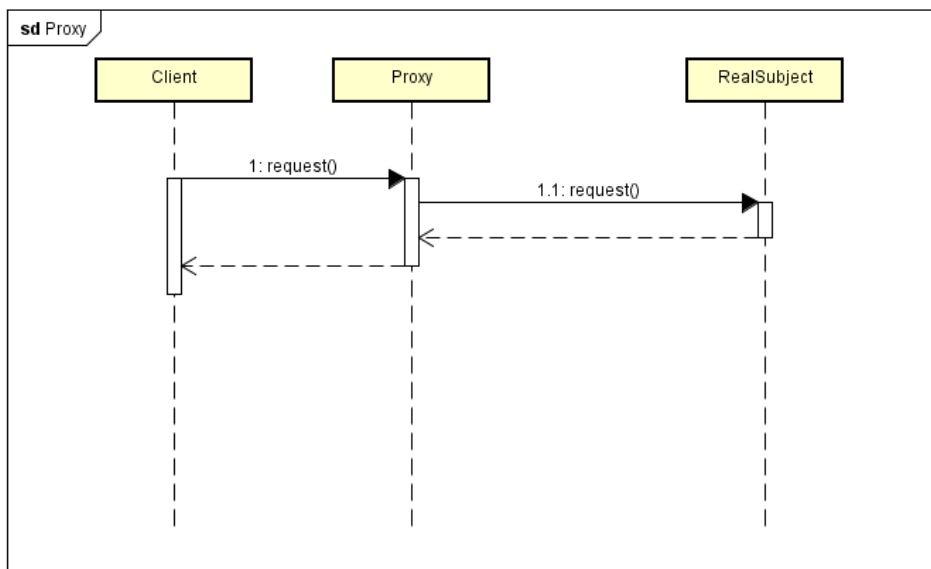
3.3.5 Cấu trúc



3.3.6 Các thành viên

- **Subject**
 - Định nghĩa một giao diện chung cho cả Real Subject và Proxy
- **Proxy**
 - Giữ một tham chiếu cho phép truy cập đến đối tượng thực sự (Real Subject).
 - Hiện thực hóa giao diện chung (Subject).
 - Giữ quyền điều khiển và quyền truy cập vào đối tượng Real Subject.
- **Real Subject**
 - Đối tượng thực sự được sử dụng thông qua Proxy

3.3.7 Sự cộng tác



Đối tượng Proxy sẽ là đối tượng trung gian và là một lớp bảo vệ cho đối tượng RealSubject, mỗi khi có một yêu cầu đến đối tượng RealSubject, yêu cầu nó sẽ đến Proxy, sau đó Proxy mới ủy nhiệm lại cho đối tượng RealSubject để xử lý. Cần chú ý là không phải tất cả các truy cập đều phải thông qua Proxy, tùy theo mục đích sử dụng Proxy đó của chúng ta.

3.3.8 Các hệ quả mang lại

Che giấu thông tin của các đối tượng thực sự đối với các client sử dụng chúng bằng cách cung cấp mức truy cập gián tiếp vào đối tượng đó và cơ chế tham chiếu vào đối tượng đích và chuyển tiếp các yêu cầu đến đối tượng đó.

Tối ưu hóa hoạt động của hệ thống nhờ cơ chế tải theo nhu cầu – demand loading.

Cả proxy và đối tượng đích đều kế thừa hoặc thực thi chung một lớp giao diện. Mã máy dịch cho lớp giao diện thường “nhẹ” hơn các lớp cụ thể và do đó có thể giảm được thời gian tải dữ liệu giữa server và client.

3.3.9 Các chú ý liên quan đến cài đặt

Mẫu Proxy cung cấp cùng một interface cho đối tượng thực và đối tượng Proxy.

Mẫu Decorator và mẫu Proxy có mục đích khác nhau nhưng cấu trúc giống nhau.

Đối tượng Proxy luôn luôn giữ một tham chiếu đến đối tượng thực sự.

3.3.10 Các ví dụ về hệ thống thực tế

Các hệ thống cần truy cập thông tin từ các hệ thống khác đa phần sử dụng kiến trúc Proxy.

Các máy khách sử dụng các dịch vụ WCF phụ thuộc vào các đối tượng proxy được WCF tự động tạo ra.

Máy ATM có một proxy ảo lưu các thông tin ngân hàng được dùng khi xác nhận thẻ tín dụng...

3.3.11 Các mẫu liên quan

Mẫu Adapter:

- Adapter hiện thực một giao diện khác cho đối tượng mà nó tham chiếu tới (đối tượng cần sự tương thích).
- Proxy hiện thực một giao diện tương tự như chủ thể của mà nó giữ tham chiếu.

Mẫu Decorator:

- Một hiện thực của decorator có thể gần giống như các proxy, tuy nhiên một decorator sẽ thêm một trách nhiệm mới cho đối tượng được tham chiếu.
- Trong khi đó, một proxy sẽ kiểm soát các truy cập vào đối tượng mà nó đang giữ tham chiếu.

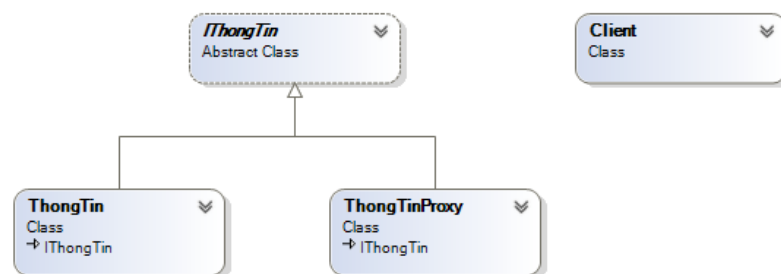
3.3.12 Mã nguồn minh họa

- Bài toán

Người dùng A muốn truy cập để xem thông tin bảo mật của người dùng B. Nhưng để xem được thì cần phải có quyền hạn cho chức năng này. Người dùng A cần có quyền hạn của Admin để truy cập những thông tin đó.

Yêu cầu: Xuất thông tin người dùng B ra màn hình khi người dùng A đủ quyền hạn truy cập (quyền Admin)

- Sơ đồ lớp



- Code mẫu

- o **Lớp trừu tượng ThongTin**, định nghĩa một giao diện cho cả subject và proxy

```
abstract class IThongTin
{
    public abstract string getName();
}
```

- **Lớp ThôngTinProxy**, là một proxy bảo vệ cung cấp một sự kiểm soát cho subject ThôngTin

```
class ThôngTinProxy : IThôngTin
{
    private bool is_Admin = false;

    public ThôngTinProxy(bool _is_admin)
    {
        this.is_Admin = _is_admin;
    }

    public override string getName()
    {
        if (this.is_Admin == true)
        {
            ThôngTin a = new ThôngTin("Linh");
            return a.getName();
        }
        return "";
    }
}
```

- **Lớp ThôngTin**: là subject cần được sự bảo vệ của một đối tượng proxy ThôngTinProxy

```
class ThôngTin : IThôngTin
{
    private string name;

    public ThôngTin()
    {
        this.name = "Nam";
    }

    public ThôngTin(string _name)
    {
        this.name = _name;
    }

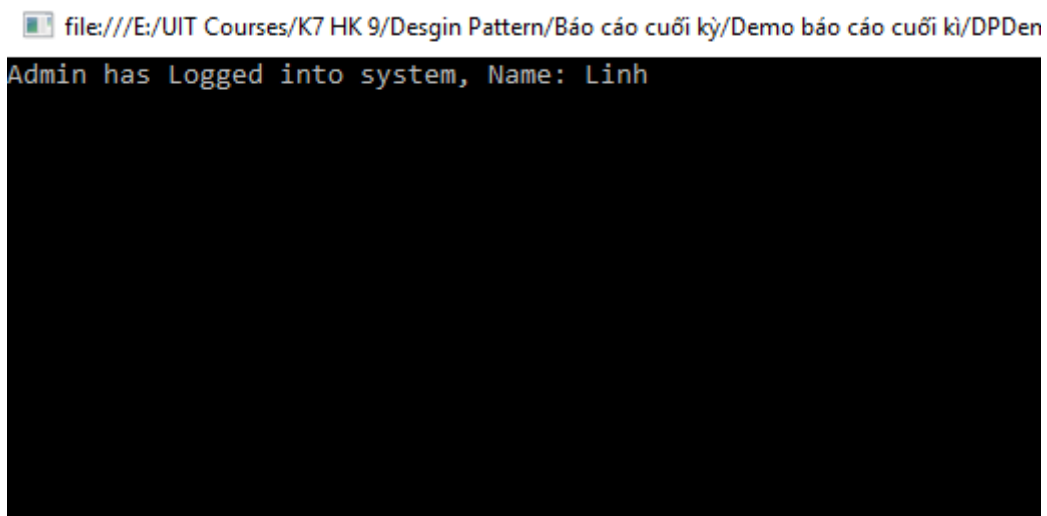
    public override string getName()
    {
        return this.name;
    }
}
```

○ Lớp thực thi chương trình

```
class Client
{
    static void Main(string[] args)
    {
        ThôngTinProxy B = new ThôngTinProxy(true);
        Console.WriteLine("Admin has Logged into system, Name: " + B.GetName());

        Console.ReadLine();
    }
}
```

○ Kết quả thực hiện chương trình



3.4 Mẫu Iterator

3.4.1 Tên, phân loại, bí danh

- Tên: Iterator pattern
- Phân loại: Mẫu hành vi
- Bí danh: không có.

3.4.2 Mục đích, ý định

Cung cấp một cách thức truy xuất tuần tự đến các phần tử của một đối tượng tập hợp mà không cần phải phơi bày cấu trúc bên trong của đối tượng này.

3.4.3 Động lực sử dụng

Một đối tượng tập hợp (aggregate object) là một đối tượng thể hiện một nhóm các đối tượng có liên quan với nhau. Ví dụ trong C#, các đối tượng của các lớp Collection

như ArrayList, Stack, ... là một đối tượng tập hợp. Một operation - hành vi thường thấy trong các đối tượng như thế này là lặp (iteration) hay xử lý tuần tự (sequential processing) trên mỗi phần tử.

Một phương pháp cổ điển mà các lớp tập hợp có thể hỗ trợ lặp là định nghĩa các hành vi như *getFirst* và *getNext*. Tuy nhiên khi sử dụng các đối tượng từ các lớp này thì client phải hard-code kiểu của lớp đó vào code, gây ra hiện tượng tightly-coupled, mặt khác client code sẽ bị phụ thuộc vào thuật toán lặp của lớp đối tượng, khi thuật toán này thay đổi thì code sử dụng đối tượng đó cũng phải đổi theo.

=> Mẫu Iterator có thể giúp chúng ta xử lý cả hai vấn đề này, áp dụng mẫu này giúp code bên phía client độc lập với lớp tập hợp, đồng thời giúp client sử dụng các đối tượng tập hợp mà không cần quan tâm đến chi tiết cách hiện thực iteration của lớp tập hợp.

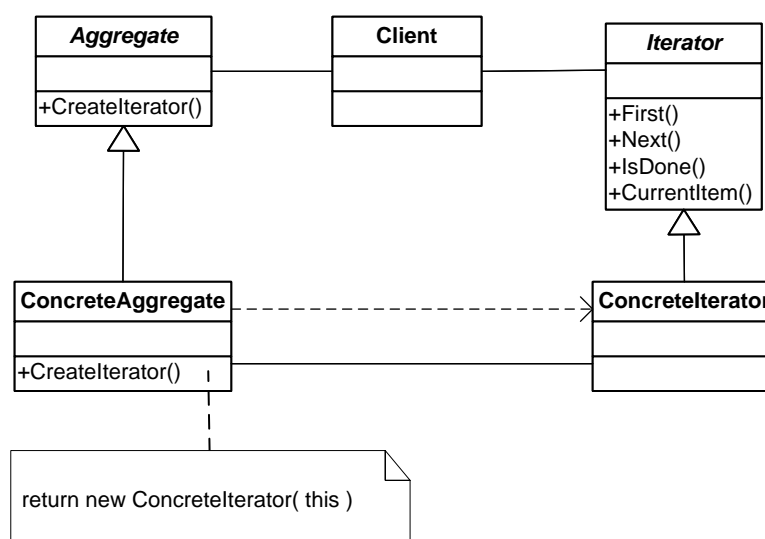
3.4.4 Khả năng ứng dụng

Khi cần truy xuất nội dung của một đối tượng tập hợp mà chúng ta không muốn biết về cấu trúc bên trong của nó.

Để hỗ trợ việc đa dịch chuyển (multiple traversals) trên các đối tượng tập hợp.

Để cung cấp một giao diện thống nhất cho việc di chuyển trên các đối tượng tập hợp (hay nói cách khác là hỗ trợ tính lặp đa hình - polymorphic iteration).

3.4.5 Cấu trúc



3.4.6 Các thành viên

- **Iterator:**
 - Định nghĩa một giao diện cho việc truy xuất và đi qua tập hợp các phần tử.
- **ConcreteIterator:**
 - Hiện thực giao diện **Iterator**, định nghĩa một thuật toán để lặp và di chuyển qua tập hợp.
 - Giữ thông tin về phần tử hiện tại trong quá trình di chuyển qua tập hợp.
- **Aggregate:**
 - Định nghĩa một giao diện cho việc tạo ra đối tượng **Iterator**,
- **ConcreteAggregate:**
 - Các lớp tập hợp hiện thực giao diện Aggregate, khi đó lớp đó sẽ định nghĩa các phương thức cần thiết để trả về một thể hiện của lớp **ConcreteIterator** cho client sử dụng.

3.4.7 Sự cộng tác

Một ConcreteIterator theo dấu đối tượng hiện tại trong tập hợp và tính toán đối tượng tiếp theo trong quá trình di chuyển.

3.4.8 Các hệ quả mang lại

Sử dụng mẫu này giúp chúng ta truy xuất tuần tự đến các phần tử của một đối tượng tập hợp mà không quan tâm đến hiện thực của lớp tập hợp tạo ra đối tượng đó, đồng thời độc lập với cách hiện thực thuật toán lặp của nó.

Client độc lập (loosely-coupling) với kiểu của đối tượng tập hợp và thuật toán lặp sử dụng trong kiểu tập hợp đó. Ta có thể thay đổi mã nguồn của chúng mà không làm ảnh hưởng đến client.

Hỗ trợ nhiều cách di chuyển trong một tập hợp:

- Các tập hợp phức tạp có thể được lướt qua bằng nhiều cách, ví dụ như theo thứ tự (inorder) hoặc preorder.
- Dễ dàng thay đổi thuật toán di chuyển:

- Chỉ cần thay đổi thể hiện iterator với một thể hiện khác.
- Hoặc định nghĩa một lớp con của Iterator để hỗ trợ một các di chuyển khác.
- Các Iterator làm đơn giản hóa giao diện Aggregate. Nếu không có các Iterator, nhiều giao diện Aggregate sẽ có phần giao diện cho việc iteration và traversal giống nhau.

3.4.9 Các chú ý liên quan đến cài đặt

Có hai kiểu iterator: internal iterator và external iterator. Trong C#, ta có thể dùng external iterator một cách dễ dàng.

- Các thuật toán di chuyển qua các phần tử trong tập hợp có thể được định nghĩa bên trong lớp tập hợp hoặc bên trong bản thân Iterator, thông thường là trong iterator, khi đó iterator là một lớp inner trong lớp tập hợp.
- Robust Iterator:
 - Khi sử dụng quá trình lặp bắt đầu, một robust iterator cho phép ta có thể thêm hay xóa các phần tử trong tập hợp mà không tạo ra một bản copy của tập hợp hiện tại; đồng thời không có phần tử nào bị bỏ qua hay bị lặp lại 2 lần.
 - Nếu ta ko cần Robust Iterator thì tức là tập hợp đó không thể chỉnh sửa được và ta cần báo điều này cho client biết. ta có thể trả về giá trị false hay quăng exception nếu client cố gắng thêm/xóa một phần tử của tập hợp.
 - Một giải pháp thay thế là ta định nghĩa thêm các hàm trả về giá trị Boolean bao gồm *remove*, *insertAfter*, *insertBefore* vào bên trong Iterator để ta có thể thêm/xóa các phần tử của tập hợp. Khi đó các hành vi này nên được thêm vào giao diện Iterator để đảm bảo tính nhất quán cho toàn bộ các ConcreteIterator.

3.4.10 Các ví dụ về hệ thống thực tế

C# hỗ trợ việc lặp và di chuyển qua các phần tử trong một đối tượng tập hợp (điều kiện là lớp của đối tượng đó phải hiện thực giao diện IEnumerable).

3.4.11 Các mẫu liên quan

Composite: Các Iterator thường được áp dụng trong các cấu trúc đệ quy chẳng hạn như Composite.

Factory Method: Các iterator đa hình phụ thuộc vào các factory method để khởi tạo lớp đối tượng Iterator phù hợp.

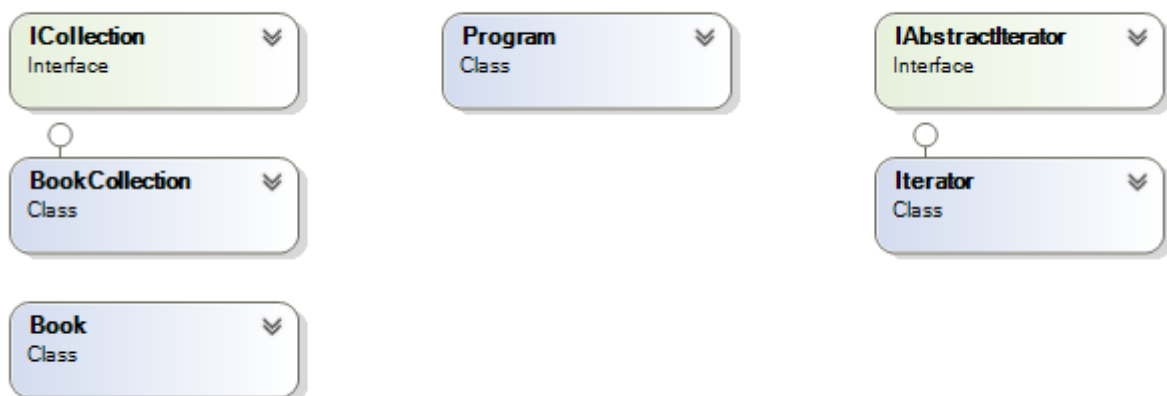
Memento thường được sử dụng chung với mẫu Iterator. Một iterator có thể sử dụng một memento để chụp (capture) lại trạng thái lặp hiện tại và lưu trữ memento đó một cách cục bộ.

3.4.12 Mã nguồn minh họa

- Giới thiệu bài toán

Hiệu sách ABC muốn sử dụng một chương trình để quản lý các cuốn sách hiện có trong cửa hàng. Chương trình này cho phép người quản lý có thể thêm các cuốn sách mới và duyệt qua danh sách các cuốn sách đã có.

- Sơ đồ lớp



- **Code mẫu**

- **Lớp Book**, thể hiện các cuốn sách của cửa hàng.

```
class Book
{
    private String _name;

    public Book(String name)
    {
        this._name = name;
    }

    public String Name
    {
        get { return _name; }
    }
}
```

- **Giao diện ICollection**, định nghĩa một giao diện chung cho các tập hợp

```
interface ICollection
{
    Iterator CreateIterator();
}
```

- **Lớp BookCollection**, biểu diễn các tập hợp các đối tượng sách

```
class BookCollection : ICollection
{
    private ArrayList _items = new ArrayList();

    public Iterator CreateIterator()
    {
        return new Iterator(this);
    }

    // Gets number of book
    public int Count
    {
        get { return _items.Count; }
    }

    // Indexer
    public object this[int index]
    {
        get { return _items[index]; }
        set { _items.Add(value); }
    }
}
```

- **Giao diện IAbstractIterator**, định nghĩa một giao diện chung cho các đối tượng iterator sẽ làm việc trên các tập hợp, ở đây ta sẽ có một iterator để làm việc trên tập hợp các cuốn sách BookCollection

```
interface IAbstractIterator
{
    Book First();
    Book Next();
    bool IsDone { get; }
    Book CurrentItem { get; }
}
```

- **Lớp Iterator:** hiện thực giao diện **IAbstractIterator** để làm việc với các đối tượng trong tập hợp BookCollection

```
class Iterator : IAbstractIterator
{
    private BookCollection _collection;
    private int _current = 0;
    private int _step = 1;

    // Constructor
    public Iterator(BookCollection collection)
    {
        this._collection = collection;
    }

    // Gets first item
    public Book First()
    {
        _current = 0;
        return _collection[_current] as Book;
    }

    // Gets next item
    public Book Next()
    {
        _current += _step;
        if (!IsDone)
            return _collection[_current] as Book;
        else
            return null;
    }

    // Gets or sets stepsize
    public int Step
    {
        get { return _step; }
        set { _step = value; }
    }

    // Gets current iterator item
    public Book CurrentItem
    {
        get { return _collection[_current] as Book; }
    }

    // Gets whether iteration is complete
    public bool IsDone
    {
        get { return _current >= _collection.Count; }
    }
}
```

- **Lớp Program,** nơi người dùng có thể thực hiện các thao tác thêm và duyệt qua danh sách các cuốn sách.

```

class Program
{
    static void Main(string[] args)
    {
        BookCollection collection = new BookCollection();
        collection[0] = new Book("Harry Potter");
        collection[1] = new Book("Lord of the Ring");
        collection[2] = new Book("Dac Nhan Tam");

        // Create iterator
        Iterator iterator = collection.CreateIterator();

        // Skip every other item
        iterator.Step = 1;

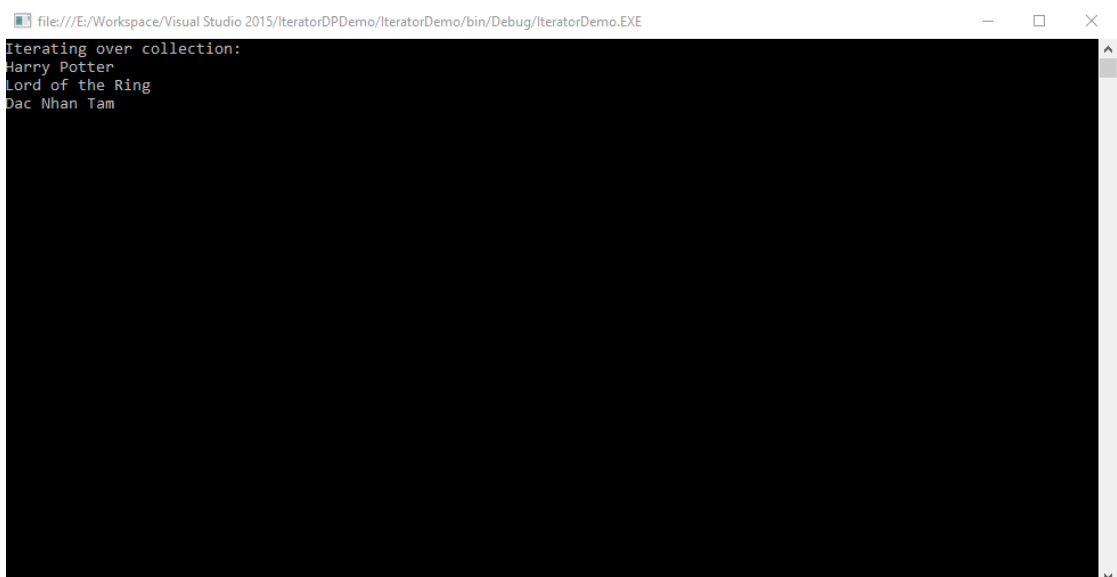
        System.Console.WriteLine("Iterating over collection:");

        for (Book book = iterator.First(); !iterator.IsDone; book = iterator.Next())
        {
            System.Console.WriteLine(book.Name);
        }

        // Wait for user
        System.Console.ReadKey();
    }
}

```

○ Kết quả khi thực hiện chương trình



```

file:///E:/Workspace/Visual Studio 2015/IteratorDPDemo/IteratorDemo/bin/Debug/IteratorDemo.EXE
Iterating over collection:
Harry Potter
Lord of the Ring
Dac Nhan Tam

```

3.5 Mẫu Observer

3.5.1 Tên, phân loại, bí danh

- Tên chính thức: Observer Pattern
- Phân loại: Structural Pattern
- Tên khác: Dependents, Publish/Subscribe hoặc Source/Listener

3.5.2 Mục đích, ý định

Định nghĩa một sự phụ thuộc 1 – nhiều giữa các đối tượng để khi có một tượng thay đổi trạng thái thì tất cả những đối tượng phụ thuộc của nó được thông báo và cập nhật tự một cách tự động

Khi đối tượng này gửi thông điệp thì các đối tượng đăng ký lắng nghe thông điệp sẽ phản ứng lại với thông điệp đó. Đối tượng gửi thông điệp sẽ biết được nó sẽ gửi cho ai và đối tượng nhận thông điệp sẽ không cần biết ai gửi thông điệp đó.

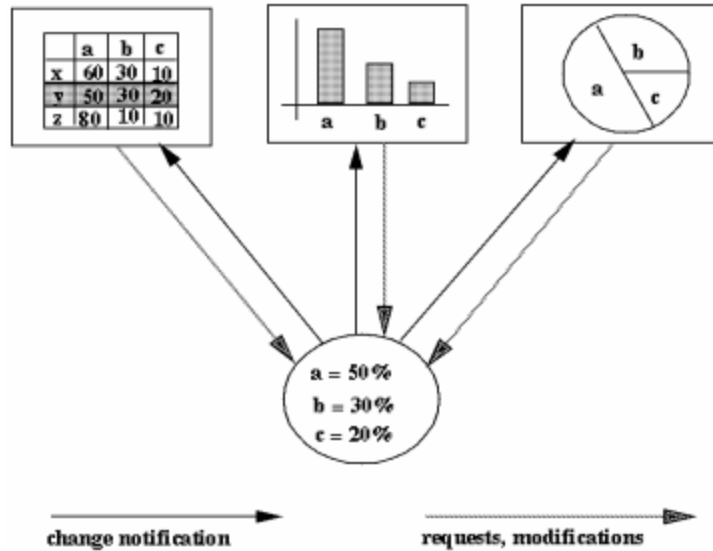
3.5.3 Động lực sử dụng

Một trong những thách thức của việc phân chia hệ thống thành một tập hợp các lớp cùng làm việc với nhau là làm thế nào để quản lý sự chặt chẽ giữa các đối tượng có liên quan của chúng. Chúng ta không muốn các lớp bị phụ thuộc vào nhau, bởi vì khi đó ta rất khó tái sử dụng lại chúng.

Mẫu Observer giúp chúng ta định nghĩa làm thế nào để thiết lập các sự liên hệ giữa các đối tượng đó. Các đối tượng chính ở đây là Subject và Observer. Một subject có thể có nhiều observer phụ thuộc. Tất cả các observer được thông báo khi có bất kì sự thay đổi nội tại nào của subject. Để phản ứng lại, mỗi observer sẽ truy vấn đến subject để đồng bộ hóa trạng thái của nó với trạng thái của subject.

Kiểu tương tác này cũng được biết dưới tên gọi publish-subscribe. Đối tượng subject là người phát ra các thông báo (publisher). Nó gửi các thông báo đó đến các đối tượng observer của nó (subscriber) mà không cần biết chính xác các observer đó là ai. Bất kỳ observer nào cũng có thể đăng kí để nhận được các thông báo trên.

Ví dụ, trong một chương trình bảng tính, một nguồn dữ liệu có thể có nhiều khung nhìn khác nhau, khi một nội dung bên trong nó bị thay đổi, các khung nhìn hiển thị dữ liệu phụ thuộc vào nó cũng sẽ thay đổi theo tương ứng.

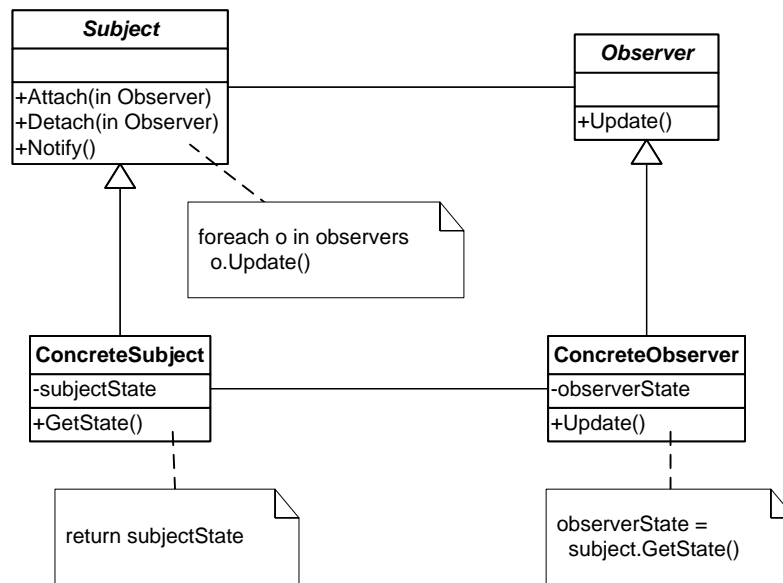


3.5.4 Khả năng ứng dụng

Ta có thể sử dụng mẫu này trong các trường hợp sau:

- Khi một sự trừu tượng hóa có hai khía cạnh, cái này phụ thuộc vào cái kia, đóng gói các khía cạnh này trong các đối tượng riêng biệt giúp chúng ta có thể thay đổi và sử dụng chúng một cách độc lập.
- Khi một sự thay đổi của một đối tượng yêu cầu các đối tượng khác phải thay đổi theo, và ta không biết chính xác số lượng các đối tượng cần thay đổi.
- Khi một đối tượng cần khả năng thông báo tới các đối tượng khác mà không cần biết các đối tượng đó là gì. Nói cách khác, ta không muốn các đối tượng đó liên kết chặt chẽ với nhau (tightly coupled).

3.5.5 Cấu trúc



3.5.6 Các thành viên

- **Subject**
 - Lớp giao diện chung cho các đối tượng. Bao gồm các phương thức chính: Thêm, xóa và thông báo các Observer.
- **ConcreteSubject**
 - Triển khai giao diện lớp Subject. Lưu trữ trạng thái mà các đối tượng Observer quan tâm. Mỗi khi thay đổi trạng thái thì thông báo sẽ được gửi đến các Observer đã được đăng ký trước đó.
- **Observer**
 - Là một giao diện với phương thức chính là `update()`. Phương thức này truy cập đối tượng Subject và cập nhật Observer khi trạng thái của Subject thay đổi.
- **ConcreteObserver**
 - Hiện thực hóa giao diện Observer. Là đối tượng sẽ nhận được thông báo khi trạng thái của Subject thay đổi.

3.5.7 Sự cộng tác

Các liên kết giữa các Subject và Observer là trừu tượng vì các đối tượng không biết những lớp cụ thể của bất kỳ Observer nào.

Các request không cần phải chỉ ra người nhận, mà các thông báo sẽ được gửi đi tự động đến tất cả các đối tượng đã đăng ký với nó.

Các đối tượng không quan đến việc có bao nhiêu đối tượng khác tồn tại. Nhiệm vụ của nó chỉ là thông báo đến các đối tượng đang quan sát nó. Điều này giúp ta dễ dàng thêm và loại bỏ các Observer bất cứ lúc nào.

3.5.8 Các hệ quả mang lại

Mẫu Observer cho phép chúng ta thay đổi các đối tượng subject và các đối tượng observer một cách độc lập. Ta có thể tái sử dụng các subject mà không cần phải sử dụng lại các observer, và ngược lại. Điều này cho phép chúng ta thêm các observer mới mà không cần phải sửa đổi các subject hay các observer khác.

Các lợi ích khác của mẫu Observer bao gồm:

1. Trừu tượng hóa sự liên kết giữa đối tượng Subject và đối tượng Observer:

Tất cả những gì Subject quan tâm là nó chứa một danh sách các đối tượng Observer, mỗi đối tượng phù hợp với giao diện đơn giản của lớp Observer thuần ảo. Subject không cần quan tâm không cần quan tâm đến bất kỳ lớp con Observer cụ thể nào. Điều đó làm sự kết dính giữa các đối tượng subject và observer được trừu tượng hóa và không đáng kể.

Bởi vì Subject và Observer không kết dính với nhau, chúng có thể thuộc về các layer trừu tượng khác nhau trong một hệ thống. Một đối tượng subject mức thấp có thể liên lạc và thông báo tới các đối tượng observer ở mức cao hơn, do đó giúp cho sự phân lớp hệ thống trở nên nguyên vẹn. Nếu Subject và Observer bị gộp lại trong đối tượng thì đối tượng đó sẽ thuộc về hai layer khác nhau và vi phạm nguyên tắc phân lớp, hoặc đối tượng đó bị bắt buộc phải lựa chọn xem nó sẽ thuộc về layer nào trong hai layer của hệ thống (một sự thỏa hiệp khi trừu tượng hóa các layer).

2. Hỗ trợ giao tiếp broadcast:

Không giống như yêu cầu bình thường, thông báo mà subject gửi đi không cần phải chỉ rõ người nhận. Thông báo này được phát sóng (broadcast) một cách tự động tới tất cả các đối tượng quan tâm đã đăng kí theo dõi subject. Subject không cần quan tâm có bao nhiêu đối tượng như thế đang tồn tại; trách nhiệm duy nhất của nó là thông báo tới các observer của nó, thế là xong. Vậy là ta được quyền tự do thêm và xóa bất kỳ observer nào tại bất cứ lúc nào ta muốn. Các observer sẽ tự quyết định xem chúng sẽ xử lý hay bỏ qua thông báo mà subject đang spam đi khắp nơi.

3. Các cập nhật không mong muốn:

Bởi vì các observer không biết chút gì về sự hiện diện của các đối tượng observer khác, chúng có thể mù quáng làm tăng thêm chi phí cho hệ thống khi chúng thay đổi subject. Khi một subject cập nhật trạng thái của chúng có thể gây ra một loạt các thao tác cập nhật do các đối tượng observer đăng kí tới subject đó thực hiện, chưa kể đến các đối tượng phụ thuộc vào các observer đó. Hơn thế nữa, nếu các sự phụ thuộc này không được định nghĩa chính xác hay được bảo trì sẽ dẫn đến các cập nhật không mong muốn và sẽ gây khó khăn cho quá trình theo dõi.

3.5.9 Các chú ý liên quan đến cài đặt

Trong trường hợp các observer quan tâm nhiều subject, trong quá trình thông báo các subject truyền trực tiếp đến các observer để chính observer quy định hành vi thay đổi.

Các subject sẽ quy định điều kiện thông báo đến các observer, lược bỏ những thông báo không cần thiết, tăng hiệu năng, trong trường hợp này, các observer sẽ thông báo cho các subject khi nào là cần thiết.

3.5.10 Các ví dụ về hệ thống thực tế

Mẫu thiết kế Observer đã được tích hợp vào package `java.util` trong Java API.

Các hệ thống thông báo tin nhắn/email tự động của các website cho phép chủ các website gửi các tin nhắn đến các tài khoản có đăng kí theo dõi.

3.5.11 Các mẫu liên quan

Mẫu **Mediator**: bằng cách đóng gói những cập nhật ngữ cảnh phức tạp, Observable hoạt động như đối tượng Mediator giữa các đối tượng và các Observer.

Mẫu **Singleton**: các Observable có thể là Singleton để nó trở nên duy nhất và được truy cập toàn cục.

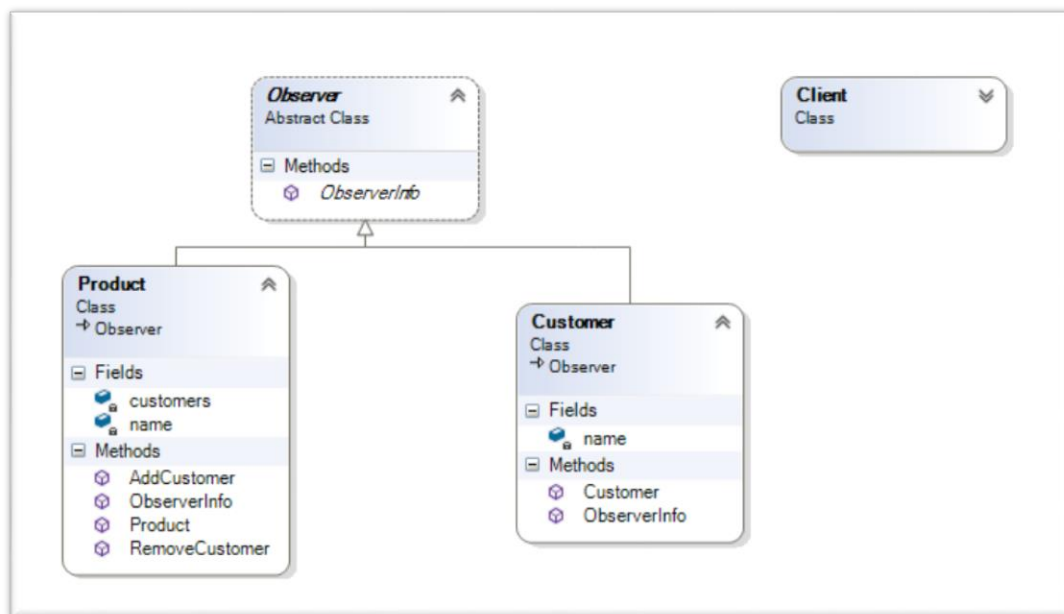
3.5.12 Mã nguồn minh họa

- Bài toán

Một sản phẩm mới chuẩn bị được tung ra thị trường bằng cách thông báo thông tin sản phẩm đến khách hàng. Hoặc khách hàng có mong muốn nhận bất kỳ thông báo nào từ sản phẩm mỗi khi có cập nhật.

Yêu cầu: Xây dựng hệ thống thông báo tự động thông tin sản phẩm đến những khách hàng đã đăng ký nhận tin từ hệ thống.

- Sơ đồ lớp



- **Code mẫu**

○ **Lớp ảo Observer:**

```
abstract class Observer
{
    public abstract void ObserverInfo();
}
```

○ **Lớp Product**

```
class Product : Observer
{
    private string name;

    List<Customer> customers = new List<Customer>();

    public Product(string name)
    {
        this.name = name;
    }

    public override void ObserverInfo()
    {
        foreach (Customer item in customers)
        {
            if (item != null)
            {
                item.ObserverInfo();
            }
        }
    }

    public void AddCustomer(Customer cus) {
        if (cus != null)
        {
            this.customers.Add(cus);
        }
    }

    public void RemoveCustomer(Customer cus)
    {
        if (cus != null)
        {
            this.customers.Remove(cus);
        }
    }
}
```

- **Lớp Customer**

```
class Customer : Observer
{
    private string name;

    public Customer(string name)
    {
        this.name = name;
    }

    public override void ObserverInfo()
    {
        Console.WriteLine("Hi " + this.name + ". You have 3 message from...");
    }
}
```

- **Lớp thực thi chương trình**

```
class Client
{
    static void Main(string[] args)
    {
        Product product = new Product("Laptop Toshiba 14 inch");

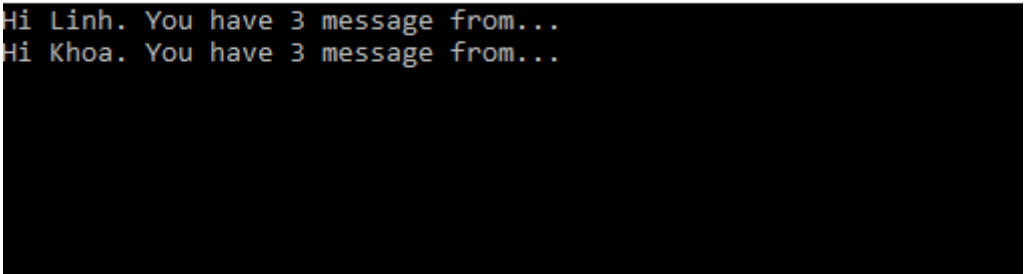
        Customer linh = new Customer("Linh");
        Customer khoa = new Customer("Khoa");

        product.AddCustomer(linh);
        product.AddCustomer(khoa);

        product.ObserverInfo();

        Console.ReadKey();
    }
}
```

- **Kết quả thực hiện**



file:///E:/UIT Courses/K7 HK 9/Desgin Pattern/Báo cáo cuối kỳ/Demo báo cáo cuối kì/DPDe
Hi Linh. You have 3 message from...
Hi Khoa. You have 3 message from...

4. Phụ lục

4.1. Phân loại

		Mục đích		
		Khởi tạo	Cấu trúc	Hành vi
Phạm vi	Lớp	Factory Method,	Adapter(class),	Interpreter, Template Method
	Đối tượng	Abstract Factory, Builder, Prototype, Singleton	Adapter(object), Bridge, Composite, Decorator, Façade, Proxy, Flyweight,	Chain Of Respository, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor
Tổng cộng		5	7	11

4.2. Tần suất sử dụng các mẫu

Tần suất sử dụng	Mẫu GoF	Số lượng
5 - Cao	Singleton, Composite, Façade, Proxy, Iterator, Observer, Template Method	7
4 – Tương đối cao	Abstract Factory, Factory Method, Adapter, Decorator, Command, State, Strategy	7
3 – Trung bình	Builder, Bridge, Chain of Respository	3
2 – Tương đối thấp	Prototype, Flyweight, Mediator, Visitor	4
1 – Thấp	Interpreter, Memento	2

4.3. Độ khó

Cấp độ	Mẫu GoF	Tổng
A – Dễ	Façade, Singleton, Mediator, Iterator, Strategy, Command, Builder, State, Template Method, Factory Method, Memento, Prototype	12
B – Trung bình	Proxy, Decorator, Adapter, Bridge, Observer	5
C – Khó	Composite, Interpreter, Chain Of Responsibility, Abstract Factory, Flyweight, Visitor	6

5. Tài liệu tham khảo

- Sách:
 - Design Elements Of Reusable Object Oriented Software Patterns – tác giả GoF – November 1994 – nhà xuất bản Addison Wesley.
- Tài liệu internet:
 - <http://sce2.umkc.edu/BIT/burris/pl/design-patterns/>
 - <http://www.oodeesign.com/>
 - <http://www.dofactory.com/net/design-patterns>