BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC DÂN LẬP HẢI PHÒNG

ISO 9001:2015

ĐỒ ÁN TỐT NGHIỆP

NGÀNH: CÔNG NGHỆ THÔNG TIN

Sinh viên : Đoàn Văn Thọ

Giảng viên hướng dẫn: TS. Nguyễn Trịnh Đông

HẢI PHÒNG - 2019

BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC DÂN LẬP HẢI PHÒNG

ÁP DỤNG DESIGN PATTERN TRONG PHÁT TRIỂN PHẦN MỀM

ĐỒ ÁN TỐT NGHIỆP ĐẠI HỌC HỆ CHÍNH QUY

NGÀNH: CÔNG NGHỆ THÔNG TIN

Sinh viên : Đoàn Văn Thọ

Giảng viên hướng dẫn : TS. Nguyễn Trịnh Đông

HẢI PHÒNG - 2019

BỘ GIÁO DỤC VÀ ĐÀO TẠO | **CỘNG HÒA XÃ HỘ CHỦ NGHĨA VIỆT NAM**

TRƯỜNG ĐẠI HỌC DÂN LẬP HẢI PHÒNG | **Độc lập – Tự do – Hạnh phúc**

NHIỆM VỤ ĐỀ TÀI TỐT NGHIỆP

Sinh viên: Đoàn Văn Thọ | **Mã SV: 1512111005**

Lớp: CT1901C | **Ngành: Công nghệ thông tin**

Tên đề tài: Áp dụng Design Pattern trong phát triển phần mềm

CÁN BỘ HƯỚNG DẪN ĐỀ TÀI TỐT NGHIỆP

Họ và tên: Nguyễn Trịnh Đông

Học hàm học vị: Tiến sĩ

Cơ quan công tác: Trường đại học Dân lập Hải Phòng

Nội dung hướng dẫn:

……..……………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………………………………………

……………………………………………………………

Đề tài tốt nghiệp được giao ngày 18 tháng 03 năm 2019

Yêu cầu phải hoàn thành trước ngày ….. tháng 06 năm 2019

Hải phòng, ngày ….. tháng 06 năm 2019

Đã nhận nhiệm vụ: Đ.T.T.N	Đã nhận nhiệm vụ: Đ.T.T.N
Sinh viên	Cán bộ hướng dẫn Đ.T.T.N

Hải Phòng, ngày….tháng….năm 2019

HIỆU TRƯỞNG

GS.TS.NGƯT TrầnHữu Nghị

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

Độc lập - Tự do - Hạnh phúc

PHIẾU NHẬN XÉT CỦA GIẢNG VIÊN HƯỚNG DẪN TỐT NGHIỆP

Họ và tên giảng viên: …………………………………………………………………..

Đơn vị công tác: ……………………………………………………………………...

Họ và tên sinh viên: ……………………………… Ngành: ……………..…………….

Nội dung hướng dẫn:

………………………………………………………………………………………

………………………………………………………………………………………..

1. Tinh thần thái độ của sinh viên trong quá trình làm đồ án tốt nghiệp:

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

2. Đánh giá chất lượng của đồ án/khóa luận (so với nội dung yêu cầu đã đề ra trong nhiệm vụ Đ.T. T.N trên các mặt lý luận, thực tiễn, tính toán số liệu…)

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

………………………………………………………………………………………

3. Ý kiến của giảng viên hướng dẫn tốt nghiệp

Đạt ☐ Không đạt ☐ Điểm:……………………………………..

Hải Phòng, ngày .…. tháng 06 năm 2019

Giảng viên hướng dẫn

(Ký và ghi rõ họ tên)

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM

Độc lập - Tự do - Hạnh phúc

PHIẾU NHẬN XÉT CỦA GIẢNG VIÊN CHẤM PHẢN BIỆN

Họ và tên giảng viên: …………………………………………………………………..

Đơn vị công tác: ……………………………………………………………………...

Họ và tên sinh viên: ……………………………… Ngành: ……………………………

Đề tài tốt nghiệp:

……………………………………………………………………………………..

……………………………………………………………………………………

1. Phần nhận xét của giảng viên chấm phản biện

....

....

....

....

....

2. Những mặt còn hạn chế

....

....

....

....

3. Ý kiến của giảng viên chấm phản biện

Được bảo vệ ☐ Không được bảo vệ ☐ Điểm:……………….

Hải Phòng, ngày …… tháng 06 năm 2019

Giảng viên chấm phản biện

(Ký và ghi rõ họ tên)

LỜI CẢM ƠN

Lời đầu tiên em xin chân thành cảm ơn các thầy, cô trong khoa Công nghệ thông tin, trường Đại học Dân lập Hải Phòng đã tạo điều kiện thuận lợi cho em trong quá trình học tập tại trường cũng như trong thời gian thực hiện đồ án tốt nghiệp. Đặc biệt, em muốn gửi lời cảm ơn tới Tiến sỹ Nguyễn Trịnh Đông – giảng viên trực tiếp hướng dẫn, chỉ bảo giúp em khắc phục những khó khăn, thiếu sót để có thể hoàn thành các phần trong đồ án tốt nghiệp từ lý thuyết cho tới thực hành sử dụng công cụ.

Mặc dù đã cố gắng với tất cả nỗ lực của bản thân để hoàn thiện đồ án, nhưng do thời gian có hạn, năng lực và kinh nghiệm còn hạn chế nên đồ án không thể tránh khỏi những thiếu sót. Kính mong nhận được sự đóng góp ý kiến từ phía thầy cô, bạn bè để em có thể nâng cao kiến thức của bản thân, hoàn thiện đồ án được tốt hơn.

Em xin chân thành cảm ơn!

Hải Phòng, ngày ….. tháng 06 năm 2019.

Sinh viên thực hiện

Đoàn Văn Thọ

MỤC LỤC

LỜI CẢM ƠN

MỤC LỤC

MỞ ĐẦU 1

DANH MỤC HÌNH VẼ VÀ BẢNG BIỂU 2

DANH MỤC TỪ VIẾT TẮT 4

CHƯƠNG 1: KIẾN THỨC CƠ BẢN 5

1.1 Vấn đề trong thiết kế phần mềm hướng đối tượng 5

1.2 Lịch sử hình thành của Design Pattern 5

1.3 Khái niệm 6

1.4 Đặc điểm chung 7

1.5 Ưu và nhược điểm của Design Pattern 8

1.5.1 Ưu điểm 8

1.5.2 Nhược điểm 8

1.6 Phân loại Design Pattern 9

1.6.1 Nhóm Creational 11

1.6.2 Nhóm Structural 12

1.6.3 Nhóm Behavioral 13

1.7 Kết luận 15

CHƯƠNG 2: CÁC KỸ THUẬT CỦA DESIGN PATTERN 16

2.1 Nhóm Creational 16

2.1.1 Singleton Design Pattern 16

2.1.2 Abstract Factory 17

2.1.3 Factory Method 18

2.1.4 Builder 19

2.1.5 Prototype 21

2.2 Nhóm Structural 23

2.2.1 Adapter 23

2.2.2 Bridge 25

2.2.3 Composite 27
2.2.4 Decorator 28
2.2.5 Facade 30
2.2.6 Flyweight 32
2.2.7 Proxy 34
2.3. Nhóm Behavioral 37
2.3.1 Chain of Responsibility 37
2.3.2 Command 39
2.3.3 Interpreter 41
2.3.4 Iterator 43
2.3.5 Mediator 45
2.3.6 Memento 47
2.3.7 Observer 48
2.3.8 State 50
2.3.9 Strategy 51
2.3.10 Template Method 53
2.3.11 Visitor 54
2.4. Kết luận 56
CHƯƠNG 3: ÁP DỤNG DESIGN PATTERN TRONG PHÁT TRIỂN PHẦN MỀM 57
3.1. Giới thiệu 57
3.2. Bài toán đăng ký tuyển sinh mầm non 57
3.3. Mô tả các nghiệp vụ 57
3.3.1 Bản mô tả công việc 57
3.3.2 Danh sách các công việc cần thực hiện 58
3.4 Phân tích thiết kế hướng đối tượng 58
3.4.1 Biểu đồ trường hợp sử dụng (Use case diagram) 59
3.4.2 Biểu đồ trình tự quản lý tuyển sinh (Sequence diagram) 60
3.4.3 Biểu đồ lớp (Class diagram) 61

3.4.4 Biểu đồ chuyển trạng thái (State transition diagram) 62
3.5 Áp dụng Design Pattern vào bài toán 62
3.5.1 Trừu tượng hóa 63
3.5.2 Quy trình tuyển sinh 64
3.5.3 Bảo mật hệ thống 64
3.6 Chương trình thực nghiệm 65
KẾT LUẬN 68
DANH MỤC TÀI LIỆU THAM KHẢO 69

MỞ ĐẦU

Ngày nay, công nghệ thông tin được coi là ngành quyền lực bậc nhất với hàng loạt ứng dụng trong mọi lĩnh vực của đời sống - từ sản xuất, kinh doanh đến giáo dục, y tế, văn hóa... Đặc biệt, ở thời kỳ Cách mạng 4.0 - mà tại Việt Nam cơ bản là ứng dụng như công nghệ tự động hóa, trao đổi dữ liệu. Trong công nghệ sản xuất, công nghệ thông tin càng khẳng định được tầm quan trọng của mình - vừa là nền tảng, vừa là động lực để bắt kịp đà phát triển của thế giới.

Vậy để hệ thống có tính tái sử dụng cao, tăng tính đóng gói, không lặp lại cũng như phạm vi logic được thu hẹp thì áp dụng Design Pattern trong phát triển phần mềm là một sự lựa chọn thích hợp.

Với mong muốn được tìm hiểu sâu về việc phát triển phần mềm nên em đã chọn đề tài "Áp dụng Design Pattern trong phát triển phần mềm." Trong quá trình làm đồ án, do còn hạn chế về thời gian và kinh nghiệm thực tế, em mong nhận được những góp ý chân thành từ thầy cô và các bạn.

Đề tài giới thiệu về những lý thuyết cơ bản của Design Pattern, phân tích đánh giá các kỹ thuật và xây dựng ứng dụng thực nghiệm.

Đồ án được tổ chức làm 5 phần như sau:

- Mở đầu: Trình bày rõ lý do chọn đề tài, mục tiêu nghiên cứu đồ án và bố cục của đồ án.

- Chương 1: *Kiến thức cơ bản*. Chương này trình bày các khái niệm cơ bản, đặc điểm, phân loại, ưu và nhược điểm của Design Pattern.

- Chương 2: *Các kỹ thuật của Design Patetrn*. Chương này trình bày chi tiết về các kỹ thuật cũng như cách xây dựng mẫu trong thiết kế phần mềm.

- Chương 3: *Áp dụng Desgin Pattern trong phát triển phần mềm.* Chương này trình bày chủ yếu về phân tích thiết kế hệ thống hướng đối tượng và áp dụng Design Pattern vào bài toán.

- Kết luận: Phần này đưa ra những kết quả đồ án đạt được, những thiếu sót chưa thực hiện được và hướng phát triển đề tài trong tương lai.

DANH MỤC HÌNH VẼ VÀ BẢNG BIỂU

Hình 1 – 1: Quy tắc thiết kế hướng đối tượng

Hình 1 – 2: Mối quan hệ giữa 23 Design Pattern

Hình 1 – 3: Các mẫu Design Pattern trong nhóm Creational

Hình 1 – 4: Các mẫu Design Pattern trong nhóm Structural

Hình 1 – 5: Các mẫu Design Pattern trong nhóm Behavioral

Hình 2 – 1: Sơ đồ UML mô tả Singleton Pattern

Hình 2 – 2: Code minh họa của Singleton Pattern

Hình 2 - 3: Sơ đồ UML mô tả Builder Pattern

Hình 2 - 4: Sơ đồ UML mô tả Prototype Pattern

Hình 2 – 5: Sơ đồ UML cách cài đặt Object Pattern

Hình 2 – 6: Sơ đồ UML cách cài đặt Class Pattern

Hình 2 – 7: Sơ đồ UML mô tả Bridge Pattern

Hình 2 – 8: Sơ đồ UML mô tả Composite Pattern

Hình 2 – 9: Sơ đồ UML mô tả Decorator Pattern

Hình 2 – 10: Sơ đồ UML mô tả Facede Pattern

Hình 2 – 11: Sơ đồ UML mô tả Flyweight Pattern

Hình 2 – 12: Sơ đồ UML mô tả Proxy Pattern

Hình 2 – 13: Quy trình thực hiện của Chain of Responsibility Pattern

Hình 2 – 14: Sơ đồ UML mô tả Chain of Responsibility Pattern

Hình 2 – 15: Sơ đồ UML mô tả Command Pattern

Hình 2 – 16: Sơ đồ UML mô tả Interpreter Pattern

Hình 2 – 17: Sơ đồ UML mô tả Iterator Pattern

Hình 2 – 18: Sơ đồ UML mô tả Mediator Pattern

Hình 2 – 19: Sơ đồ UML mô tả Memento Pattern

Hình 2 – 20: Sơ đồ UML mô tả Observer Pattern

Hình 2 – 21: Sơ đồ UML mô tả State Pattern

Hình 2 – 21: Sơ đồ UML mô tả Strategy Pattern

Hình 2 – 22: Sơ đồ UML mô tả Template Method Pattern

Hình 3 -1: Biểu đồ Use case diagram Quản lý tuyển sinh
Hình 3 -2: Biểu đồ Sequence diagram quản lý tuyển sinh
Hình 3 - 3: Biểu đồ Class diagram
Hình 3 - 4: Biểu đồ State transition diagram
Hình 3 - 5: Trừu tượng hóa các lớp của hệ thống
Hình 3 - 6: Quy trình tuyển sinh nhập học của trẻ
Hình 3 - 7: Quy trình đăng nhập vào hệ thống
Hình 3 - 8: Login vào hệ thống
Hình 3 - 9: Giao diện chức năng nghiệp vụ quản lý tuyển sinh
Hình 3 - 10: Chức năng thêm học sinh vào hệ thống
Hình 3 - 11: Chức năng sửa thông tin học sinh trong hệ thống
Hình 3 - 12: Chức năng xóa học sinh khỏi hệ thống
Hình 3 - 13: Áp dụng Singleton Pattern vào quy trình đăng nhập hệ thống
Hình 3 - 14: Áp dụng Singleton Pattern vào quy trình kết nối dữ liệu

DANH MỤC TỪ VIẾT TẮT

STT	KÝ HIỆU	CỤM TỪ ĐẦY ĐỦ	Ý NGHĨA
1	SDLC	System Development Life Cycle	Vòng đời phát triển phần mềm
2	UML	Unified Modeling Language	Ngôn ngữ mô hình thống nhất
3	PloP	Pattern Language of Programming Design	
4	DP	Design Pattern	Mẫu thiết kế
5	UC	Use case	
6			
7			
8			
9			
10			

CHƯƠNG 1: KIẾN THỨC CƠ BẢN

Phát triển phần mềm hướng đối tượng là một kỹ thuật khó trong lập trình phần mềm. Chúng ta cần mô tả các thuộc tính, hành vi của đối tượng một cách chính xác. Sau đó sử dụng các kỹ thuật trong lập trình hướng đối tượng như kế thừa, tính đa hình, lớp trừu tượng, phương thức ảo, v.v. là một công việc khó. Đặc biệt, việc tái sử dụng và đảm bảo tính đúng đắn của phần mềm là một trong những yêu cầu khó hơn. Design Pattern là một phương pháp nhằm khắc phục những khó khăn trong phát triển phần mềm hướng đối tượng. Phương pháp này được đánh giá là kỹ thuật có nhiều tính ưu việt như khắc phục yếu điểm của kế thừa, tính đa hình, đảm bảo tính đúng đắn của phần mềm. Trong khi đó vẫn đảm bảo được các tính chất của lập trình hướng đối tượng. Chương này trình bày các khái niệm cơ bản, đặc điểm, phân loại, ưu và nhược điểm của Design Pattern.

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

Việc thiết kế một phần mềm hướng đối tượng là một công việc khó và việc thiết kế một phần mềm hướng đối tượng phục vụ cho mục đích dùng lại càng khó hơn. Vì thế, phải tìm ra những đối tượng phù hợp, đại diện cho một lớp các đối tượng. Sau đó thiết kế giao diện, tạo cây kế thừa cho chúng và thiết lập các mối quan hệ. Thiết kế 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 chí 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 không thể mong chờ thiết kế của mình sẽ là đúng và đảm bảo các tiêu chí trên ngay được. Thực tế là nó cần phải được thử nghiệm sau vài lần và sau đó sẽ được sửa chữa lại. *[4]*

Đứng trước một vấn đề, một người phân tích thiết kế tốt có thể đưa ra nhiều phương án giải quyết, 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 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 nhau.

1.2 Lịch sử hình thành của Design Pattern.

Năm 1994, tại hội nghị PloP (Pattern Language of Programming Design) đã được tổ chức. Cũng trong năm này quyển sách Design Pattern: 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 nổi 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 với định dạng chuẩn được dùng làm tài liệu cho mỗi mẫu và nổi tiếng với tên Gang of Four 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. *[1]*

Năm 2000, Evitts có tổng kết về cách các mẫu xâm nhập vào thế giới phần mềm. 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. *[1]*

1.3 Khái niệm

Theo Christopher Alexander nói: "*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*". *[4]*

Theo cuốn Software Engineering thì một mẫu thiết kế phần mềm là một giải pháp chung, có thể tái sử dụng cho một vấn đề thường xảy ra trong một bối cảnh nhất định trong thiết kế phần mềm. Các mẫu này không phải là một thiết kế đã hoàn chỉnh để có thể được chuyển đổi trực tiếp thành mã nguồn hoặc mã máy. Do đó, các mẫu thiết kế là một mô tả hoặc khuôn mẫu cho cách giải quyết vấn đề có thể được sử dụng trong nhiều tình huống khác nhau. Các mẫu thiết kế là các mô hình hóa các tình huống thực tế một cách tốt nhất mà lập trình viên có thể sử dụng để giải quyết các vấn đề phổ biến khi thiết kế một ứng dụng hoặc hệ thống.

Qua quá trình nghiên cứu, các tác giả đã tổng hợp các yếu tố chính về Design Pattern gồm:

- *Là tập các giải pháp 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 thì 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.*

1.4 Đặc điểm chung

Mẫu được hiểu theo nghĩa tái sử dụng ý tưởng hơn là mã lệnh. Mẫu cho phép các nhà thiết kế có thể cùng ngồi lại với nhau và cùng giải quyết một vấn đề nào đó mà không phải mất nhiều thời gian tranh cãi. Ngoài ra, mẫu cũng cung cấp những thuật ngữ và khái niệm chung trong thiết kế. Nói một cách đơn giản, khi đề cập đến một mẫu nào đấy, bất kỳ ai biết mẫu đó đều có thể nhanh chóng hình dung ra "bức tranh" của giải pháp. Và cuối cùng, nếu áp dụng mẫu hiệu quả thì việc bảo trì phần mềm cũng được tiến hành thuận lợi hơn, nắm bắt kiến trúc hệ thống nhanh hơn.

Mẫu hỗ trợ tái sử dụng kiến trúc và mô hình thiết kế phần mềm theo quy mô lớn. Cần phân biệt Design Pattern với Framework. Framework hỗ trợ tái sử dụng mô hình thiết kế và mã nguồn ở mức chi tiết hơn. Trong khi đó, Design Pattern được vận dụng ở mức tổng quát hơn, giúp các nhà phát triển hình dung và ghi nhận các cấu trúc tĩnh và động cũng như quan hệ tương tác giữa các giải pháp trong quá trình thiết kế ứng dụng.

Một cách tổng quát, mẫu có bốn thành phần chính sau đây:

- ***Tên mẫu** (**pattern name**): Là một tên mang tính tổng quát nhất để mô tả giải pháp và kết quả của một bài toán thiết kế. Tên này thường ngắn gọn khoảng một vài từ. Tên của mẫu còn đóng vai trò gia tăng vốn từ vựng về mẫu. Tên của mẫu được sử dụng để trao đổi với các thành viên trong nhóm, sử dụng trong các văn bản, tài liệu, v.v. Lựa chọn một tên mẫu tốt là một trong những công việc khó nhất để mọi người hiểu đúng nội dung của mẫu và có thể trao đổi với những cái khác.*

- ***Bài toán (problem):** Bài toán để trả lời câu hỏi khi nào chúng ta áp dụng mẫu? Bài toán diễn giải vấn đề và tình huống cần giải quyết. Nó có thể diễn giải các vấn đề mẫu cụ thể chẳng hạn như làm thế nào biểu diễn các thuật toán cũng như các đối tượng. Nó cũng có thể miêu tả lớp hoặc các cấu trúc đối tượng là những dấu hiệu của một thiết kế không linh hoạt. Đôi khi, vấn đề bao gồm danh sách các điều kiện phải thỏa trước khi hiểu để áp dụng chúng.*

- ***Giải pháp (solution):** Giải pháp mô tả các thành phần tạo nên thiết kế, các quan hệ và sự tương tác giữa chúng. Giải pháp không mô tả một cài đặt hay một thiết kế cụ thể bởi vì một mẫu giống như cái khuôn mà có thể áp dụng trong nhiều tính huống khác nhau. Do đó, mẫu cung cấp một sự mô tả ở mức trừu tượng của vấn đề thiết kế và làm cách nào để sắp xếp*

một cách tổng quát các thành phần (các lớp, các đối tượng trong từng trường hợp cụ thể) để giải quyết chúng.

- ***Hậu quả (consequences):*** *Hậu quả là sự trao đổi và thành quả của việc áp dụng các mẫu. Dù hậu quả thường không tránh khỏi khi chúng ta mô tả các quyết định thiết kế, chúng quan trọng cho việc đánh giá sự lựa chọn thay thế mẫu và hiểu chi phí và lợi ích của việc áp dụng các mẫu. Hậu quả đối với phần mềm thường liên quan đến hai yếu tố không gian và thời gian. Chúng định hướng các chủ đề ngôn ngữ lập trình cũng như sự thực thi. Vì tái sử dụng là một nhân tố thường sử dụng trong lập trình hướng đối tượng, hậu quả của mẫu bao gồm sự tác động của chúng lên độ phức tạp hệ thống, sự mở rộng hoặc tính khả chuyển. Liệt kê các hậu quả một cách rõ ràng giúp chúng ta hiểu và đánh giá chúng chính xác hơn.*

Đặc điểm chung của mẫu là đa tương thích, không phụ thuộc vào ngôn ngữ lập trình, công nghệ hoặc các nền tảng triển khai.

1.5 Ưu và nhược điểm của Design Pattern

1.5.1 Ưu điểm

Mẫu có thể tái sử dụng trong nhiều dự án, cung cấp các giải pháp giúp xác định kiến trúc hệ thống. Mẫu nắm bắt những kinh nghiệm kỹ thuật phần mềm, cung cấp sự minh bạch cho việc thiết kế một ứng dụng. Mẫu là những giải pháp đã được chứng minh và chứng thực vì chúng được xây dựng dựa trên kiến thức và kinh nghiệm của các nhà chuyên gia phát triển phần mềm. Các mẫu thiết kế không đảm bảo một giải pháp tuyệt đối cho một vấn đề. Chúng cung cấp sự rõ ràng cho kiến trúc hệ thống và khả năng xây dựng một hệ thống tốt hơn.

1.5.2 Nhược điểm

Dù thiết kế mẫu đem lại nhiều lợi ích trong phát triển phần mềm. Tuy nhiên việc sử dụng mẫu cũng còn tùy thuộc vào từng tình huống cụ thể trong từng dự án cụ thể. Nhược điểm của mẫu cũng bộc lộ qua một số tính huống sau đây:

- *Việc sử dụng quá nhiều mẫu cũng như buộc chúng phải phù hợp với chương trình sẽ làm cho các đoạn mã trở nên rắc rối và khó hiểu hơn.*
- *Không có một phương pháp phát triển phần mềm nào là hoàn thiện và Design Pattern không phải là một ngoại lệ.*
- *Không thích hợp cho những lập trình viên còn ít kinh nghiệm cũng như chưa*

hiểu hết về Design Pattern mà áp dụng vào trong chương trình.

1.6 Phân loại Design Pattern

Phân loại mẫu nhằm mục đích nhanh chóng tìm ra loại mẫu phù hợp trong phát triển phần mềm. Năm 1994, bốn tác giả Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides đã cho xuất bản một cuốn sách với tiêu đề Design Patterns – Elements of Reusable Object-Oriented Software, đây là khởi nguồn của khái niệm Design Pattern trong lập trình phần mềm.

Bốn tác giả trên được biết đến rộng rãi dưới tên Gang of Four. Theo quan điểm của bốn người, Design Pattern chủ yếu được dựa theo những quy tắc sau đây về thiết kế hướng đối tượng.

By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

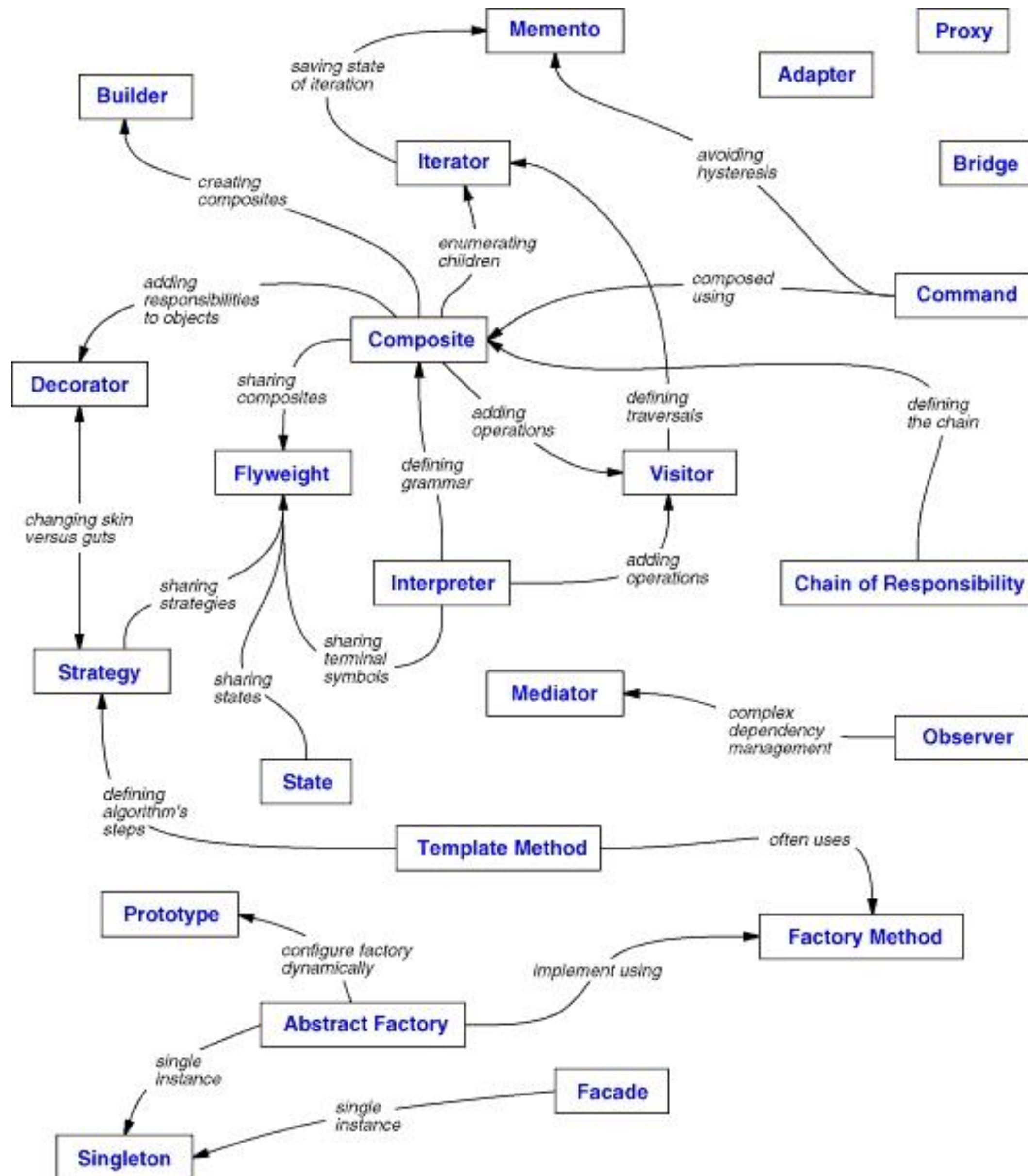
Hình 1 – 1: Quy tắc thiết kế hướng đối tượng

Hệ thống các mẫu Design Pattern hiện có 23 mẫu được định nghĩa trong cuốn "Design Patterns Elements of Reusable Object Oriented Software" và được chia thành 3 nhóm:

- ***Creational Pattern*** *(nhóm khởi tạo – 5 mẫu) gồm: Factory Method, Abstract Factory, Builder, Prototype, Singleton. Những Design Pattern loại này cung cấp một giải pháp để tạo ra các object và che giấu được logic của việc tạo ra nó, thay vì tạo ra object một cách trực tiếp bằng cách sử dụng method new. Điều này giúp cho chương trình trở nên mềm dẻo hơn trong việc quyết định object nào cần được tạo ra trong những tình huống được đưa ra. [3]*
- ***Structural Pattern*** *(nhóm cấu trúc – 7 mẫu) gồm: Adapter, Bridge, Composite, Decorator, Facade, Flyweight và Proxy. Những Design*

Pattern loại này liên quan tới class và các thành phần của object. Nó dùng để thiết lập, định nghĩa quan hệ giữa các đối tượng. [3]

- ***Behavioral Pattern*** *(nhóm tương tác / hành vi – 11 mẫu) gồm: Interpreter, Template Method, Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy và Visitor. Nhóm này dùng trong thực hiện các hành vi của đối tượng, sự giao tiếp giữa các object với nhau. [3]*



Hình 1 – 2: Mối quan hệ giữa 23 Design Pattern

1.6.1 Nhóm Creational

Hình 1 – 3: Các mẫu Design Pattern trong nhóm Creational

Singleton:

- Đảm bảo 1 class chỉ có 1 instance và cung cấp 1 điểm truy xuất toàn cục đến nó.

Abstract Factory:

- Cung cấp một interface cho việc tạo lập các đối tượng (có liên hệ với nhau) mà không cần quy định lớp khi hay xác định lớp cụ thể (concrete) 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.

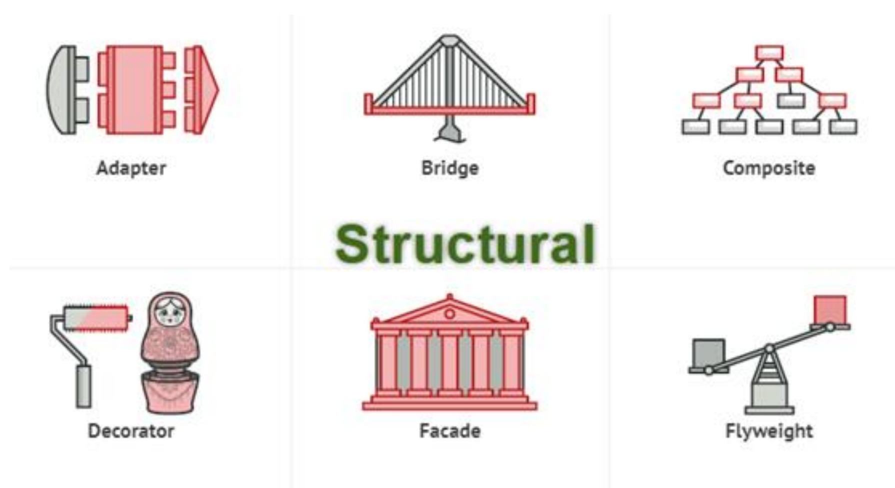
Builder:

- 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.

Prototype:

- Quy 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.

1.6.2 Nhóm Structural



Hình 1 – 4: Các mẫu Design Pattern trong nhóm Structural

Adapter:

- Do vấn đề tương thích, thay đổi interface của một lớp thành một interface khác phù hợp với yêu cầu người sử dụng lớp.

Bridge:

- Tách rời ngữ nghĩa của một vấn đề khỏi việc cài đặt, mục đích để cả hai bộ phận (ngữ nghĩa và cài đặt) có thể thay đổi độc lập 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 thuần nhất như nhau.
- Tạo quan hệ thứ bậc bao gộp giữa các đối tượng. Client có thể xem đối tượng bao gộp và bị bao gộp như nhau -> khả năng tổng quát hoá trong code của client -> dễ phát triển, nâng cấp, bảo trì.

Decorator:

- Gán thêm trách nhiệm cho đối tượng (mở rộng chức năng) vào lúc chạy (dynamically).

Facade:

- Cung cấp một interface thuần nhất cho một tập hợp các interface trong một "hệ thống con" (subsystem). Nó định nghĩa 1 interface cao hơn các interface có sẵn để làm cho hệ thống con dễ sử dụng hơn.

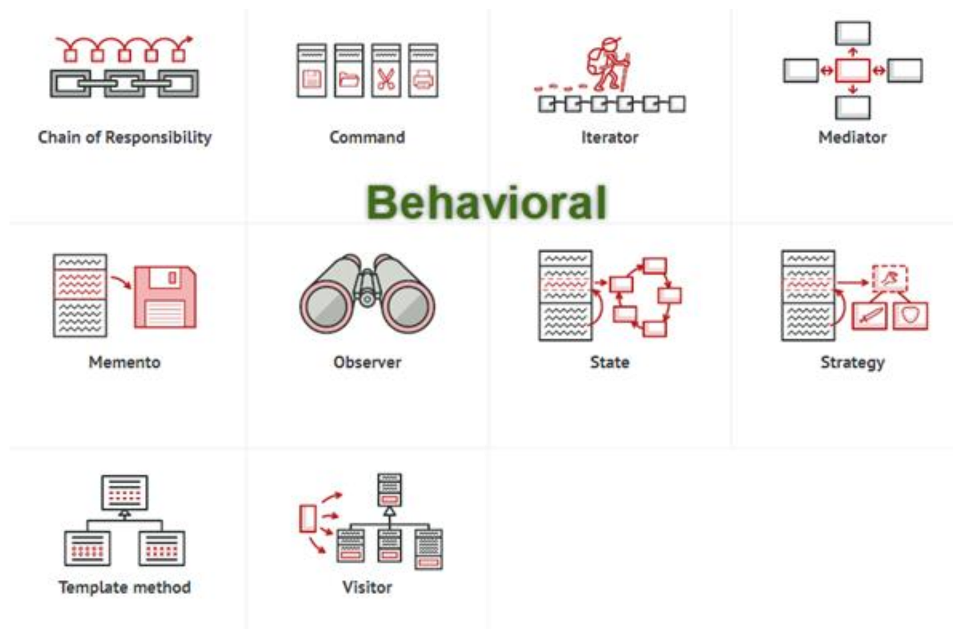
Flyweight:

- Sử dụng việc chia sẻ để thao tác hiệu quả trên một số lượng lớn đối tượng "cỡ nhỏ" (chẳng hạn paragraph, dòng, cột, ký tự…).

Proxy:

- Cung cấp đối tượng đại diện cho một đối tượng khác để hỗ trợ hoặc kiểm soát quá trình truy xuất đối tượng đó. Đối tượng thay thế gọi là proxy.

1.6.3 Nhóm Behavioral



Hình 1 – 5: Các mẫu Design Pattern trong nhóm Behavioral

Chain of Responsibility:

- Khắc phục việc ghép cặp giữa bộ gửi và bộ nhận thông điệp. 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 hơn 1 đối tượng có có cơ hội xử lý request. Liên kết các đối tượng nhận request thành 1 dây chuyền rồi gửi request xuyên qua từng đối tượng xử lý đến khi gặp đối tượng xử lý cụ thể.

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. Đóng gói request vào trong một Object, nhờ đó có thể thông số hoá chương trình

nhận request và thực hiện các thao tác trên request: sắp xếp, log, undo…

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:

- Hiệu chỉnh và trả lại như cũ trạng thái bên trong của đối tượng mà vẫn không vi phạm việc bao bọc dữ liệu.

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:

- Cho phép một đối tượng thay đổi hành vi khi trạng thái bên trong của nó thay đổi, ta có cảm giác như class của đối tượng bị 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.

1.7 Kết luận

Chương này khóa luận trình bày những khái niệm cơ bản cũng như ưu nhược điểm của Design Pattern. Design Pattern thể hiện tính kinh nghiệm của công việc lập trình, xây dựng và thiết kế phần mềm. Người hiểu và vận dụng được Design Pattern trong quá trình thiết kế hệ thống sẽ tiết kiệm được rất nhiều thời gian, chi phí, dễ phát triển, mở rộng và bảo trì. Tuy nhiên không nên quá lạm dụng Design Pattern trong phát triển phần mềm. Khi muốn tiếp cận đến một Design Pattern mới thì hãy tập trung chú ý vào ba yếu tố quan trọng sau:

- *Mẫu được sử dụng khi nào, vấn đề mà Design Pattern đó giải quyết là gì.*
- *Sơ đồ UML mô tả Design Pattern.*
- *Code minh họa, ứng dụng thực tiễn của mẫu là gì. [2]*

CHƯƠNG 2: CÁC KỸ THUẬT CỦA DESIGN PATTERN

Hiểu và nắm vững các thiết kế mẫu là một trong những yếu tố quan trọng đối với lập trình viên hiện nay. Với ưu thế có sẵn, thiết kế mẫu giúp cho triển khai phần mềm có quy mô lớn được nhanh chóng. Dựa trên các mẫu đã được áp dụng trong thiết kế, các thành viên trong nhóm và các thành viên liên quan có thể hình dung cụ thể những vấn đề cần được giải quyết. Chương này khóa luận trình bày chi tiết các mẫu và các thành phần liên quan để từ đó có cái nhìn tổng quát trong quá trình lựa chọn các mẫu để giải quyết bài toán đặt ra.

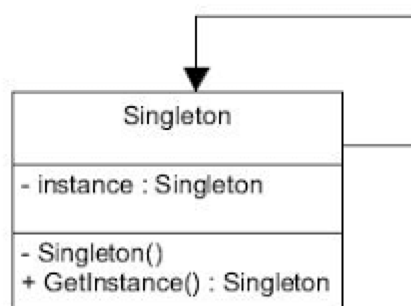
2.1 Nhóm Creational

2.1.1 Singleton Design Pattern

Đôi khi, trong quá trình phân tích thiết kế một hệ thống, chúng ta mong muốn có những đối tượng cần tồn tại duy nhất và có thể truy xuất mọi lúc mọi nơi. Làm thế nào để hiện thực được một đối tượng như thế khi xây dựng mã nguồn? Chúng ta có thể nghĩ tới việc sử dụng một biến toàn cục (*global variable*). Tuy nhiên, việc sử dụng biến toàn cục nó phá vỡ quy tắc của OOP (encapsulation). Để giải bài toán trên, người ta hướng đến một giải pháp là sử dụng *Singleton Pattern*.

2.1.1.1 Giới thiệu về Singleton Pattern

Singleton đảm bảo chỉ duy nhất một thể hiện (instance) được tạo ra và nó sẽ cung cấp cho bạn một phương thức để có thể truy xuất được thể hiện duy nhất đó mọi lúc mọi nơi trong chương trình.



Hình 2 – 1: Sơ đồ UML mô tả Singleton Pattern

Sử dụng Singleton khi chúng ta muốn:

- Đảm bảo rằng chỉ có một instance của lớp.

- Việc quản lý việc truy cập tốt hơn vì chỉ có một thể hiện duy nhất.
- Có thể quản lý số lượng thể hiện của một lớp trong giớn hạn chỉ định.

2.1.1.2 Cài đặt Singleton Pattern

Có rất nhiều cách để implement Singleton Pattern. Nhưng dù cho việc implement bằng cách nào đi nữa cũng dựa vào nguyên tắc dưới đây cơ bản dưới đây:

- **private constructor** để hạn chế truy cập từ class bên ngoài.
- Đặt **private static final variable** đảm bảo biến chỉ được khởi tạo trong class.
- Có một method **public static** để **return instance** được khởi tạo ở trên.

```
public class BillPughSingleton {
        private BillPughSingleton() { }

    public static BillPughSingleton getInstance() {
        return SingletonHelper.INSTANCE;
    }

    private static class SingletonHelper {
        private static final BillPughSingleton INSTANCE = new BillPughSingleton();
    }
}
```

Hình 2 – 2: Code minh họa của Singleton Pattern

2.1.1.3 Sử dụng Singleton Pattern

- Vì class dùng Singleton chỉ tồn tại 1 instance (thể hiện) nên nó thường được dùng cho các trường hợp giải quyết các bài toán cần truy cập vào các ứng dụng như: Shared resource, Logger, Configuration, Caching, Thread pool, …
- Một số Design Pattern khác cũng sử dụng Singleton để triển khai: Abstract Factory, Builder, Prototype, Facade, …

2.1.2 Abstract Factory

2.1.2.1 Giới thiệu về Abstract Factory Pattern

Là phương pháp tạo ra một Super-Factory dùng để tạo ra các Factory khác. Hay còn được gọi là factory của các factory. Abstract Factory Pattern là một mẫu cấp cao hơn so với Factory Method Pattern. Trong Abstract Factory Pattern, một interface có

nhiệm vụ tạo ra một Factory của các object có liên quan tới nhau mà không cần phải chỉ ra trực tiếp các class của object. Mỗi Factory được tạo ra có thể tạo ra các object bằng phương pháp giống như Factory Pattern. Hãy tưởng tượng, Abstract Factory như là một nhà máy lớn chứa nhiều nhà máy nhỏ, trong các nhà máy đó có những xưởng sản xuất, các xưởng đó tạo ra những sản phẩm khác nhau.

2.1.2.2 Cài đặt Abstract Factory Pattern

Một Abstract Factory Pattern bao gồm các thành phần cơ bản sau:

- **AbstractFactory**: Khai báo dạng interface hoặc abstract class chứa các phương thức để tạo ra các đối tượng abstract.
- **ConcreteFactory**: Xây dựng, cài đặt các phương thức tạo các đối tượng cụ thể.
- **AbstractProduct**: Khai báo dạng interface hoặc abstract class để định nghĩa đối tượng abstract.
- **Product**: Cài đặt của các đối tượng cụ thể, cài đặt các phương thức được quy định tại AbstractProduct.
- **Client**: là đối tượng sử dụng AbstractFactory và các AbstractProduct.

2.1.2.3 Lợi ích của Abstract Factory Pattern

Cung cấp hướng tiếp cận với interface thay thì các implement, che giấu sự phức tạp của việc khởi tạo các đối tượng với người dùng (client), độc lập giữa việc khởi tạo đối tượng và hệ thống sử dụng, …

Giúp tránh được việc sử dụng điều kiện logic bên trong Factory Pattern. Khi một Factory Method lớn (có quá nhiều sử lý if-else hay switch-case), chúng ta nên sử dụng theo mô hình Abstract Factory để dễ quản lý hơn (cách phân chia có thể là gom nhóm các sub-class cùng loại vào một Factory).

Abstract Factory Pattern là factory của các factory, có thể dễ dạng mở rộng để chứa thêm các factory và các sub-class khác.

2.1.3 Factory Method

2.1.3.1 Giới thiệu về Factory Method Pattern

Nhiệm vụ của Factory Pattern là quản lý và trả về các đối tượng theo yêu cầu, giúp cho việc khởi tạo đổi tượng một cách linh hoạt hơn. Factory Pattern đúng nghĩa là một nhà máy và nhà máy này sẽ "**sản xuất**" các đối tượng theo yêu cầu của chúng ta. Trong Factory Pattern, chúng ta tạo đối tượng mà không để lộ logic tạo đối tượng ở

phía người dùng và tham chiếu đến đối tượng mới được tạo ra bằng cách sử dụng một interface chung. Factory Pattern được sử dụng khi có một class cha (super-class) với nhiều class con (sub-class), dựa trên đầu vào và phải trả về 1 trong những class con đó.

2.1.3.2 Cài đặt Factory Pattern

Một Factory Pattern bao gồm các thành phần cơ bản sau:

- **Super Class**: một supper class trong Factory Pattern có thể là một interface, abstract class hay một class thông thường.
- **Sub Classes**: các sub class sẽ implement các phương thức của supper class theo nghiệp vụ riêng của nó.
- **Factory Class**: một class chịu tránh nhiệm khởi tạo các đối tượng sub class dựa theo tham số đầu vào. Lưu ý: lớp này là Singleton hoặc cung cấp một public static method cho việc truy xuất và khởi tạo đối tượng. Factory class sử dụng if-else hoặc switch-case để xác định class con đầu ra.

2.1.3.3 Sử dụng Factory Pattern

Chúng ta có một super class với nhiều class con và dựa trên đầu vào, chúng ta cần trả về một class con. Mô hình này giúp chúng ta đưa trách nhiệm của việc khởi tạo một lớp từ phía người dùng (client) sang lớp Factory. Chúng ta không biết sau này sẽ cần đến những lớp con nào nữa. Khi cần mở rộng, hãy tạo ra sub class và implement thêm vào factory method cho việc khởi tạo sub class này.

2.1.3.4 Lợi ích của Factory Pattern

Factory Pattern giúp giảm sự phụ thuộc giữa các module (loose coupling): cung cấp 1 hướng tiếp cận với interface thay thì các implement. Giúp chuơng trình độc lập với những lớp cụ thể mà chúng ta cần tạo 1 đối tượng, code ở phía client không bị ảnh hưởng khi thay đổi logic ở factory hay sub class.

Mở rộng code dễ dàng hơn: khi cần mở rộng, chỉ việc tạo ra sub class và implement thêm vào factory method.

2.1.4 Builder

2.1.4.1 Giới thiệu về Builder Pattern

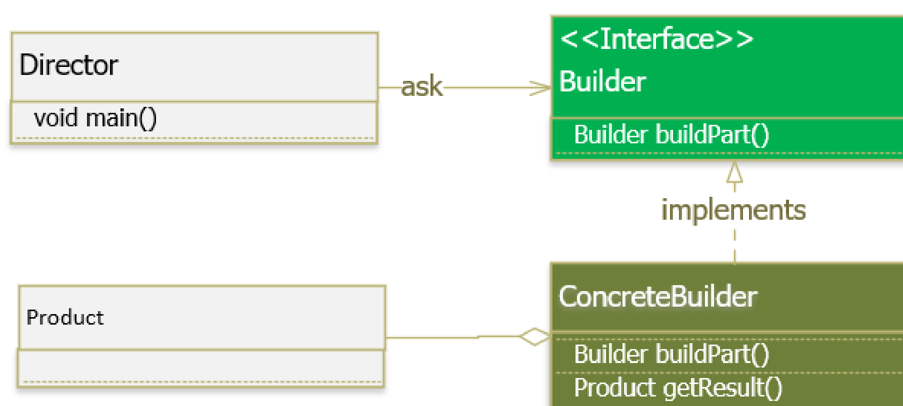
Là mẫu thiết kế đối tượng được tạo ra để xây dựng một đối tượng phức tạp bằng cách sử dụng các đối tượng đơn giản và sử dụng tiếp cận từng bước, việc xây dựng

các đối tượng độc lập với các đối tượng khác. Builder Pattern được xây dựng để khắc phục một số nhược điểm của Factory Pattern và Abstract Factory Pattern khi mà Object có nhiều thuộc tính.

Có ba vấn đề chính với Factory Pattern và Abstract Factory Pattern khi Object có nhiều thuộc tính:

- Quá nhiều tham số phải truyền vào từ phía client tới Factory Class.
- Một số tham số có thể là tùy chọn nhưng trong Factory Pattern, chúng ta phải gửi tất cả tham số, với tham số tùy chọn nếu không nhập gì thì sẽ truyền là null.
- Nếu một Object có quá nhiều thuộc tính thì việc tạo sẽ phức tạp.

2.1.4.2 Cài đặt Builder Pattern



Hình 2 - 3: Sơ đồ UML mô tả Builder Pattern

Một builder gồm các thành phần cơ bản sau:

- **Product** : đại diện cho đối tượng cần tạo, đối tượng này phức tạp, có nhiều thuộc tính.
- **Builder** : là abstract class hoặc interface khai báo phương thức tạo đối tượng.
- **ConcreteBuilder** : kế thừa Builder và cài đặt chi tiết cách tạo ra đối tượng. Nó sẽ xác định và nắm giữ các thể hiện mà nó tạo ra, đồng thời nó cũng cung cấp phương thức để trả các các thể hiện mà nó đã tạo ra trước đó.
- **Director** / Client: là nơi sẽ gọi tới Builder để tạo ra đối tượng.

2.1.4.3 Lợi ích của Builder Pattern

Hỗ trợ, loại bớt việc phải viết nhiều constructor. Code dễ đọc, dễ bảo trì hơn khi số lượng thuộc tính (propery) bắt buộc để tạo một object từ 4 hoặc 5 propery. Giảm bớt số lượng constructor, không cần truyền giá trị null cho các tham số không sử dụng.

Ít bị lỗi do việc gán sai tham số khi mà có nhiều tham số trong constructor: bởi vì người dùng đã biết được chính xác giá trị gì khi gọi phương thức tương ứng. Đối tượng được xây dựng an toàn hơn: bởi vì nó đã được tạo hoàn chỉnh trước khi sử dụng.

2.1.4.4 Nhược điểm của Builder Pattern

Builder Pattern có nhược điểm là duplicate code khá nhiều: do cần phải copy tất cả các thuộc tính từ class Product sang class Builder.

Tăng độ phức tạp của code (tổng thể) do số lượng class tăng lên.

2.1.4.5 Sử dụng Builder Pattern

Tạo một đối tượng phức tạp: có nhiều thuộc tính (nhiều hơn 4) và một số bắt buộc (requried), một số không bắt buộc (optional). Khi có quá nhiều hàm constructor, bạn nên nghĩ đến Builder.

Muốn tách rời quá trình xây dựng một đối tượng phức tạp từ các phần tạo nên đối tượng. Muốn kiểm soát quá trình xây dựng.

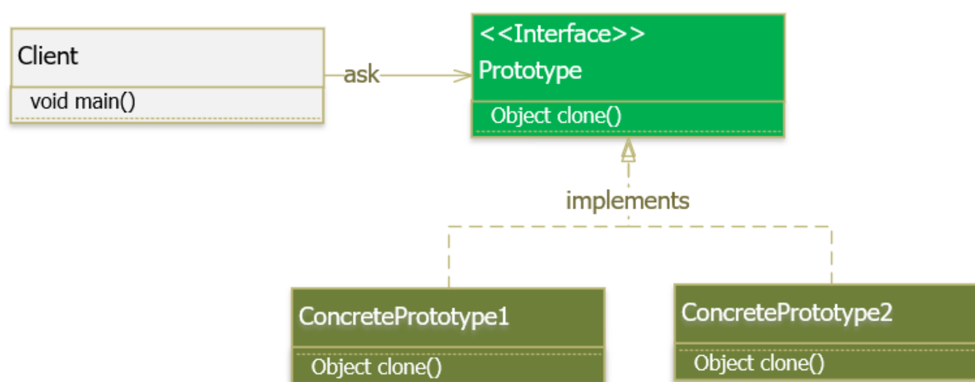
Khi người dùng (client) mong đợi nhiều cách khác nhau cho đối tượng được xây dựng.

2.1.5 Prototype

2.1.5.1 Giới thiệu về Prototype Pattern

Nó có nhiệm vụ khởi tạo một đối tượng bằng cách clone một đối tượng đã tồn tại thay vì khởi tạo với từ khoá new. Đối tượng mới là một bản sao có thể giống 100% với đối tượng gốc, chúng ta có thể thay đổi dữ liệu của nó mà không ảnh hưởng đến đối tượng gốc. Prototype Pattern được dùng khi việc tạo một object tốn nhiều chi phí và thời gian trong khi bạn đã có một object tương tự tồn tại.

2.1.5.2 Cài đặt Prototype Pattern



Hình 2 - 4: Sơ đồ UML mô tả Prototype Pattern

Một Prototype Pattern gồm các thành phần cơ bản sau:

- **Prototype** : khai báo một class, interface hoặc abtract class cho việc clone chính nó.
- **ConcretePrototype** class : các lớp này thực thi interface (hoặc kế thừa từ lớp abstract) được cung cấp bởi Prototype để copy (nhân bản) chính bản thân nó. Các lớp này chính là thể hiện cụ thể phương thức clone(). Lớp này có thể không cần thiết nếu: Prototype là một class và nó đã implement việc clone chính nó.
- **Client** class : tạo mới object bằng cách gọi Prototype thực hiện clone chính nó.

2.1.5.3 Lợi ích của Prototype Pattern

Cải thiện hiệu suất: giảm chi phí để tạo ra một đối tượng mới theo chuẩn, điều này sẽ làm tăng hiệu suất so với việc sử dụng từ khóa new để tạo đối tượng mới.

Giảm độ phức tạp cho việc khởi tạo đối tượng: do mỗi lớp chỉ implement cách clone của chính nó. Giảm việc phân lớp, tránh việc tạo nhiều lớp con cho việc khởi tạo đối tượng như của Abstract Factory Pattern.

2.1.5.4 Sử dụng Prototype

Có một object và cần phải tạo 1 object mới khác dựa trên object bạn đầu mà không thể sử dụng toán tử new hay các hàm constructor để khởi tạo. Lý do đơn giản là ở đây chúng ta ko hề được biết thông tin nội tại của object đó hoặc object đó đã có thể bị che dấu đi nhiều thông tin khác mà chỉ cho ta một thông tin rất giới hạn không đủ để hiểu được. Do vậy ta ko thể dùng toán tử new để khởi tạo nó được. Giải pháp: để

cho chính object mẫu tự xác định thông tin và dữ liệu sao chép.

Khởi tạo đối tượng lúc run-time: chúng ta có thể xác định đối tượng cụ thể sẽ được khởi tạo lúc runtime nếu class được implement / extend từ một Prototype.

Muốn truyền đối tượng vào một hàm nào đó để xử lý, thay vì truyền đối tượng gốc có thể ảnh hưởng dữ liệu thì ta có thể truyền đối tượng sao chép.

2.2 Nhóm Structural

2.2.1 Adapter

2.2.1.1 Giới thiệu Adapter Pattern

Adapter Pattern cho phép các inteface (giao diện) không liên quan tới nhau có thể làm việc cùng nhau. Đối tượng giúp kết nối các interface gọi là Adapter. Adapter Pattern giữ vai trò trung gian giữa hai lớp, chuyển đổi interface của một hay nhiều lớp có sẵn thành một interface khác, thích hợp cho lớp đang viết. Điều này cho phép các lớp có các interface khác nhau có thể dễ dàng giao tiếp tốt với nhau thông qua interface trung gian, không cần thay đổi code của lớp có sẵn cũng như lớp đang viết.

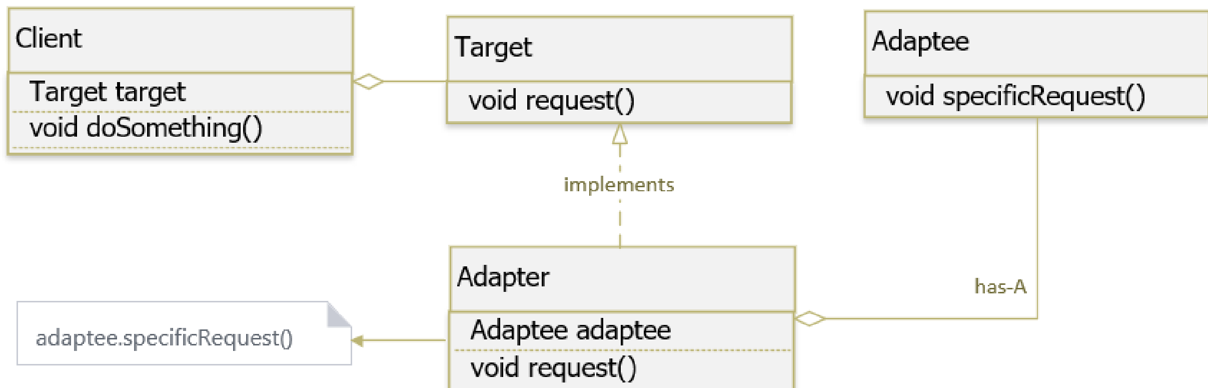
2.2.1.2 Cài đặt Adapter Pattern

Một Adapter Pattern bao gồm các thành phần cơ bản sau:

- **Adaptee**: định nghĩa interface không tương thích, cần được tích hợp vào.
- **Adapter**: lớp tích hợp, giúp interface không tương thích tích hợp được với interface đang làm việc. Thực hiện việc chuyển đổi interface cho Adaptee và kết nối Adaptee với Client.
- **Target**: một interface chứa các chức năng được sử dụng bởi Client (domain specific).
- **Client**: lớp sử dụng các đối tượng có interface Target.

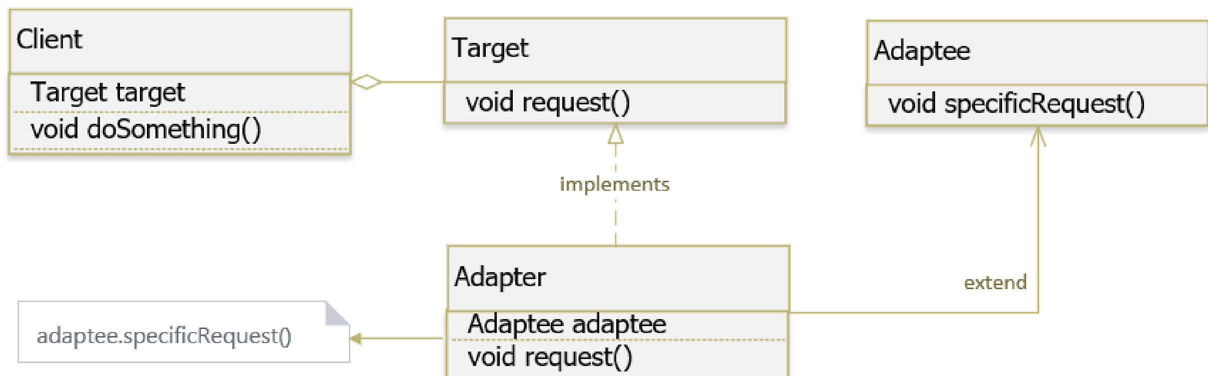
Có hai cách để thực hiện Adapter Pattern dựa theo cách cài đặt (implement) của chúng:

- Object Adapter – Composition (Tổng hợp): trong mô hình này, một lớp mới (Adapter) sẽ tham chiếu đến một (hoặc nhiều) đối tượng của lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới này, khi cài đặt các phương thức của interface người dùng mong muốn, sẽ gọi phương thức cần thiết thông qua đối tượng thuộc lớp có interface không tương thích.



Hình 2 – 5: Sơ đồ UML cách cài đặt Object Pattern

- Class Adapter – Inheritance (Kế thừa) : trong mô hình này, một lớp mới (Adapter) sẽ kế thừa lớp có sẵn với interface không tương thích (Adaptee), đồng thời cài đặt interface mà người dùng mong muốn (Target). Trong lớp mới, khi cài đặt các phương thức của interface người dùng mong muốn, phương thức này sẽ gọi các phương thức cần thiết mà nó thừa kế được từ lớp có interface không tương thích.



Hình 2 – 6: Sơ đồ UML cách cài đặt Class Pattern

So sánh Class Adapter với Object Adapter:

- Sự khác biệt chính là Class Adapter sử dụng Inheritance (kế thừa) để kết nối Adapter và Adaptee trong khi Object Adapter sử dụng Composition (tổng hợp) để kết nối Adapter và Adaptee.
- Trong cách tiếp cận Class Adapter, nếu một Adaptee là một class và không phải là một interface thì Adapter sẽ là một lớp con của Adaptee. Do đó, nó sẽ không phục vụ tất cả các lớp con khác theo cùng một cách vì Adapter là một lớp phụ cụ thể của Adaptee.

Tại sao Object Adapter lại tốt hơn?

- Nó sử dụng Composition để giữ một thể hiện của Adaptee, cho phép một Adapter hoạt động với nhiều Adaptee nếu cần thiết.

2.2.1.3 Lợi ích của Adapter Pattern

Cho phép nhiều đối tượng có interface giao tiếp khác nhau có thể tương tác và giao tiếp với nhau. Tăng khả năng sử dụng lại thư viện với interface không thay đổi do không có mã nguồn.

Bên cạnh những lợi ích trên, nó cũng nó một số khuyết điểm nhỏ sau:

- Tất cả các yêu cầu được chuyển tiếp, do đó làm tăng thêm một ít chi phí.
- Đôi khi có quá nhiều Adapter được thiết kế trong một chuỗi Adapter (chain) trước khi đến được yêu cầu thực sự.

2.2.1.4 Sử dụng Adapter Pattern

Adapter Pattern giúp nhiều lớp có thể làm việc với nhau dễ dàng mà bình thường không thể. Một trường hợp thường gặp phải và có thể áp dụng Adapter Pattern là khi không thể kế thừa lớp A, nhưng muốn một lớp B có những xử lý tương tự như lớp A. Khi đó chúng ta có thể cài đặt B theo Object Adapter, các xử lý của B sẽ gọi những xử lý của A khi cần.

Khi muốn sử dụng một lớp đã tồn tại trước đó nhưng interface sử dụng không phù hợp như mong muốn.

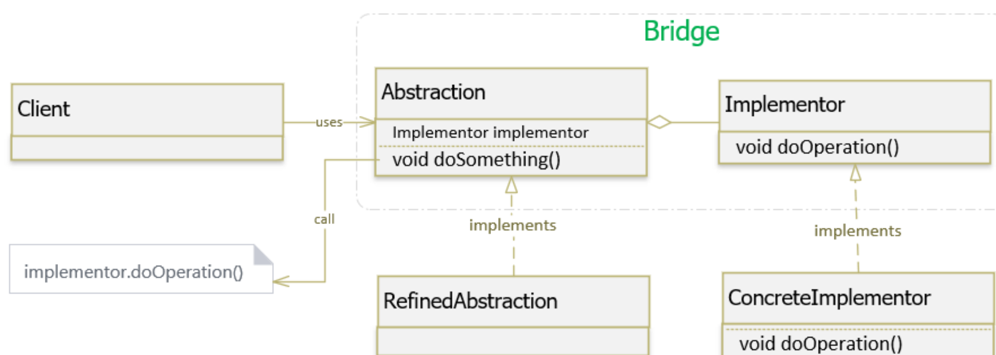
Khi muốn tạo ra những lớp có khả năng sử dụng lại, chúng phối hợp với các lớp không liên quan hay những lớp không thể đoán trước được và những lớp này không có những interface tương thích.

2.2.2 Bridge

2.2.2.1 Giới thiệu về Bridge Pattern

Ý tưởng của nó là tách tính trừu tượng (abstraction) ra khỏi tính hiện thực (implementation) của nó. Từ đó có thể dễ dàng chỉnh sửa hoặc thay thế mà không làm ảnh hưởng đến những nơi có sử dụng lớp ban đầu. Điều đó có nghĩa là, ban đầu chúng ta thiết kế một class với rất nhiều xử lý, bây giờ chúng ta không muốn để những xử lý đó trong class đó nữa. Vì thế, chúng ta sẽ tạo ra một class khác và move các xử lý đó qua class mới. Khi đó, trong lớp cũ sẽ giữ một đối tượng thuộc về lớp mới, và đối tượng này sẽ chịu trách nhiệm xử lý thay cho lớp ban đầu.

2.2.2.2 Cài đặt Bridge Pattern



Hình 2 – 7: Sơ đồ UML mô tả Bridge Pattern

Một Bridge Pattern bao gồm các thành phần sau:

- **Client**: đại diện cho khách hàng sử dụng các chức năng thông qua Abstraction.
- **Abstraction**: định ra một abstract interface quản lý việc tham chiếu đến đối tượng hiện thực cụ thể (Implementor).
- **Refined Abstraction (AbstractionImpl)**: hiện thực (implement) các phương thức đã được định ra trong Abstraction bằng cách sử dụng một tham chiếu đến một đối tượng của Implementer.
- **Implementor**: định ra các interface cho các lớp hiện thực. Thông thường nó là interface định ra các tác vụ nào đó của Abstraction.
- **ConcreteImplementor**: hiện thực Implementor interface.

2.2.2.3 Lợi ích của Bridge Pattern

Giảm sự phụ thuộc giữa abstraction và implementation (loose coupling): tính kế thừa trong OOP thường gắn chặt abstraction và implementation lúc build chương trình. Bridge Pattern có thể được dùng để cắt đứt sự phụ thuộc này và cho phép chúng ta chọn implementation phù hợp lúc runtime.

Giảm số lượng những lớp con không cần thiết: một số trường hợp sử dụng tính inheritance sẽ tăng số lượng subclass rất nhiều.

Code sẽ gọn gàn hơn và kích thước ứng dụng sẽ nhỏ hơn: do giảm được số class không cần thiết.

Dễ bảo trì hơn: các Abstraction và Implementation của nó sẽ dễ dàng thay đổi lúc runtime cũng như khi cần thay đổi thêm bớt trong tương lai.

Dễ dàng mở rộng về sau: thông thường các ứng dụng lớn thường yêu cầu chúng ta thêm module cho ứng dụng có sẵn nhưng không được sửa đổi framework/ứng dụng có sẵn vì các framework/ứng dụng đó có thể được công ty nâng cấp lên version mới. Bridge Pattern sẽ giúp chúng ta trong trường hợp này.

2.2.2.4 Sử dụng Bridge Pattern

Khi bạn muốn tách ràng buộc giữa Abstraction và Implementation, để có thể dễ dàng mở rộng độc lập nhau.

Cả Abstraction và Implementation của chúng nên được mở rộng bằng subsclass.

Sử dụng ở những nơi mà những thay đổi được thực hiện trong implement không ảnh hưởng đến phía client.

2.2.3 Composite

2.2.3.1 Giới thiệu về Composite Pattern

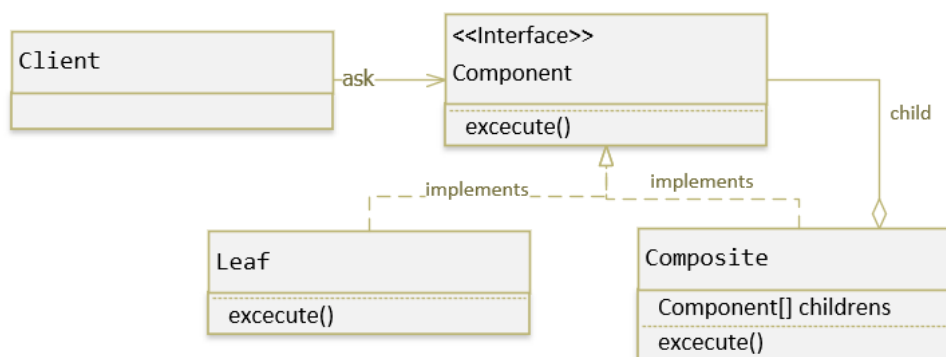
Là một sự tổng hợp những thành phần có quan hệ với nhau để tạo ra thành phần lớn hơn. Nó cho phép thực hiện các tương tác với tất cả đối tượng trong mẫu tương tự nhau. Composite Pattern được sử dụng khi chúng ta cần xử lý một nhóm đối tượng tương tự theo cách xử lý 1 object. Composite pattern sắp xếp các object theo cấu trúc cây để diễn giải 1 phần cũng như toàn bộ hệ thống phân cấp. Pattern này tạo một lớp chứa nhóm đối tượng của riêng nó. Lớp này cung cấp các cách để sửa đổi nhóm của cùng 1 object. Pattern này cho phép Client có thể viết code giống nhau để tương tác với composite object này, bất kể đó là một đối tượng riêng lẻ hay tập hợp các đối tượng.

2.2.3.2 Cài đặt Composite Pattern

Một Composite Pattern bao gồm các thành phần cơ bản sau:

- **Base Component**: là một interface hoặc abstract class quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
- **Leaf**: là lớp hiện thực (implements) các phương thức của Component. Nó là các object không có con.
- **Composite**: lưu trữ tập hợp các Leaf và cài đặt các phương thức của Base Component. Composite cài đặt các phương thức được định nghĩa trong interface Component bằng cách ủy nhiệm cho các thành phần con xử lý.
- **Client:** sử dụng Base Component để làm việc với các đối tượng

trong Composite.



Hình 2 – 8: Sơ đồ UML mô tả Composite Pattern

2.2.3.3 Lợi ích của Composite Pattern

Cung cấp cùng một cách sử dụng đối với từng đối tượng riêng lẻ hoặc nhóm các đối tượng với nhau.

2.2.3.4 Sử dụng Composite Pattern

Composite Pattern chỉ nên được áp dụng khi nhóm đối tượng phải hoạt động như một đối tượng duy nhất (theo cùng một cách).

Composite Pattern có thể được sử dụng để tạo ra một cấu trúc giống như cấu trúc cây.

2.2.4 Decorator

2.2.4.1 Giới thiệu về Decorator Pattern

Nó cho phép người dùng thêm chức năng mới vào đối tượng hiện tại mà không muốn ảnh hưởng đến các đối tượng khác. Kiểu thiết kế này có cấu trúc hoạt động như một lớp bao bọc (wrap) cho lớp hiện có. Mỗi khi cần thêm tính năng mới, đối tượng hiện có được wrap trong một đối tượng mới (decorator class). Decorator pattern sử dụng composition thay vì inheritance (thừa kế) để mở rộng đối tượng. Decorator pattern còn được gọi là Wrapper hay Smart Proxy.

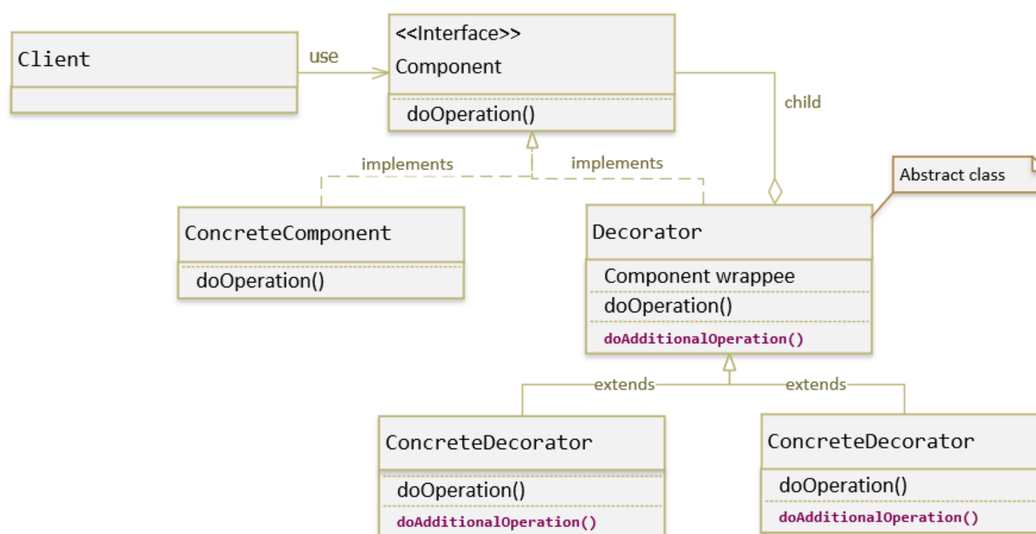
2.2.4.2 Cài đặt Decorator Pattern

Decorator Pattern hoạt động dựa trên một đối tượng đặc biệt được gọi là decorator (wrapper). Nó có cùng một interface như một đối tượng mà nó cần bao bọc (wrap), vì vậy phía client sẽ không nhận thấy khi bạn đưa cho nó một wrapper thay vì đối tượng gốc.

Tất cả các wrapper có một trường để lưu trữ một giá trị của một đối tượng gốc.

Hầu hết các wrapper khởi tạo trường đó với một đối tượng được truyền vào constructor của chúng.

Vậy làm thế nào để có thể thay đổi hành vi của đối tượng? Như đã đề cập, wrapper có cùng interface với các đối tượng đích. Khi bạn gọi một phương thức decorator, nó thực hiện cùng một phương thức trong một đối tượng được wrap và sau đó thêm một cái gì đó (tính năng mới) vào kết quả, công việc này tùy thuộc vào logic nghiệp vụ.



Hình 2 – 9: Sơ đồ UML mô tả Decorator Pattern

Các thành phần trong mẫu thiết kế Decorator:

- **Component**: là một interface quy định các method chung cần phải có cho tất cả các thành phần tham gia vào mẫu này.
- **ConcreteComponent** : là lớp hiện thực (implements) các phương thức của Component.
- **Decorator** : là một abstract class dùng để duy trì một tham chiếu của đối tượng Component và đồng thời cài đặt các phương thức của Component interface.
- **ConcreteDecorator** : là lớp hiện thực (implements) các phương thức của Decorator, nó cài đặt thêm các tính năng mới cho Component.
- **Client** : đối tượng sử dụng Component.

2.2.4.3 Lợi ích của Decorator Pattern

Tăng cường khả năng mở rộng của đối tượng, bởi vì những thay đổi được thực

hiện bằng cách implement trên các lớp mới.

Client sẽ không nhận thấy sự khác biệt khi bạn đưa cho nó một wrapper thay vì đối tượng gốc.

Một đối tượng có thể được bao bọc bởi nhiều wrapper cùng một lúc.

Cho phép thêm hoặc xóa tính năng của một đối tượng lúc thực thi (run-time).

2.2.4.4 Sử dụng Decorator Pattern

Khi muốn thêm tính năng mới cho các đối tượng mà không ảnh hưởng đến các đối tượng này.

Khi không thể mở rộng một đối tượng bằng cách thừa kế (inheritance). Chẳng hạn, một class sử dụng từ khóa final, muốn mở rộng class này chỉ còn cách duy nhất là sử dụng decorator.

Trong một số nhiều trường hợp mà việc sử dụng kế thừa sẽ mất nhiều công sức trong việc viết code. Ví dụ trên là một trong những trường hợp như vậy.

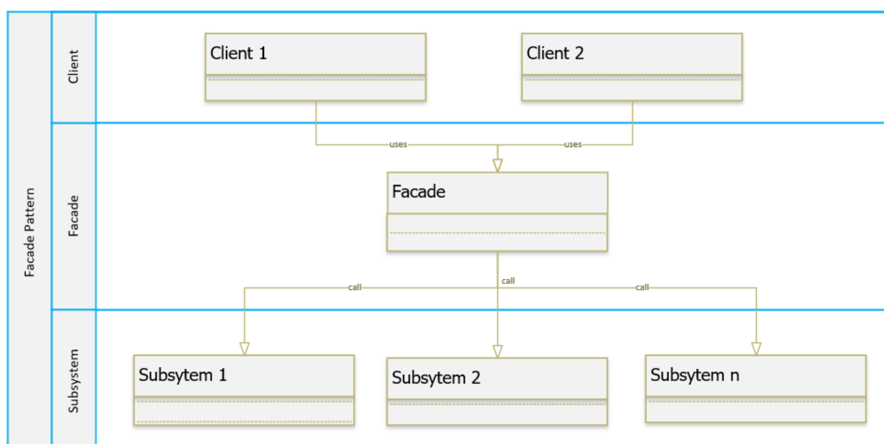
2.2.5 Facade

2.2.5.1 Giới thiệu về Facede Pattern

Cung cấp một giao diện chung đơn giản thay cho một nhóm các giao diện có trong một hệ thống con (subsystem). Facade 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.

Facade Pattern cho phép các đối tượng truy cập trực tiếp giao diện chung này để giao tiếp với các giao diện có trong hệ thống con. Mục tiêu là che giấu các hoạt động phức tạp bên trong hệ thống con, làm cho hệ thống con dễ sử dụng hơn.

2.2.5.2 Cài đặt Facade Pattern



Hình 2 – 10: Sơ đồ UML mô tả Facede Pattern

Các thành phần cơ bản của một Facade Pattern:

- **Facade**: biết rõ lớp của hệ thống con nào đảm nhận việc đáp ứng yêu cầu của client, sẽ chuyển yêu cầu của client đến các đối tượng của hệ thống con tương ứng.
- **Subsystems**: cài đặt các chức năng của hệ thống con, xử lý công việc được gọi bởi Facade. Các lớp này không cần biết Facade và không tham chiếu đến nó.
- **Client**: đối tượng sử dụng Facade để tương tác với các subsystem.

Các đối tượng Facade thường là Singleton bởi vì chỉ cần duy nhất một đối tượng Facade.

2.2.5.3 Lợi ích của Facade Pattern

Giúp cho hệ thống của bạn trở nên đơn giản hơn trong việc sử dụng và trong việc hiểu nó, vì một mẫu Facade có các phương thức tiện lợi cho các tác vụ chung.

Giảm sự phụ thuộc của các mã code bên ngoài với hiện thực bên trong của thư viện, vì hầu hết các code đều dùng Facade, vì thế cho phép sự linh động trong phát triển các hệ thống.

2.2.5.4 Sử dụng Facade Pattern

Khi hệ thống có rất nhiều lớp làm người sử dụng rất khó để có thể hiểu được quy trình xử lý của chương trình. Và khi có rất nhiều hệ thống con mà mỗi hệ thống con

đó lại có những giao diện riêng lẻ của nó nên rất khó cho việc sử dụng phối hợp. Khi đó có thể sử dụng Facade Pattern để tạo ra một giao diện đơn giản cho người sử dụng một hệ thống phức tạp.

Khi người sử dụng phụ thuộc nhiều vào các lớp cài đặt. Việc áp dụng Façade Pattern sẽ tách biệt hệ thống con của người dùng và các hệ thống con khác, do đó tăng khả năng độc lập và khả chuyển của hệ thống con, dễ chuyển đổi nâng cấp trong tương lai.

Khi bạn muốn phân lớp các hệ thống con. Dùng Façade Pattern để định nghĩa cổng giao tiếp chung cho mỗi hệ thống con, do đó giúp giảm sự phụ thuộc của các hệ thống con vì các hệ thống này chỉ giao tiếp với nhau thông qua các cổng giao diện chung đó.

Khi bạn muốn bao bọc, che giấu tính phức tạp trong các hệ thống con đối với phía Client.

2.2.6 Flyweight

2.2.6.1 Giới thiệu về Flyweight Pattern

Nó cho phép tái sử dụng đối tượng tương tự đã tồn tại bằng cách lưu trữ chúng hoặc tạo đối tượng mới khi không tìm thấy đối tượng phù hợp. Flyweight Pattern được sử dụng khi chúng ta cần tạo một số lượng lớn các đối tượng của 1 lớp nào đó. Do mỗi đối tượng đều đòi hỏi chiếm giữ một khoảng không gian bộ nhớ, nên với một số lượng lớn đối tượng được tạo ra có thể gây nên vấn đề nghiêm trọng đặc biệt đối với các thiết bị có dung lượng nhớ thấp. Flyweight Pattern có thể được áp dụng để giảm tải cho bộ nhớ thông qua cách chia sẻ các đối tượng. Vì vậy performance của hệ thống được tối ưu.

2.2.6.2 Hai trạng thái của Flyweight Object

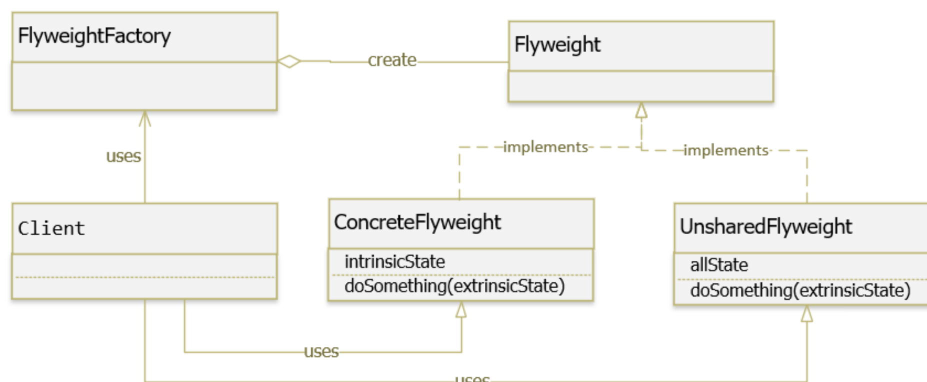
Trạng thái của flyweight object là một phần quan trọng trong việc thiết kế Flyweight Pattern. Mục tiêu chính của Flyweight Pattern là giảm bộ nhớ bằng cách chia sẽ các đối tượng. Điều này có thể đạt được bằng cách tách các thuộc tính của đối tượng thành hai trạng thái: độc lập và phụ thuộc. Hay còn gọi là **Intrinsic** (trạng thái nội tại) và **Extrinsic** (trạng thái bên ngoài).

- **Intrinsic State (trạng thái nội tại)** : Trạng thái này chứa dữ liệu không thể thay đổi (unchangeable) và không phụ thuộc (independent) vào ngữ cảnh (context) của đối tượng Flyweight . Những dữ liệu đó có thể được lưu trữ vĩnh viễn bên trong đối tượng Flyweight. Vì vậy mà Flyweight

object có thể chia sẻ. Dữ liệu nội tại là phi trạng thái (stateless) và thường không thay đổi (unchanged). Tính năng này cho phép khả năng tái tạo các thuộc tính đối tượng Flyweight giữa các đối tượng tương tự khác. Điều quan trọng cần lưu ý là các đối tượng Flyweight chỉ nên nhận trạng thái bên trong của chúng thông qua các tham số của hàm tạo và không cung cấp các phương thức setter hay các biến public.

- **Extrinsic State (trạng thái bên ngoài)** : Trạng thái bên ngoài thể hiện tính chất phụ thuộc ngữ cảnh của đối tượng flyweight. Trạng thái này chứa các thuộc tính và dữ liệu được áp dụng hoặc được tính toán trong thời gian thực thi (runtime). Do đó, những dữ liệu đó không được lưu trữ trong bộ nhớ. Vì trạng thái bên ngoài là phụ thuộc ngữ cảnh và có thể thay đổi nên các đối tượng đó không thể được chia sẻ. Do đó, client chịu trách nhiệm truyền dữ liệu liên quan đến trạng thái bên ngoài cho đối tượng flyweight khi cần thiết, có thể thông qua các tham số (argument).

2.2.6.3 Cài đặt Flyweight Pattern



Hình 2 – 11: Sơ đồ UML mô tả Flyweight Pattern

Các thành phần trong mẫu thiết kế Flyweight:

- **Flyweight** : là một interface/ abstract class, định nghĩa các các thành phần của một đối tượng.
- **ConcreteFlyweight** : triển khai các phương thức đã được định nghĩa trong Flyweight. Việc triển khai này phải thực hiện các khả năng của trạng thái nội tại. Đó là dữ liệu phải không thể thay đổi (unchangeable) và có thể chia sẻ (shareable). Các đối tượng là phi trạng thái (stateless) trong triển khai này. Vì vậy, đối tượng ConcreteFlyweight giống nhau có thể được sử dụng trong các ngữ cảnh khác nhau.

- **UnsharedFlyweight** : mặc dù mẫu thiết kế Flyweight cho phép chia sẻ thông tin, nhưng có thể tạo ra các thể hiện không được chia sẻ (not shared). Trong những trường hợp này, thông tin của các đối tượng có thể là stateful.
- **FlyweightFactory** (Cache): lớp này có thể là một Factory Pattern được sử dụng để giữ tham chiếu đến đối tượng Flyweight đã được tạo ra. Nó cung cấp một phương thức để truy cập đối tượng Flyweight được chia sẽ. FlyweightFactory bao gồm một Pool (có thể là HashMap, không cho phép bên ngoài truy cập vào) để lưu trữ đối tượng Flyweight trong bộ nhớ. Nó sẽ trả về đối tượng Flyweight đã tồn tại khi được yêu cầu từ Client hoặc tạo mới nếu không tồn tại.
- **Client** : sử dụng FlyweightFactory để khởi tạo đối tượng Flyweight.

2.2.6.4 Lợi ích của Flyweight Pattern

Giảm số lượng đối tượng được tạo ra bằng cách chia sẻ đối tượng. Vì vậy, tiết kiệm bộ nhớ và các thiết bị lưu trữ cần thiết.

Cãi thiện khả năng cache dữ liệu vì thời gian đáp ứng nhanh.

Tăng performance.

2.6.5 Sử dụng Flyweight Pattern

Khi có một số lớn các đối tượng được ứng dụng tạo ra một cách lặp đi lặp lại.

Khi việc tạo ra đối tượng đòi hỏi nhiều bộ nhớ và thời gian.

Khi muốn tái sử dụng đối tượng đã tồn tại thay vì phải tốn thời gian để tạo mới.

Khi nhóm đối tượng chứa nhiều đối tượng tương tự và hai đối tượng trong nhóm không khác nhau nhiều.

2.2.7 Proxy

2.2.7.1 Giới thiệu về Proxy Pattern

Proxy có nghĩa là "ủy quyền" hay "đại diện". Mục đích xây dựng Proxy pattern cũng chính vì muốn tạo ra một đối tượng sẽ ủy quyền, thay thế cho một đối tượng khác. Proxy Pattern là mẫu thiết kế mà ở đó tất cả các truy cập trực tiếp đến một đối tượng nào đó sẽ được chuyển hướng vào một đối tượng trung gian (Proxy Class). Mẫu Proxy (người đại diện) đại diện cho một đối tượng khác thực thi các phương thức, phương thức đó có thể được định nghĩa lại cho phù hợp với mục đích sử dụng. Để đơn giản hơn bạn có thể nghĩ đến khái niệm HTTP proxy trong mạng máy tính, nó là một

gateway giữa trình duyệt (client) và máy chủ (subject). HTTP proxy giúp nâng cao trải nghiệm người dùng, tăng tốc với lưu đệm các dữ liệu, loại bỏ các trang quảng cáo, giới hạn các vùng thông tin được xem… Proxy Pattern cũng có chung một mục đích như với HTTP proxy.

2.2.7.2 Phân loại Proxy

Virtual Proxy : Virtual Proxy tạo ra một đối tượng trung gian mỗi khi có yêu cầu tại thời điểm thực thi ứng dụng, nhờ đó làm tăng hiệu suất của ứng dụng.

Protection Proxy : Phạm vi truy cập của các client khác nhau sẽ khác nhau. Protection proxy sẽ kiểm tra các quyền truy cập của client khi có một dịch vụ được yêu cầu.

Remote Proxy : Client truy cập qua Remote Proxy để chiếu tới một đối tượng được bảo về nằm bên ngoài ứng dụng (trên cùng máy hoặc máy khác).

Monitor Proxy : Monitor Proxy sẽ thiết lập các bảo mật trên đối tượng cần bảo vệ, ngăn không cho client truy cập một số trường quan trọng của đối tượng. Có thể theo dõi, giám sát, ghi log việc truy cập, sử dụng đối tượng.

Firewall Proxy : bảo vệ đối tượng từ chối các yêu cầu xuất xứ từ các client không tín nhiệm.

Cache Proxy : Cung cấp không gian lưu trữ tạm thời cho các kết quả trả về từ đối tượng nào đó, kết quả này sẽ được tái sử dụng cho các client chia sẻ chung một yêu cầu gửi đến. Loại Proxy này hoạt động tương tự như Flyweight Pattern.

Smart Reference Proxy : Là nơi kiểm soát các hoạt động bổ sung mỗi khi đối tượng được tham chiếu.

Synchronization Proxy : Đảm bảo nhiều client có thể truy cập vào cùng một đối tượng mà không gây ra xung đột. Khi một client nào đó chiếm dụng khóa khá lâu khiến cho số lượng các client trong danh sách hàng đợi cứ tăng lên, và do đó hoạt động của hệ thống bị ngừng trệ, có thể dẫn đến hiện tượng "tắc nghẽn".

Copy-On-Write Proxy : Loại này đảm bảo rằng sẽ không có client nào phải chờ vô thời hạn. Copy-On-Write Proxy là một thiết kế rất phức tạp.

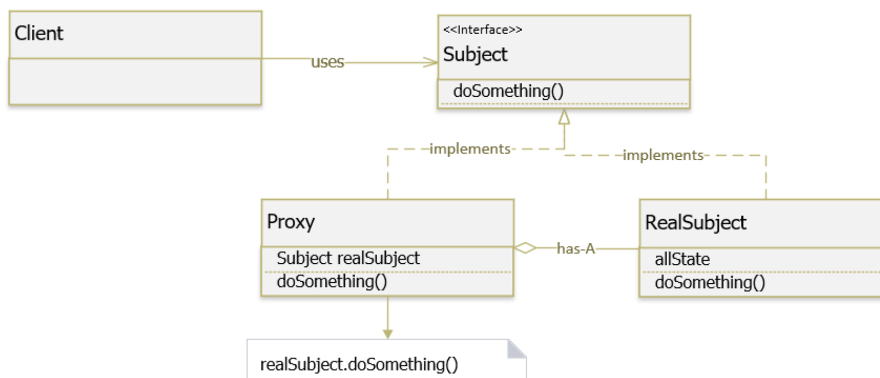
2.2.7.3 Cài đặt Proxy Pattern

Proxy Pattern có những đặc điểm chung sau đây:

- Cung cấp mức truy cập gián tiếp vào một đối tượng.
- Tham chiếu vào đối tượng đích và chuyển tiếp các yêu cầu đến đối tượng

đó.

- 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.



Hình 2 – 12: Sơ đồ UML mô tả Proxy Pattern

Các thành phần tham gia vào mẫu Proxy Pattern:

- **Subject** : là một interface định nghĩa các phương thực để giao tiếp với client. Đối tượng này xác định giao diện chung cho RealSubject và Proxy để Proxy có thể được sử dụng bất cứ nơi nào mà RealSubject mong đợi.
- **Proxy** : là một class sẽ thực hiện các bước kiểm tra và gọi tới đối tượng của class service thật để thực hiện các thao tác sau khi kiểm tra. Nó duy trì một tham chiếu đến RealSubject để Proxy có thể truy cập nó. Nó cũng thực hiện các giao diện tương tự như RealSubject để Proxy có thể được sử dụng thay cho RealSubject. Proxy cũng điều khiển truy cập vào RealSubject và có thể tạo hoặc xóa đối tượng này.
- **RealSubject** : là một class service sẽ thực hiện các thao tác thực sự. Đây là đối tượng chính mà proxy đại diện.
- **Client** : Đối tượng cần sử dụng RealSubject nhưng thông qua Proxy.

2.2.7.4 Lợi ích của Proxy Pattern

Cải thiện Performance thông qua lazy loading, chỉ tải các tài nguyên khi chúng được yêu cầu.

Nó cung cấp sự bảo vệ cho đối tượng thực từ thế giới bên ngoài.

Giảm chi phí khi có nhiều truy cập vào đối tượng có chi phí khởi tạo ban đầu lớn.

Dễ nâng cấp, bảo trì.

2.2.7.5 Sử dụng Proxy Pattern

Khi muốn bảo vệ quyền truy xuất vào các phương thức của object thực.

Khi cần một số thao tác bổ sung trước khi thực hiện phương thức của object thực.

Khi tạo đối tượng ban đầu là theo yêu cầu hoặc hệ thống yêu cầu sự chậm trễ khi tải một số tài nguyên nhất định (lazy loading).

Khi có nhiều truy cập vào đối tượng có chi phí khởi tạo ban đầu lớn.

Khi đối tượng gốc tồn tại trong môi trường từ xa (remote).

Khi đối tượng gốc nằm trong một hệ thống cũ hoặc thư viện của bên thứ ba.

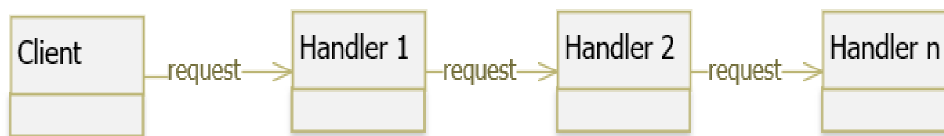
Khi muốn theo dõi trạng thái và vòng đời đối tượng.

2.3. Nhóm Behavioral

2.3.1 Chain of Responsibility

2.3.1.1 Giới thiệu về Chain of Responsibility Pattern

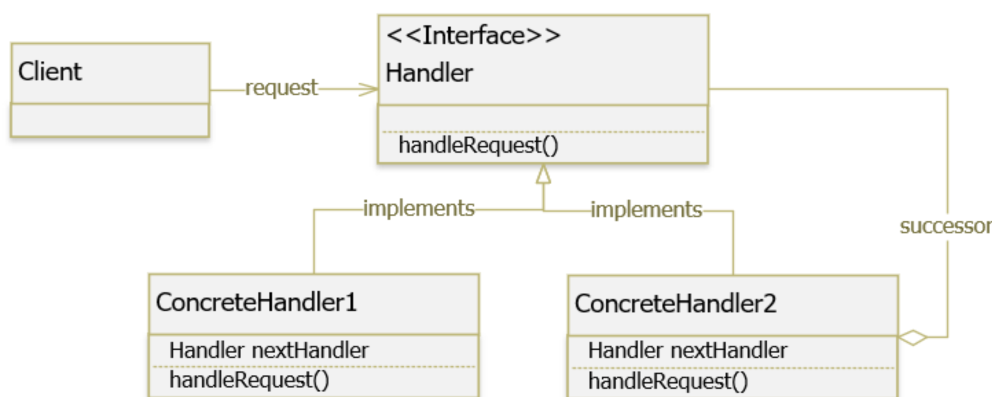
Chain of Responsiblity cho phép một đối tượng gửi một yêu cầu nhưng không biết đối tượng nào sẽ nhận và xử lý nó. Điều này được thực hiện bằng cách kết nối các đối tượng nhận yêu cầu thành một chuỗi (chain) và gửi yêu cầu theo chuỗi đó cho đến khi có một đối tượng xử lý nó.



Hình 2 – 13: Quy trình thực hiện của Chain of Responsibility Pattern

Chain of Responsibility Pattern hoạt động như một danh sách liên kết (Linked list) với việc đệ quy duyệt qua các phần tử (recursive traversal).

2.3.1.2 Cài đặt Chain of Responsibility Pattern



Hình 2 – 14: Sơ đồ UML mô tả Chain of Responsibility Pattern

Các thành phần tham gia mẫu Chain of Responsibility:

- **Handler** : định nghĩa 1 interface để xử lý các yêu cầu. Gán giá trị cho đối tượng successor (không bắt buộc).
- **ConcreteHandler** : xử lý yêu cầu. Có thể truy cập đối tượng successor (thuộc class Handler). Nếu đối tượng ConcreateHandler không thể xử lý được yêu cầu, nó sẽ gởi lời yêu cầu cho successor của nó.
- **Client** : tạo ra các yêu cầu và yêu cầu đó sẽ được gửi đến các đối tượng tiếp nhận.

Client gửi một yêu cầu để được xử lý gửi nó đến chuỗi (chain) các trình xử lý (handers), đó là các lớp mở rộng lớp Handler. Mỗi Hanlder trong chuỗi lần lượt cố gắng xử lý yêu cầu nhận được từ Client. Nếu trình xử lý đầu tiên (ConcreteHandler) có thể xử lý nó, thì yêu cầu sẽ được xử lý. Nếu không được xử lý thì sẽ gửi đến trình xử lý tiếp theo trong chuỗi (ConcreteHandler + 1).

2.3.1.3 Lợi ích của Chain of Responsibility Pattern

Giảm kết nối (loose coupling): 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. Tránh sự liên kết trực tiếp giữa đối tượng gửi yêu cầu (sender) và các đối tượng nhận yêu cầu (receivers).

Tăng tính linh hoạt : đảm bảo Open/Closed Principle.

Phân chia trách nhiệm cho các đối tượng: đảm bảo Single Responsibility Principle.

Có khả năng thay đổi dây chuyền (chain) trong thời gian chạy.

2.3.1.4 Sử dụng Chain of Responsibility Pattern

Có nhiều 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.

Khi cần phải thực thi các trình xử lý theo một thứ tự nhất định..

Khi một tập hợp các đối tượng xử lý có thể thay đổi động: tập hợp các đối tượng có khả năng xử lý yêu cầu có thể không biết trước, có thể thêm bớt hay thay đổi thứ tự sau này.

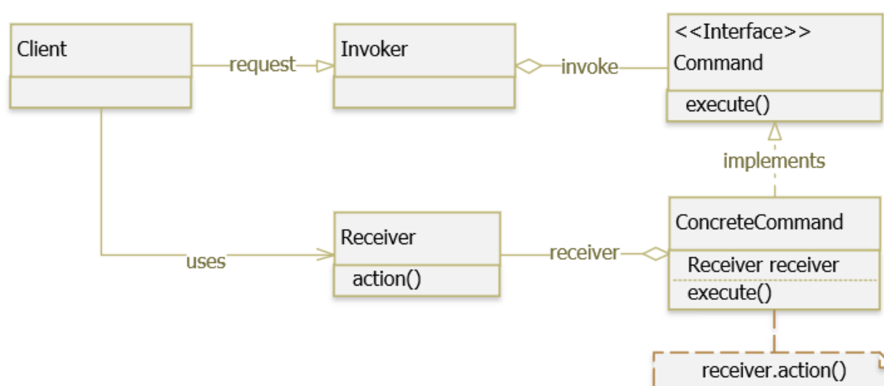
2.3.2 Command

2.3.2.1 Giới thiệu về Command Pattern

Nó cho phép chuyển yêu cầu thành đối tượng độc lập, có thể được sử dụng để tham số hóa các đối tượng với các yêu cầu khác nhau như log, queue (undo / redo), transtraction. Nói cho dễ hiểu, Command Pattern cho phép tất cả những Request gửi đến object được lưu trữ trong chính object đó dưới dạng một object Command. Khái niệm Command Object giống như một class trung gian được tạo ra để lưu trữ các câu lệnh và trạng thái của object tại một thời điểm nào đó.

Command dịch ra nghĩa là ra lệnh. Commander nghĩa là chỉ huy, người này không làm mà chỉ ra lệnh cho người khác làm. Như vậy, phải có người nhận lệnh và thi hành lệnh. Người ra lệnh cần cung cấp một class đóng gói những mệnh lệnh. Người nhận mệnh lệnh cần phân biệt những interface nào để thực hiện đúng mệnh lệnh. Command Pattern còn được biết đến như là Action hoặc Transaction.

2.3.2.2 Cài đặt Command Pattern



Hình 2 – 15: Sơ đồ UML mô tả Command Pattern

Các thành phần tham gia trong Command Pattern:

- **Command** : là một interface hoặc abstract class, chứa một phương thức trừu tượng thực thi (execute) một hành động (operation). Request sẽ được đóng gói dưới dạng Command.
- **ConcreteCommand** : là các implementation của Command. Định nghĩa một sự gắn kết giữa một đối tượng Receiver và một hành động. Thực thi execute() bằng việc gọi operation đang hoãn trên Receiver. Mỗi một ConcreteCommand sẽ phục vụ cho một case request riêng.
- **Client** : tiếp nhận request từ phía người dùng, đóng gói request thành ConcreteCommand thích hợp và thiết lập receiver của nó.
- **Invoker** : tiếp nhận ConcreteCommand từ Client và gọi execute() của ConcreteCommand để thực thi request.
- **Receiver** : đây là thành phần thực sự xử lý business logic cho case request. Trong phương execute() của ConcreteCommand chúng ta sẽ gọi method thích hợp trong Receiver.

Như vậy, Client và Invoker sẽ thực hiện việc tiếp nhận request. Còn việc thực thi request sẽ do Command, ConcreteCommand và Receiver đảm nhận.

2.3.2.3 Lợi ích của Command Pattern

Dễ dàng thêm các Command mới trong hệ thống mà không cần thay đổi trong các lớp hiện có. Đảm bảo Open/Closed Principle.

Tách đối tượng gọi operation từ đối tượng thực sự thực hiện operation. Giảm kết nối giữa Invoker và Receiver.

Cho phép tham số hóa các yêu cầu khác nhau bằng một hành động để thực hiện.

Cho phép lưu các yêu cầu trong hàng đợi.

Đóng gói một yêu cầu trong một đối tượng. Dễ dàng chuyển dữ liệu dưới dạng đối tượng giữa các thành phần hệ thống.

2.3.2.4 Sử dụng Command Pattern

Khi cần tham số hóa các đối tượng theo một hành động thực hiện.

Khi cần tạo và thực thi các yêu cầu vào các thời điểm khác nhau.

Khi cần hỗ trợ tính năng undo, log , callback hoặc transaction.

2.3.3 Interpreter

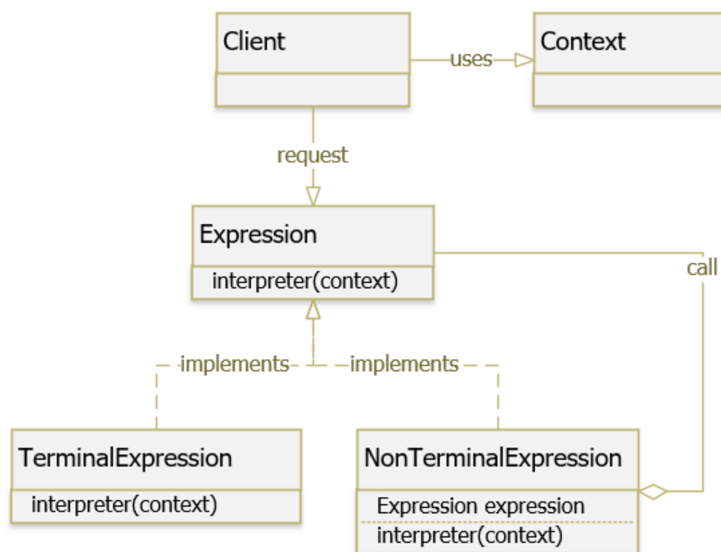
2.3.3.1 Giới thiệu về Interpreter Pattern

Interpreter nghĩa là thông dịch, mẫu này nói rằng "để xác định một biểu diễn ngữ pháp của một ngôn ngữ cụ thể, cùng với một thông dịch viên sử dụng biểu diễn này để diễn dịch các câu trong ngôn ngữ". Nói cho dễ hiểu, Interpreter Pattern giúp người lập trình có thể "xây dựng" những đối tượng "động" bằng cách đọc mô tả về đối tượng rồi sau đó "xây dựng" đối tượng đúng theo mô tả đó.

Metadata (mô tả) –> [Interpreter Pattern] –> Đối tượng tương ứng.

Interpreter Pattern có hạn chế về phạm vi áp dụng. Mẫu này thường được sử dụng để định nghĩa bộ ngữ pháp đơn giản (grammar), trong các công cụ quy tắc đơn giản (rule), …

2.3.3.2 Cài đặt Interpreter Pattern



Hình 2 – 16: Sơ đồ UML mô tả Interpreter Pattern

Các thành phần tham gia mẫu Interpreter:

- **Context** : là phần chứa thông tin biểu diễn mẫu chúng ta cần xây dựng.
- **Expression** : là một interface hoặc abstract class, định nghĩa phương thức interpreter chung cho tất cả các node trong cấu trúc cây phân tích ngữ pháp. Expression được biểu diễn như một cấu trúc cây phân cấp, mỗi implement của Expression có thể gọi một node.
- **TerminalExpression** (biểu thức đầu cuối): cài đặt các phương thức của Expression, là những biểu thức có thể được diễn giải trong một đối tượng duy nhất, chứa các xử lý logic để đưa thông tin của context thành đối tượng cụ thể.
- **NonTerminalExpression** (biểu thức không đầu cuối): cài đặt các phương thức của Expression, biểu thức này chứa một hoặc nhiều biểu thức khác nhau, mỗi biểu thức có thể là biểu thức đầu cuối hoặc không phải là biểu thức đầu cuối. Khi một phương thức interpret() của lớp biểu thức không phải là đầu cuối được gọi, nó sẽ gọi đệ quy đến tất cả các biểu thức khác mà nó đang giữ.
- **Client** : đại diện cho người dùng sử dụng lớp Interpreter Pattern. Client sẽ xây dựng cây biểu thức đại diện cho các lệnh được thực thi, gọi phương thức interpreter() của node trên cùng trong cây, có thể truyền context để thực thi tất cả các lệnh trong cây.

2.3.2.3 Lợi ích của Interpreter Pattern

Dễ dàng thay đổi và mở rộng ngữ pháp. Vì mẫu này sử dụng các lớp để biểu diễn các quy tắc ngữ pháp, chúng ta có thể sử dụng thừa kế để thay đổi hoặc mở rộng ngữ pháp. Các biểu thức hiện tại có thể được sửa đổi theo từng bước và các biểu thức mới có thể được định nghĩa lại các thay đổi trên các biểu thức cũ.

Cài đặt và sử dụng ngữ pháp rất đơn giản. Các lớp xác định các nút trong cây cú pháp có các implement tương tự. Các lớp này dễ viết và các phân cấp con của chúng có thể được tự động hóa bằng trình biên dịch hoặc trình tạo trình phân tích cú pháp.

2.3.2.4 Sử dụng Interpreter Pattern

Bộ ngữ pháp đơn giản. Pattern này cần xác định ít nhất một lớp cho mỗi quy tắc trong ngữ pháp. Do đó ngữ pháp có chứa nhiều quy tắc có thể khó quản lý và bảo trì.

Không quan tâm nhiều về hiệu suất. Do bộ ngữ pháp được phân tích trong cấu trúc phân cấp (cây) nên hiệu suất không được đảm bảo.

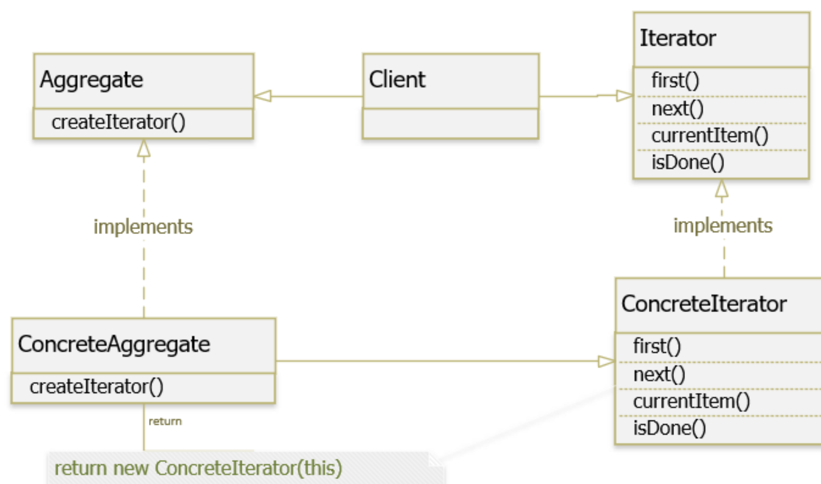
2.3.4 Iterator

2.3.4.1 Giới thiệu về Iterator Pattern

Nó được sử dụng để “Cung cấp một cách thức truy cập tuần tự tới các phần tử của một đối tượng tổng hợp, mà không cần phải tạo dựng riêng các phương pháp truy cập cho đối tượng tổng hợp này”. Nói cách khác, một Iterator được thiết kế cho phép xử lý nhiều loại tập hợp khác nhau bằng cách truy cập những phần tử của tập hợp với cùng một phương pháp, cùng một cách thức định sẵn, mà không cần phải hiểu rõ về những chi tiết bên trong của những tập hợp này.

Tách biệt trách nhiệm giữa các lớp rất hữu dụng khi một lớp bị thay đổi. Nếu có quá nhiều thứ bên trong một lớp đơn lẻ, sẽ rất khó khăn để viết lại mã nguồn. Khi diễn ra sự thay đổi, một lớp “đơn trách nhiệm” sẽ chỉ có một lý do duy nhất để thay đổi. Nó cung cấp một giao diện đơn giản, nhất quán để làm việc với các tập hợp khác nhau.

2.3.4.2 Cài đặt Iterator Pattern



Hình 2 – 17: Sơ đồ UML mô tả Iterator Pattern

Các thành phần tham gia mẫu Iterator:

- **Aggregate** : là một interface định nghĩa định nghĩa các phương thức để tạo Iterator object.
- **ConcreteAggregate** : cài đặt các phương thức của Aggregate, nó cài đặt interface tạo Iterator để trả về một thể hiện của ConcreteIterator thích hợp.
- **Iterator** : là một interface hay abstract class, định nghĩa các phương thức để truy cập và duyệt qua các phần tử.
- **ConcreteIterator** : cài đặt các phương thức của Iterator, giữ index khi duyệt qua các phần tử.
- **Client** : đối tượng sử dụng Iterator Pattern, nó yêu cầu một iterator từ một đối tượng collection để duyệt qua các phần tử mà nó giữ. Các phương thức của iterator được sử dụng để truy xuất các phần tử từ collection theo một trình tự thích hợp.

2.3.4.3 Lợi ích của Iterator Pattern

Đảm bảo nguyên tắc Single responsibility principle (SRP) : chúng ta có thể tách phần cài đặt các phương thức của tập hợp và phần duyệt qua các phần tử (iterator) theo từng class riêng lẻ.

Đảm bảo nguyên tắc Open/Closed Principle (OCP) : chúng ta có thể implement các loại collection mới và iterator mới, sau đó chuyển chúng vào code hiện có mà

không vi phạm bất cứ nguyên tắc gì.

Chúng ta có thể truy cập song song trên cùng một tập hợp vì mỗi đối tượng iterator có chứa trạng thái riêng của nó.

Một số điểm cần xem xét khi sử dụng Iterator:

- Sử dụng iterator có thể kém hiệu quả hơn so với việc duyệt qua các phần tử của bộ sưu tập một cách trực tiếp.
- Có thể không cần thiết nếu ứng dụng chỉ hoạt động với các collection đơn giản.

2.3.4.4 Sử dụng Iterator Pattern

Cần truy cập nội dung của đối tượng trong tập hợp mà không cần biết nội dung cài đặt bên trong nó.

Hỗ trợ truy xuất nhiều loại tập hợp khác nhau.

Cung cấp một interface duy nhất để duyệt qua các phần tử của một tập hợp.

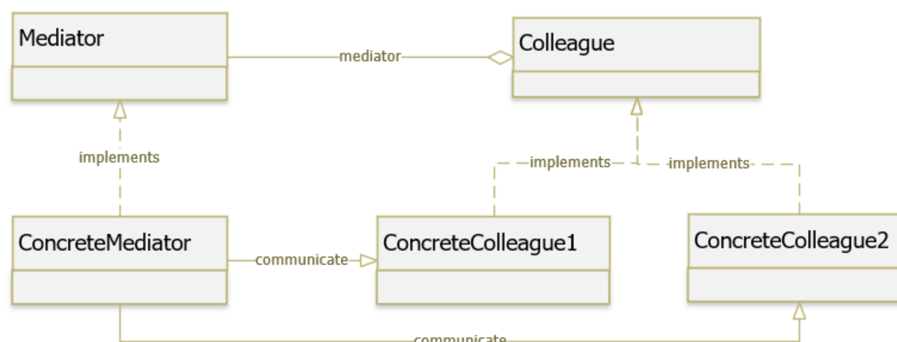
2.3.5 Mediator

2.3.5.1 Giới thiệu về Mediator Pattern

Mediator có nghĩa là người trung gian. Pattern này nói rằng "Định nghĩa một đối tượng gói gọn cách một tập hợp các đối tượng tương tác. Mediator thúc đẩy sự khớp nối lỏng lẻo bằng cách ngăn không cho các đối tượng đề cập đến nhau một cách rõ ràng và nó cho phép bạn thay đổi sự tương tác của họ một cách độc lập". Mediator Patern (mô hình trung gian) được sử dụng để giảm sự phức tạp trong "giao tiếp" giữa các lớp và các đối tượng. Mô hình này cung cấp một lớp trung gian có nhiệm vụ xử lý thông tin liên lạc giữa các tầng lớp, hỗ trợ bảo trì mã code dễ dàng bằng cách khớp nối lỏng lẻo.

Khớp nối lỏng lẻo ở đây được hiểu là các đối tượng tương đồng không "giao tiếp" trực tiếp với nhau mà giao tiếp thông qua người trung gian, và nó cho phép thay thay đổi cách tương tác giữa chúng một cách độc lập. Mediator Patern thúc đẩy mối quan hệ nhiều – nhiều (many-to-many) giữa các đối tượng tượng với nhau để đạt đến được kết quả mong muốn.

2.3.5.2 Cài đặt Mediator Pattern



Hình 2 – 18: Sơ đồ UML mô tả Mediator Pattern

Các thành phần tham gia Mediator Pattern:

- **Colleague** : là một abstract class, giữ tham chiếu đến Mediator object.
- **ConcreteColleague** : cài đặt các phương thức của Colleague. Giao tiếp thông qua Mediator khi cần giao tiếp với Colleague khác.
- **Mediator** : là một interface, định nghĩa các phương thức để giao tiếp với các Colleague object.
- **ConcreteMediator** : cài đặt các phương thức của Mediator, biết và quản lý các Colleague object.

2.3.5.3 Lợi ích của Mediator Pattern

Đảm bảo nguyên tắc Single responsibility principle (SRP) : chúng ta có thể tách phần giao tiếp giữa các thành phần (component) ra một nơi khác.

Đảm bảo nguyên tắc Open/Closed Principle (OCP) : chúng ta có thể implement thêm một Mediator mới mà không ảnh hưởng đến các component hiện có.

Tái sử dụng các component dễ dàng hơn.

Đơn giản hóa cách giao tiếp giữa các đối tượng. Một mediator sẽ thay thế mối quan hệ nhiều-nhiều (many-to-many) giữa các component bằng quan hệ một-nhiều (one-to-many) giữa một mediator với các component.

Quản lý tập trung, giúp làm rõ các component tương tác trong hệ thống như thế nào trong hệ thống.

2.3.5.4 Sử dụng Mediator Pattern

Khi tập hợp các đối tượng giao tiếp theo những cách thức được xác định rõ ràng nhưng cách thức đó quá phức tạp. Sự phụ thuộc lẫn nhau giữa các đối tượng tạo ra kết

quả là cách tổ chức không có cấu trúc và khó hiểu.

Khi cần tái sử dụng một đối tượng nhưng rất khó khăn vì nó tham chiếu và giao tiếp với nhiều đối tượng khác.

Thường được sử dụng trong các hệ thống truyền thông điệp (message-based system), chẳng hạn như hệ thống chat.

Khi giao tiếp giữa các object trong hệ thống quá phức tạp, có quá nhiều quan hệ giữa các object trong hệ thống. Một điểm chung để kiểm soát hoặc giao tiếp là cần thiết.

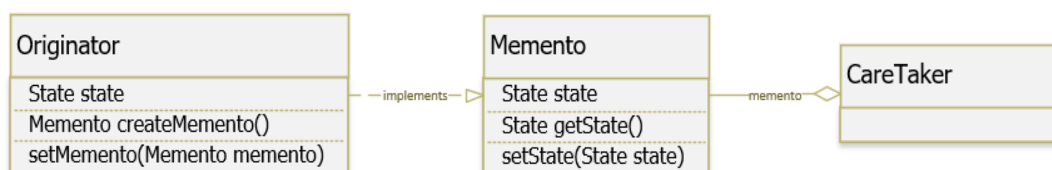
2.3.6 Memento

2.3.6.1 Giới thiệu về Memento Pattern

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. Dữ liệu trạng thái đã lưu trong đối tượng memento không thể truy cập bên ngoài đối tượng được lưu và khôi phục. Điều này bảo vệ tính toàn vẹn của dữ liệu trạng thái đã lưu.

Mẫu thiết kế Memento được sử dụng để thực hiện thao tác Undo. Điều này được thực hiện bằng cách lưu trạng thái hiện tại của đối tượng mỗi khi nó thay đổi trạng thái, từ đó chúng ta có thể khôi phục nó trong mọi trường hợp có lỗi.

2.3.6.2 Cài đặt Memento Pattern



Hình 2 – 19: Sơ đồ UML mô tả Memento Pattern

Các thành phần tham gia mẫu Memento:

- **Originator** : đại diện cho đối tượng mà chúng ta muốn lưu. Nó sử dụng memento để lưu và khôi phục trạng thái bên trong của nó.
- **Caretaker** : Nó không bao giờ thực hiện các thao tác trên nội dung của memento và thậm chí nó không kiểm tra nội dung. Nó giữ đối tượng memento và chịu trách nhiệm bảo vệ an toàn cho các đối tượng. Để khôi phục trạng thái trước đó, nó trả về đối tượng memento cho Originator.
- **Memento** : đại diện cho một đối tượng để lưu trữ trạng thái của Originator. Nó bảo vệ chống lại sự truy cập của các đối tượng khác ngoài

Originator.

- Lớp Memento cung cấp 2 interfaces: 1 interface cho Caretaker và 1 cho Originator. Interface Caretaker không được cho phép bất kỳ hoạt động hoặc bất kỳ quyền truy cập vào trạng thái nội bộ được lưu trữ bởi memento và do đó đảm bảo nguyên tắc đóng gói. Interface Originator cho phép nó truy cập bất kỳ biến trạng thái nào cần thiết để có thể khôi phục trạng thái trước đó.
- Lớp Memento thường là một lớp bên trong của Originator. Vì vậy, originator có quyền truy cập vào các trường của memento, nhưng các lớp bên ngoài không có quyền truy cập vào các trường này.

2.3.6.3 Lợi ích của Memento Pattern

Bảo bảo nguyên tắc đóng gói: sử dụng trực tiếp trạng thái của đối tượng có thể làm lộ thông tin chi tiết bên trong đối tượng và vi phạm nguyên tắc đóng gói.

Đơn giản code của Originator bằng cách để Memento lưu giữ trạng thái của Originator và Caretaker quản lý lịch sử thay đổi của Originator.

Một số vấn đề cần xem xét khi sử dụng Memento Pattern:

- Khi có một số lượng lớn Memento được tạo ra có thể gặp vấn đề về bộ nhớ, performance của ứng dụng.
- Khó đảm bảo trạng thái bên trong của Memento không bị thay đổi.

2.3.6.4 Sử dụng Memento Pattern

Các ứng dụng cần chức năng cần Undo/ Redo: lưu trạng thái của một đối tượng bên ngoài và có thể restore/ rollback sau này.

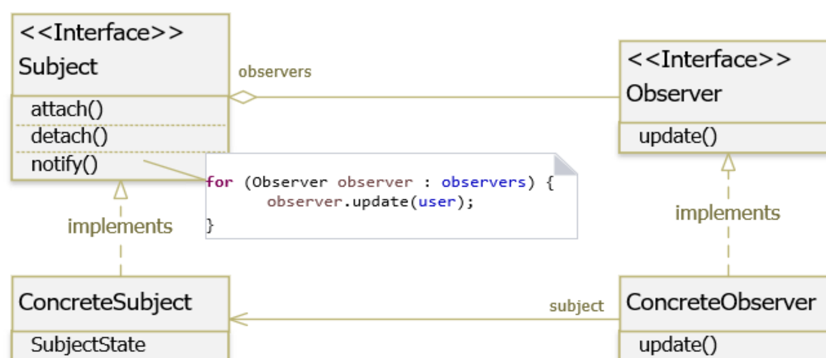
Thích hợp với các ứng dụng cần quản lý transaction.

2.3.7 Observer

2.3.7.1 Giới thiệu về Observer Pattern

Nó định nghĩa mối phụ thuộc một – nhiều giữa các đối tượng để khi mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động. Observer có thể đăng ký với hệ thống. Khi hệ thống có sự thay đổi, hệ thống sẽ thông báo cho Observer biết. Khi không cần nữa, mẫu Observer sẽ được gỡ khỏi hệ thống.

2.3.7.2 Cài đặt Observer Pattern



Hình 2 – 20: Sơ đồ UML mô tả Observer Pattern

Các thành phần tham gia Observer Pattern:

- **Subject** : chứa danh sách các observer, cung cấp phương thức để có thể thêm và loại bỏ observer.
- **Observer** : định nghĩa một phương thức update() cho các đối tượng sẽ được subject thông báo đến khi có sự thay đổi trạng thái.
- **ConcreteSubject** : cài đặt các phương thức của Subject, lưu trữ trạng thái danh sách các ConcreateObserver, gửi thông báo đến các observer của nó khi có sự thay đổi trạng thái.
- **ConcreteObserver** : cài đặt các phương thức của Observer, lưu trữ trạng thái của subject, thực thi việc cập nhật để giữ cho trạng thái đồng nhất với subject gửi thông báo đến.

Sự tương tác giữa subject và các observer như sau: mỗi khi subject có sự thay đổi trạng thái, nó sẽ duyệt qua danh sách các observer của nó và gọi phương thức cập nhật trạng thái ở từng observer, có thể truyền chính nó vào phương thức để các observer có thể lấy ra trạng thái của nó và xử lý.

2.3.7.3 Lợi ích của Observer Pattern

Dễ dàng mở rộng với ít sự thay đổi : mẫu này cho phép thay đổi Subject và Observer một cách độc lập. Chúng ta có thể tái sử dụng các Subject mà không cần tái sử dụng các Observer và ngược lại. Nó cho phép thêm Observer mà không sửa đổi Subject hoặc Observer khác. Vì vậy, nó đảm bảo nguyên tắc Open/Closed Principle (OCP).

Một đối tượng có thể thông báo đến một số lượng không giới hạn các đối tượng khác.

2.3.7.4 Sử dụng Observer Pattern

Thường được sử dụng trong mối quan hệ 1-n giữa các object với nhau. Trong đó một đối tượng thay đổi và muốn thông báo cho tất cả các object liên quan biết về sự thay đổi đó.

Khi thay đổi một đối tượng, yêu cầu thay đổi đối tượng khác và chúng ta không biết có bao nhiêu đối tượng cần thay đổi, những đối tượng này là ai.

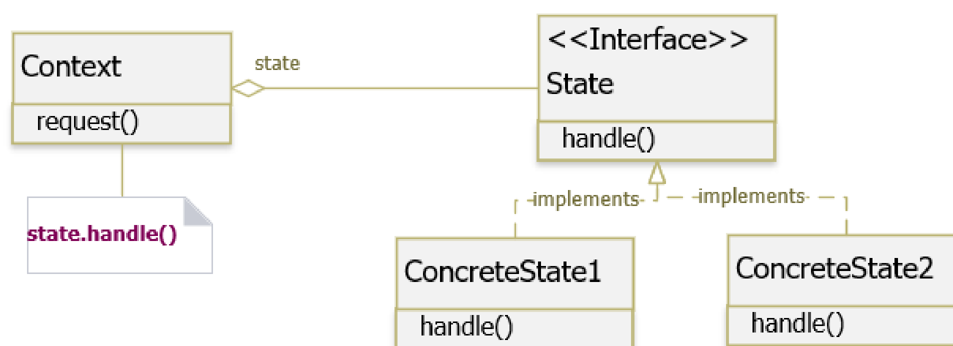
Sử dụng trong mẫu mô hình MVC (Model View Controller Pattern) : trong MVC, mẫu này được sử dụng để tách Model khỏi View. View đại diện cho Observer và Model là đối tượng Observable.

2.3.8 State

2.3.8.1 Giới thiệu State Pattern

Nó cho phép một đối tượng thay đổi hành vi của nó khi trạng thái nội bộ của nó thay đổi. Đối tượng sẽ xuất hiện để thay đổi lớp của nó.

2.3.8.2 Cài đặt State Pattern



Hình 2 – 21: Sơ đồ UML mô tả State Pattern

Các thành phần tham gia State Pattern:

- **Context** : được sử dụng bởi Client. Client không truy cập trực tiếp đến State của đối tượng. Lớp Context này chứa thông tin của ConcreteState object, cho hành vi nào tương ứng với trạng thái nào hiện đang được thực hiện.
- **State** : là một interface hoặc abstract class xác định các đặc tính cơ bản của tất cả các đối tượng ConcreteState. Chúng sẽ được sử dụng bởi đối tượng Context để truy cập chức năng có thể thay đổi.
- **ConcreteState** : cài đặt các phương thức của State. Mỗi ConcreteState có

thể thực hiện logic và hành vi của riêng nó tùy thuộc vào Context.

Một vài điểm cần ghi nhớ khi áp dụng pattern này:

- Một đối tượng nên thay đổi hành vi của nó khi trạng thái bên trong của nó thay đổi.
- Mỗi State nên được xác định độc lập.
- Thêm các trạng thái mới sẽ không làm ảnh hưởng đến các trạng thái hoặc chức năng khác.

2.3.8.3 Lợi ích của State Pattern

Đảm bảo nguyên tắc Single responsibility principle (SRP): tách biệt mỗi State tương ứng với 1 class riêng biệt.

Đảm bảo nguyên tắc Open/Closed Principle (OCP) : chúng ta có thể thêm một State mới mà không ảnh hưởng đến State khác hay Context hiện có.

Giữ hành vi cụ thể tương ứng với trạng thái.

Giúp chuyển trạng thái một cách rõ ràng.

2.3.8.4 Sử dụng State Pattern

Khi hành vi của đối tượng phụ thuộc vào trạng thái của nó và nó phải có khả năng thay đổi hành vi của nó lúc run-time theo trạng thái mới.

Khi nhiều điều kiện phức tạp buộc đối tượng phụ thuộc vào trạng thái của nó.

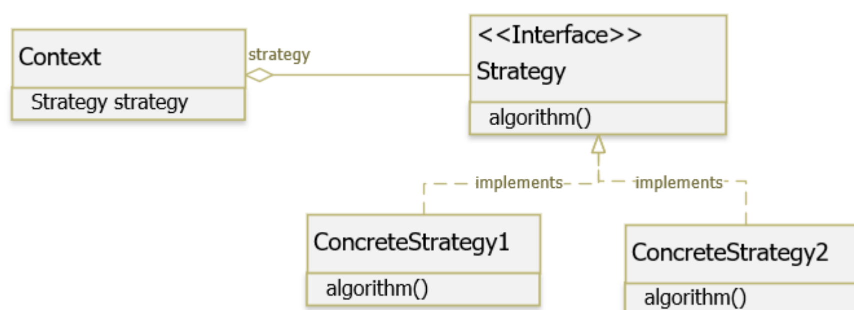
2.3.9 Strategy

2.3.9.1 Giới thiệu về Strategy Pattern

Nó cho phép định nghĩa tập hợp các thuật toán, đóng gói từng thuật toán lại, và dễ dàng thay đổi linh hoạt các thuật toán bên trong object. Strategy cho phép thuật toán biến đổi độc lập khi người dùng sử dụng chúng.

Ý nghĩa thực sự của Strategy Pattern là giúp tách rời phần xử lý một chức năng cụ thể ra khỏi đối tượng. Sau đó tạo ra một tập hợp các thuật toán để xử lý chức năng đó và lựa chọn thuật toán nào mà chúng ta thấy đúng đắn nhất khi thực thi chương trình. Mẫu thiết kế này thường được sử dụng để thay thế cho sự kế thừa, khi muốn chấm dứt việc theo dõi và chỉnh sửa một chức năng qua nhiều lớp con.

2.3.9.2 Cài đặt Strategy Pattern



Hình 2 – 21: Sơ đồ UML mô tả Strategy Pattern

Các thành phần tham gia Strategy Pattern:

- **Strategy** : định nghĩa các hành vi có thể có của một Strategy.
- **ConcreteStrategy** : cài đặt các hành vi cụ thể của Strategy.
- **Context** : chứa một tham chiếu đến đối tượng Strategy và nhận các yêu cầu từ Client, các yêu cầu này sau đó được ủy quyền cho Strategy thực hiện.

2.3.9.3 Lợi ích của Strategy Pattern

Đảm bảo nguyên tắc Single responsibility principle (SRP) : một lớp định nghĩa nhiều hành vi và chúng xuất hiện dưới dạng với nhiều câu lệnh có điều kiện. Thay vì nhiều điều kiện, chúng ta sẽ chuyển các nhánh có điều kiện liên quan vào lớp Strategy riêng lẻ của nó.

Đảm bảo nguyên tắc Open/Closed Principle (OCP) : chúng ta dễ dàng mở rộng và kết hợp hành vi mới mà không thay đổi ứng dụng.

Cung cấp một sự thay thế cho kế thừa.

2.3.9.4 Sử dụng Strategy Pattern

Khi muốn có thể thay đổi các thuật toán được sử dụng bên trong một đối tượng tại thời điểm run-time.

Khi có một đoạn mã dễ thay đổi, và muốn tách chúng ra khỏi chương trình chính để dễ dàng bảo trì.

Tránh sự rắc rối, khi phải hiện thực một chức năng nào đó qua quá nhiều lớp con.

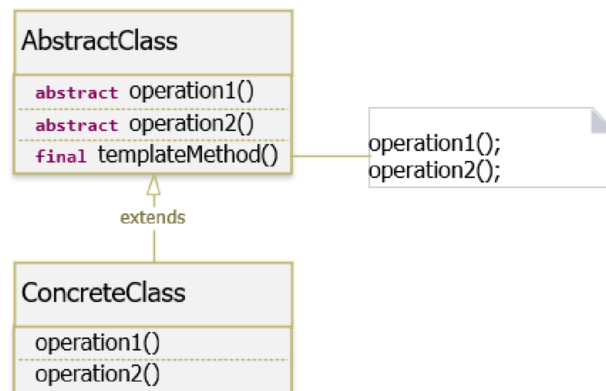
Cần che dấu sự phức tạp, cấu trúc bên trong của thuật toán.

2.3.10 Template Method

2.3.10.1 Giới thiệu về Template Method Pattern

Pattern này nói rằng "Định nghĩa một bộ khung của một thuật toán trong một chức năng, chuyển giao việc thực hiện nó cho các lớp con. Mẫu Template Method cho phép lớp con định nghĩa lại cách thực hiện của một thuật toán, mà không phải thay đổi cấu trúc thuật toán". Điều này có nghĩa là Template method giúp cho chúng ta tạo nên một bộ khung (template) cho một vấn đề đang cần giải quyết. Trong đó các đối tượng cụ thể sẽ có cùng các bước thực hiện, nhưng trong mỗi bước thực hiện đó có thể khác nhau. Điều này sẽ tạo nên một cách thức truy cập giống nhau nhưng có hành động và kết quả khác nhau.

2.3.10.2 Cài đặt Template Method Pattern



Hình 2 – 22: Sơ đồ UML mô tả Template Method Pattern

Các thành phần tham gia Template Method Pattern:

- AbstractClass :
  - Định nghĩa các phương thức trừu tượng cho từng bước có thể được điều chỉnh bởi các lớp con.
  - Cài đặt một phương thức duy nhất điều khiển thuật toán và gọi các bước riêng lẻ đã được cài đặt ở các lớp con.
- ConcreteClass : là một thuật toán cụ thể, cài đặt các phương thức của AbstractClass. Các thuật toán này ghi đè lên các phương thức trừu tượng để cung cấp các triển khai thực sự. Nó không thể ghi đè phương thức duy nhất đã được cài đặt ở AbstractClass (templateMethod).

2.3.10.3 Lợi ích của Template Method Pattern

Tái sử dụng code (reuse), tránh trùng lặp code (duplicate): đưa những phần trùng

lặp vào lớp cha (abstract class).

Cho phép người dùng override chỉ một số phần nhất định của thuật toán lớn, làm cho chúng ít bị ảnh hưởng hơn bởi những thay đổi xảy ra với các phần khác của thuật toán.

2.3.10.4 Sử dụng Template Method Pattern

Khi có một thuật toán với nhiều bước và mong muốn cho phép tùy chỉnh chúng trong lớp con.

Mong muốn chỉ có một triển khai phương thức trừu tượng duy nhất của một thuật toán.

Mong muốn hành vi chung giữa các lớp con nên được đặt ở một lớp chung.

Các lớp cha có thể gọi các hành vi trong các lớp con của chúng một cách thống nhất (step by step).

2.3.11 Visitor

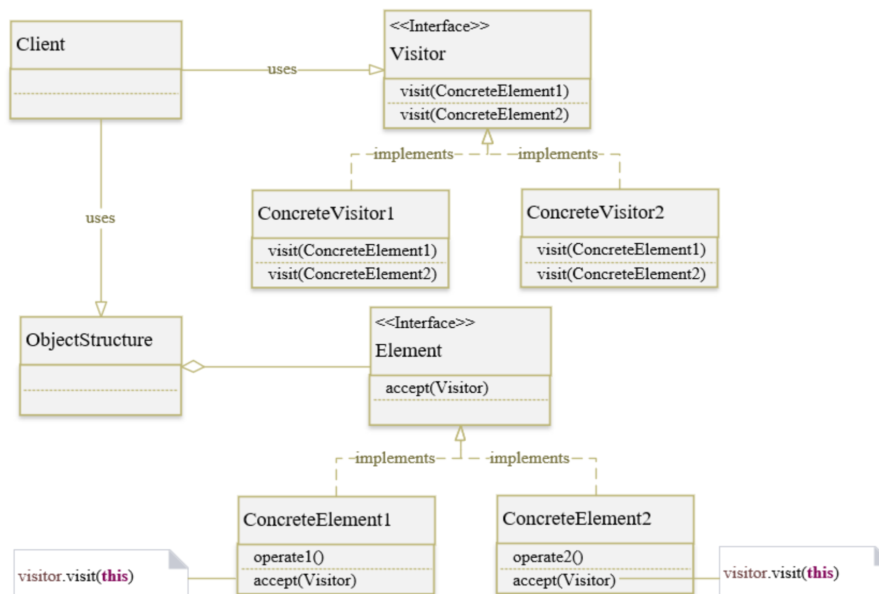
2.3.11.1 Giới thiệu về Visitor Pattern

Visitor cho phép định nghĩa các thao tác (operations) trên một tập hợp các đối tượng (objects) không đồng nhất (về kiểu) mà không làm thay đổi định nghĩa về lớp (classes) của các đối tượng đó. Để đạt được điều này, trong mẫu thiết kế visitor ta định nghĩa các thao tác trên các lớp tách biệt gọi các lớp visitors, các lớp này cho phép tách rời các thao tác với các đối tượng mà nó tác động đến. Với mỗi thao tác được thêm vào, một lớp visitor tương ứng được tạo ra.

Đây là một kỹ thuật giúp chúng ta phục hồi lại kiểu dữ liệu bị mất (thay vì dùng instanceof). Nó thực hiện đúng thao tác dựa trên tên của phương thức, kiểu của cả đối tượng gọi và kiểu của đối số truyền vào.

Visitor còn được biết đến như là **Double dispatch**.

2.3.11.2 Cài đặt Visitor Pattern như thế nào ?



Hình 2 – 23: Sơ đồ UML mô tả Visitor Pattern

Các thành phần tham gia Visitor Pattern:

- **Visitor** :
  - Là một interface hoặc một abstract class được sử dụng để khai báo các hành vi cho tất cả các loại visitor.
  - Class này định nghĩa một loạt các các phương thức truy cập chấp nhận các ConcreteElement cụ thể khác nhau làm tham số. Điều này sẽ hơi giống với cơ chế nạp chồng (overloading) nhưng các loại tham số nên khác nhau do đó các hành vi hoàn toàn khác nhau. Các hành vi truy cập sẽ được thực hiện trên từng phần tử cụ thể trong cấu trúc đối tượng thông qua phương thức visit(). Loại phần tử cụ thể đầu vào sẽ quyết định phương thức được gọi.
- **ConcreteVisitor** : cài đặt tất cả các phương thức abstract đã khai báo trong **Visitor**. Mỗi visitor sẽ chịu trách nhiệm cho các hành vi khác nhau của đối tượng.
- **Element** (Visitable): là một thành phần trừu tượng, nó khai báo phương thức accept() và chấp nhận đối số là Visitor.
- **ConcreteElement** (ConcreteVisitable): cài đặt phương thức đã được khai báo trong Element dựa vào đối số visitor được cung cấp.

- **ObjectStructure** : là một lớp chứa tất cả các đối tượng Element, cung cấp một cơ chế để duyệt qua tất cả các phần tử. Cấu trúc đối tượng này có thể là một tập hợp (collection) hoặc một cấu trúc phức tạp giống như một đối tượng tổng hợp (composite).
- **Client** : không biết về ConcreteElement và chỉ gọi phương thức accept() của Element.

2.3.11.3 Lợi ích của Visitor Pattern

Cho phép một hoặc nhiều hành vi được áp dụng cho một tập hợp các đối tượng tại thời điểm run-time, tách rời các hành vi khỏi cấu trúc đối tượng.

Đảm bảo nguyên tắc Open/ Close: đối tượng gốc không bị thay đổi, dễ dàng thêm hành vi mới cho đối tượng thông qua visitor.

2.3.11.4 Sử dụng Visitor Pattern

Khi có một cấu trúc đối tượng phức tạp với nhiều class và interface. Người dùng cần thực hiện một số hành vi cụ thể của riêng đối tượng, tùy thuộc vào concrete class của chúng.

Khi chúng ta phải thực hiện một thao tác trên một nhóm các loại đối tượng tương tự. Chúng ta có thể di chuyển logic hành vi từ các đối tượng sang một lớp khác.

Khi cấu trúc dữ liệu của đối tượng ít khi thay đổi nhưng hành vi của chúng được thay đổi thường xuyên.

Khi muốn tránh sử dụng toán tử instanceof.

2.4. Kết luận

Trong chương này, khóa luận trình bày chi tiết các loại mẫu. Các mẫu này được phân loại theo cách giải quyết các vấn đề trên thực tế. Với mỗi mẫu, khả năng được áp dụng cũng khác nhau tùy vào bài toán, tình huống cụ thể. Đối với thành viên nhóm phân tích thiết kế, cần xác định rõ các vấn đề cần giải quyết sau đó phân rã các vấn đề thành các bài toán nhỏ hơn. Từ các bài toán nhỏ đó hoặc là có các mẫu có sẵn thì sẽ chọn các mẫu đưa vào sử dụng hoặc có thể phải xây dựng các mẫu mới. Việc xây dựng các mẫu mới dựa trên nguyên tắc xây dựng các mẫu có trước. Sau đó tổng hợp các mẫu và xác định mối quan hệ giữa các mẫu để giải quyết vấn đề ban đầu. Việc sử dụng mẫu có nhiều lợi ích, tuy nhiên không nên lạm dụng kỹ thuật này cho các bài toán nhỏ.

CHƯƠNG 3: ÁP DỤNG DESIGN PATTERN TRONG PHÁT TRIỂN PHẦN MỀM

3.1. Giới thiệu

Design Pattern được ứng dụng nhiều trong phát triển phần mềm. Cách tiếp cận này đem lại nhiều lợi ích như phát triển nhanh, tái sử dụng, đảm bảo tính đúng đắn của phần mềm. Trong chương này, khóa luận trình bày cách áp dụng Design Pattern trong phát triển phần mềm. Do giới hạn của khóa luận, bài toán được lựa chọn là một phần nhỏ trong bài toán quản lý của một trường mầm non để minh họa cách áp dụng Design Pattern trong phát triển phần mềm. Đầu tiên khóa luận xác định các bài toán cần giải quyết trong phạm vi phần mềm, tiếp theo lần lượt giải quyết từng vấn đề bằng cách lựa chọn các mẫu phù hợp với từng bài toán, cuối cùng tổng hợp và ghép các mẫu để hoàn thành công việc. Bài toán được mô tả cụ thể trong phần tiếp theo.

3.2. Bài toán đăng ký tuyển sinh mầm non

Hiện nay, dịch vụ trông trẻ đang phát triển theo quy mô lớn. Số lượng nơi trông trẻ đang ngày càng tăng, dẫn đến việc không thể đáp ứng được nhu cầu của phụ huynh. Nếu vẫn quản lý theo kiểu thủ công sẽ rất mất nhiều thời gian lẫn công sức, hiệu quả thì không được cao. Thậm chí sẽ bị nơi trông trẻ khác chiếm mất phần thị trường. Vậy nên việc nâng cấp quy trình làm việc, sử dụng công nghệ trong quản lý cũng như tăng chất lượng dịch vụ là điều tất yếu.

3.3. Mô tả các nghiệp vụ

3.3.1 Bản mô tả công việc

Phụ huynh khi có nhu cầu gửi trẻ sẽ đăng ký nhập học cho trẻ. Nếu đủ điều kiện nhập học, phụ huynh sẽ nộp sơ yếu lý lịch của trẻ. Phụ huynh có thể đăng ký lớp năng khiếu cho trẻ khi đăng ký nhập học.

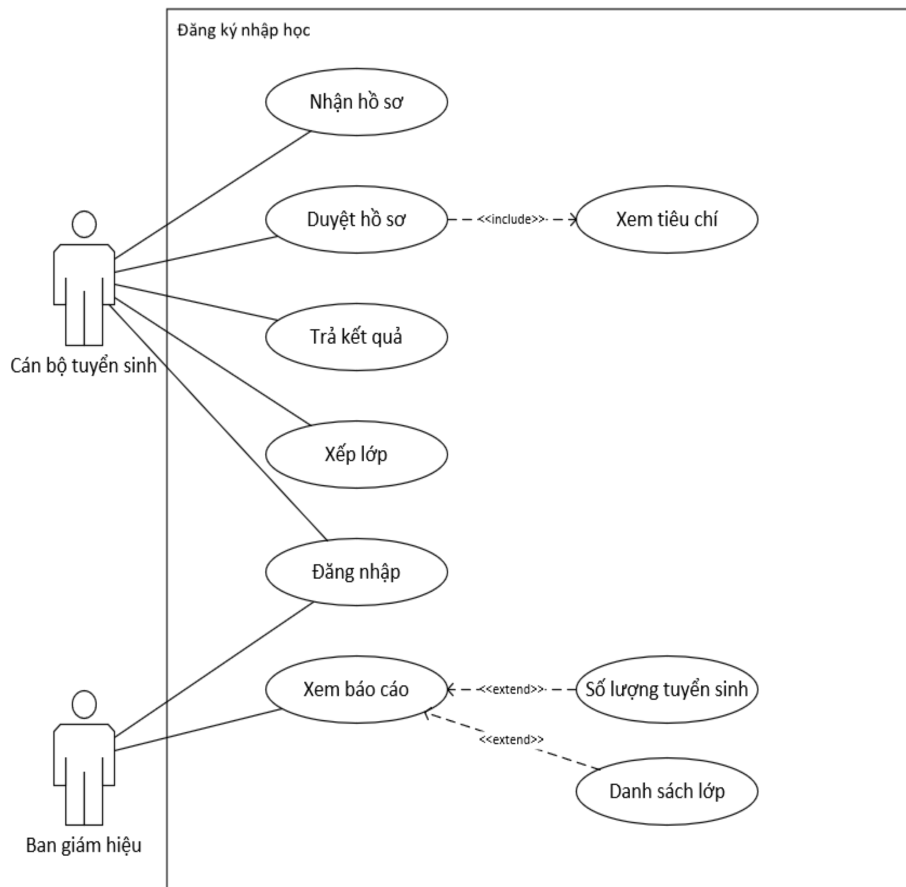
Cán bộ tuyển sinh sẽ nhận đơn đăng ký nhập học và xem xét duyệt đơn. Nếu không đủ điều kiện nhập học thì sẽ thông báo lại cho phụ huynh không đủ điều kiện nhập học. Nếu đủ điều kiện nhập học thì sẽ thông báo nhập học cho phụ huynh. Cán bộ tuyển sinh sẽ nhận sơ yếu lý lịch của trẻ, dựa vào sơ yếu lý lịch của trẻ sẽ xếp lớp sao cho hợp lý. Trẻ được chia lớp hành chính dựa theo độ tuổi và lớp năng khiếu dựa theo đăng ký của phụ huynh. Nếu lớp hành chính quá đông các trẻ sẽ được chia làm hai lớp dựa theo danh sách. Sau đó cán tuyển sinh sẽ thông báo cho giáo viên phụ trách lớp học.

3.3.2 Danh sách các công việc cần thực hiện

STT	Tác nhân	Use case	Mẫu đơn
1	Cán bộ tuyển sinh	Nhận hồ sơ	Đơn đăng ký nhập học
2		Duyệt hồ sơ	Đơn tiêu chí nhập học
3		Trả kết quả	Đơn thông báo đủ / không điều kiện nhập học
4		Xếp lớp	
5	Ban giám hiệu	Xem báo cáo	

3.4 Phân tích thiết kế hướng đối tượng

3.4.1 Biểu đồ trường hợp sử dụng (Use case diagram)



Hình 3 -1: Biểu đồ Use case diagram Quản lý tuyển sinh

Danh sách tác nhân :

- Cán bộ tuyển sinh : Người quản lý danh sách học sinh, duyệt đơn nhập học, xếp lớp, làm việc với phụ huynh.

- Người quản lý: Người chịu trách nhiệm quản lý cao nhất

- Phụ huynh: Người nộp hồ sơ tuyển sinh của trẻ

- Học sinh:

Các ca sử dụng :

- Nhận hồ sơ: Cán bộ tuyển sinh đưa mẫu đơn có sẵn cho phụ huynh.

- Duyệt hồ sơ: Dựa vào đơn đăng ký, cán bộ tuyển sinh sẽ xem xét các tiêu chí đã được quy định sẵn

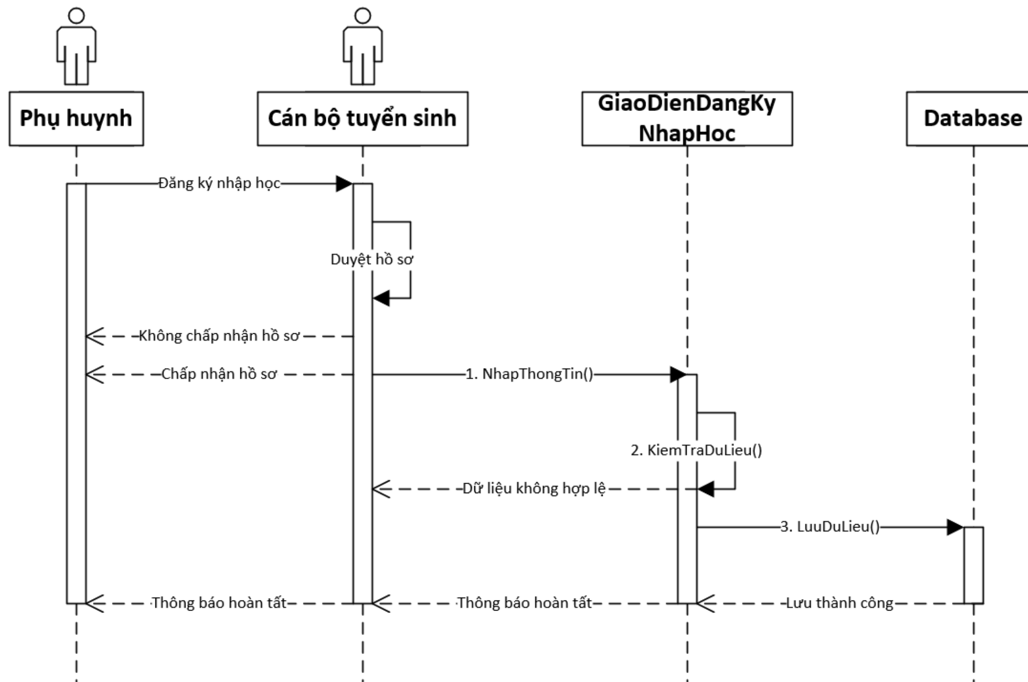
- Trả kết quả : Cán bộ tuyển sinh sẽ thông báo đủ điện kiện hoặc không đủ

điều kiện nhập học cho phụ huynh

- Xếp lớp : Cán bộ tuyển sinh sẽ xếp lớp dựa theo độ tuổi

- Xem báo cáo: Người quản lý sẽ xem báo cáo chi tiết về số lượng tuyển sinh, danh sách lớp, …

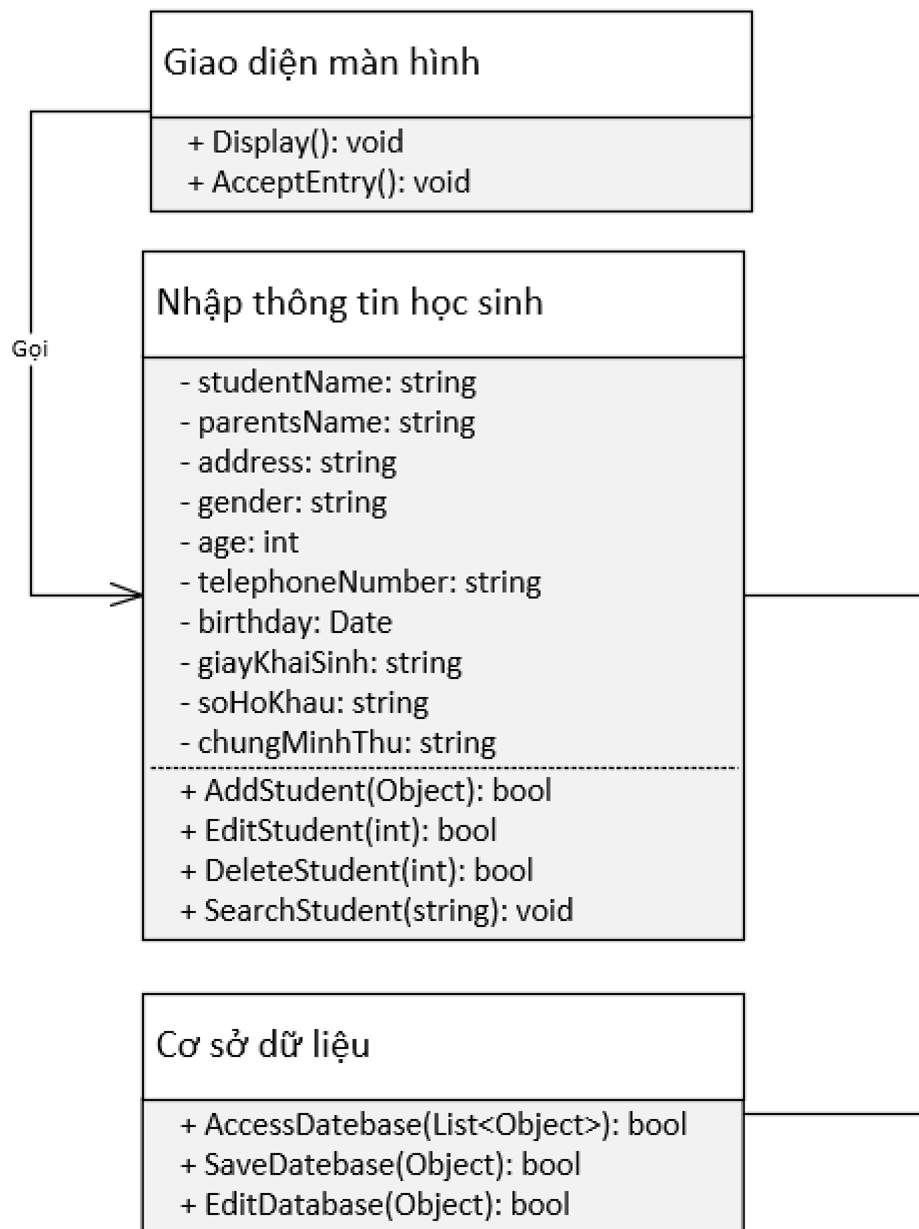
3.4.2 Biểu đồ trình tự quản lý tuyển sinh (Sequence diagram)



Hình 3 -2: Biểu đồ Sequence diagram quản lý tuyển sinh

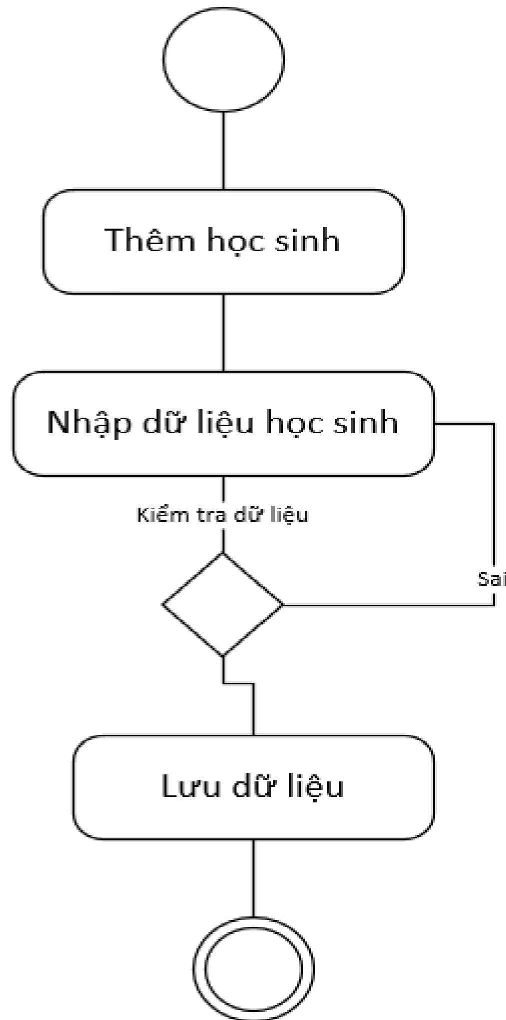
Mô tả chi tiết hoạt động đăng ký tuyển sinh: Phụ huynh đến xin nhập học cho trẻ, cán bộ tuyển sinh sẽ lấy mẫu đơn có sẵn đưa cho phụ huynh, phụ huynh sau khi điền đầy đủ thông tin thì sẽ trả lại mẫu đơn cho cán bộ tuyển sinh, cán bộ tuyển sinh duyệt hồ sơ của trẻ. Nếu không đủ điều kiện thì thông báo ngay cho phụ huynh. Còn nếu đủ điều kiện thì sẽ làm thủ tục nhập học cho trẻ. Cán bộ tuyển sinh sẽ xếp lớp dựa theo độ tuổi.

3.4.3 Biểu đồ lớp (Class diagram)



Hình 3 - 3: Biểu đồ Class diagram

3.4.4 Biểu đồ chuyển trạng thái (State transition diagram)



Hình 3 - 4: Biểu đồ State transition diagram

Mô tả hoạt động: Cán bộ tuyển sinh sẽ nhập thông tin học sinh vào trong hệ thống. Nếu thông tin không chính xác thì hệ thống sẽ yêu cầu nhập lại. Khi hệ thống báo dữ liệu hợp lệ thì sẽ lưu dữ liệu và database.

3.5 Áp dụng Design Pattern vào bài toán

Có 3 vấn đề thiết kế chính cần được giải quyết đó là:

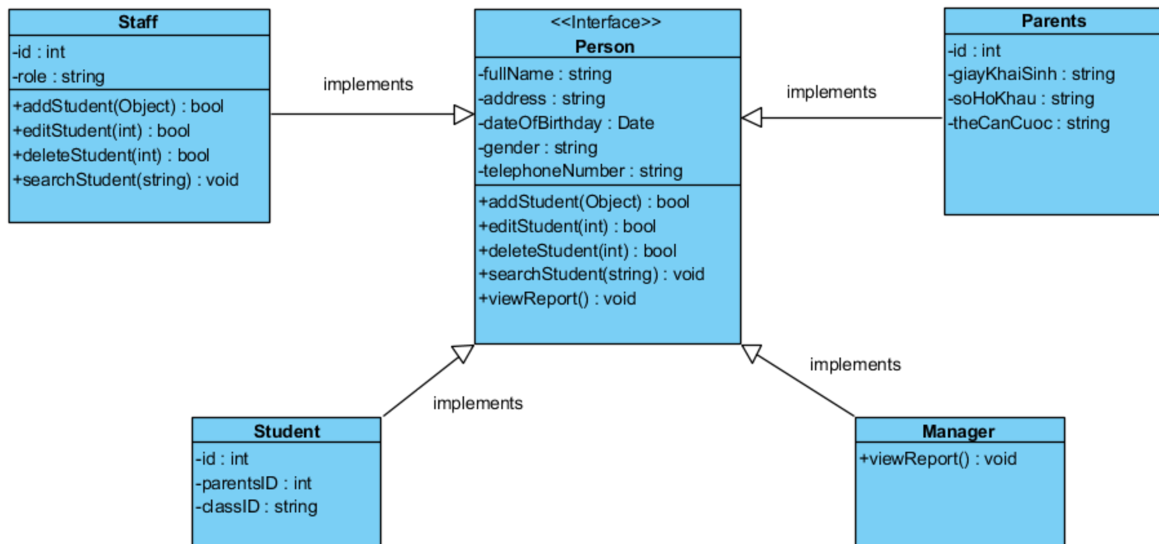
- *Trừu tượng hóa*: Các tác nhân trong hệ thống phải được thiết kế sao cho kế thừa từ một lớp trừu tượng hay còn gọi là super class.

- *Quy trình tuyển sinh*: Quá trình từ lúc tiếp nhận hồ sơ, duyệt hồ sơ và cho đến khi đọc kết quả cho phụ huynh phải được đảm bảo và tuân thủ theo đúng quy định của nơi trông trẻ.

- *Bảo mật hệ thống*: Khi các tác nhân thao tác trên hệ thống thì phải được đăng nhập theo username / password đã được cấp sẵn trước đó. Tùy từng vào vai trò mà các tác nhân có thể truy cập sâu vào các chức năng của hệ thống.

Phần trình bày dưới đây sẽ đi vào chi tiết từng vấn đề cụ thể, cách tiếp cận và hướng dẫn quyết ra sao.

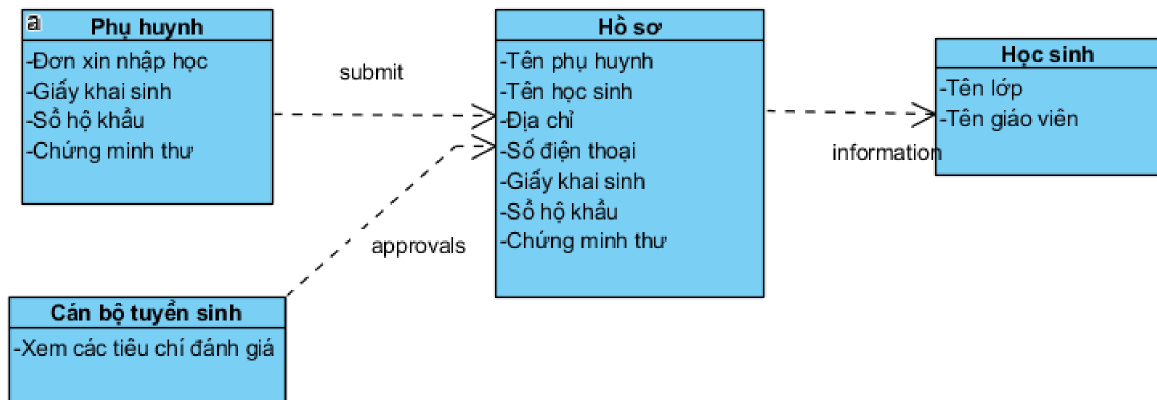
3.5.1 Trừu tượng hóa



Hình 3 - 5: Trừu tượng hóa các lớp của hệ thống

Sơ đồ trên khái quát hóa các tác nhân của hệ thống về một lớp trừu tượng đó là lớp Person. Lớp này biểu diễn chung nhất mà các tác nhân trong hệ thống có những thuộc tính giống nhau và các phương thức giống nhau. Giúp ta có cái nhìn tổng quát nhất về các tác nhân trong hệ thống.

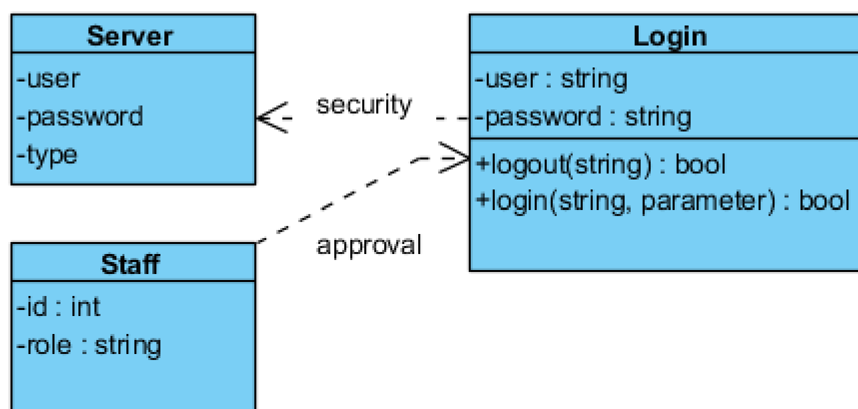
3.5.2 Quy trình tuyển sinh



Hình 3 - 6: Quy trình tuyển sinh nhập học của trẻ

Hồ sơ gồm những tiêu chí đã được quy định sẵn của nơi giữ trẻ. Phụ huynh nộp hồ sơ cho cán bộ tuyển sinh, cán bộ tuyển sinh sẽ phải kiểm trả xem hồ sơ đã đầy đủ thông tin chưa, cũng như đã nộp đủ giấy tờ cần thiết. Nếu đầy đủ hồ sơ thì cán bộ tuyển sinh sẽ đồng ý nhận trẻ và thông báo cho phụ huynh. Nếu thiếu một số giấy tờ có thể bổ sung thì vẫn cho trẻ nhận học.

3.5.3 Bảo mật hệ thống



Hình 3 - 7: Quy trình đăng nhập vào hệ thống

Để bảo mật hệ thống thì cần phải xây dựng chức năng Login cho hệ thống. Dựa vào username / password thì các tác nhân có thể sử dụng được các chức năng được phân rõ quyền hạn. Như vậy, hệ thống sẽ chạy ổn định và bảo toàn dữ liệu.

3.6 Chương trình thực nghiệm

HỆ THỐNG QUẢN LÝ GIÁO DỤC MẦM NON

Đăng nhập

Tên đăng nhập:

Mật khẩu:

Show Password

Đăng nhập

Quên mật khẩu

Hình 3 - 8: Login vào hệ thống

Thông tin học sinh

THÔNG TIN HỌC SINH

Thông tin học sinh

Mã lớp: 4T

Ngày sinh: 09/05/2015

Tên học sinh: Cù Thế Huy

Giới tính: Nam / Nữ

Thông tin phụ huynh

Tên phụ huynh: Đoàn Thị Ngọc

Số điện thoại: 0383412572

Địa chỉ: Thái Bình

Các loại giấy tờ

☑ Đơn xin nhập học

☑ Bản sao sổ hộ khẩu

☐ Bản sao giấy khai sinh

☑ Bảo sao thẻ căn cước công dân

Thao tác xử lý

Thêm học sinh

Xóa học sinh

Sửa học sinh

Tìm kiếm

Mã HS	Tên HS	Tên PH	Mã Lớp	Giới Tính	Ngày Sinh	Địa Chỉ	SĐT	Đơn Nhập Học	Giấy Khai Sinh
1	Cù Thế Huy	Đoàn Thị Ngọc	4T	Nữ	09/05/2015	Thái Bình	0383412572	Đã nộp	Chưa nộp
2	Nguyễn Văn Bắc	Đào Khả Doanh	4T	Nam	21/05/2015	Hải Phòng	0352412572	Đã nộp	Đã nộp
3	Trần Thị Bình	Trần Minh Phương Thảo	4T	Nam	09/10/2015	Hải Phòng	0383485572	Chưa nộp	Đã nộp
4	Trần Văn Ngọc	Đoàn Thị Ngọc	5T	Nữ	09/05/2014	Hải Phòng	0842684102	Đã nộp	Đã nộp
5	Nguyễn Văn Thắng	Đoàn Thị Ngọc	5T	Nam	21/05/2014	Hải Phòng	0311412572	Đã nộp	Đã nộp
6	Trần Thị Kiều	Đoàn Thị Ngọc	5T	Nữ	09/10/2014	Hải Phòng	0383005572	Chưa nộp	Đã nộp
7	Bùi Thị Nga	Nguyễn Tấn Dũng	3T	Nữ	03/12/2016	Bạc Liêu	0123456789	Đã nộp	Đã nộp

Hình 3 - 9: Giao diện chức năng nghiệp vụ quản lý tuyển sinh

Thêm dữ liệu học sinh vào hệ thống

```
// Câu lệnh insert
string query = "INSERT INTO tblHocSinh VALUES(N'" + studentName + "', '" + classID + "', N'" + gender + "', " + brithday + ", N'"
    + address + "', '" + telephoneNumber + "', N'" + parentsName + "', N'" + giayKhaiSinh + "', N'" +
    soHoKhau + "', N'" + chungMinhThu + "');";

try
{
    SingletonDB.ExcuteQuery(query);

    // Refresh DB
    dataTable = SingletonDB.GetDataToDataTable("SELECT * FROM tblHocSinh");
    dgrvDSHocSinh.DataSource = dataTable;

    // Cell Click
    DataDisplay();
    MessageBox.Show("Inserted successful !");
}
catch (Exception ex)
{
    MessageBox.Show("Error: " + ex.Message);
}
```

Hình 3 - 10: Chức năng thêm học sinh vào hệ thống

Sửa dữ liệu học sinh trong hệ thống

```
// Câu lệnh update
string query = "UPDATE tblHocSinh SET TenHocSinh = N'" + studentName + "', MaLop = '" + classID + "', GioiTinh = N'" +
    + address + "', SDT = '" + telephoneNumber + "', TenPhuHuynh = N'" + parentsName + "', BanSaoGiayKhaiSinh = N'" +
    giayKhaiSinh + "', BanSaoSoHoKhau = N'" + soHoKhau + "', BanSaoTCC = N'" + chungMinhThu + "' WHERE MaHocSinh = '" +

try
{
    SingletonDB.ExcuteQuery(query);

    // Refresh DB
    dataTable = SingletonDB.GetDataToDataTable("SELECT * FROM tblHocSinh");

    // Cell Click
    DataView dataView = dataTable.DefaultView;

    dataView.RowFilter = "TenHocSinh LIKE '%" + txtSearch.Text.Trim() + "%' OR DiaChi LIKE '%" + txtSearch.Text.Trim()
        + txtSearch.Text.Trim() + "%' OR MaLop LIKE '%" + txtSearch.Text.Trim() + "%'";
    dgrvDSHocSinh.DataSource = dataView;

    DataDisplay();
    MessageBox.Show("Updated successful !");
}
catch (Exception ex)
{
    MessageBox.Show("Error: " + ex.Message);
}
```

Hình 3 - 11: Chức năng sửa thông tin học sinh trong hệ thống

Xóa dữ liệu học sinh trong hệ thống

```
// Câu lệnh delete
string maHocSinh = dgrvDSHocSinh.CurrentRow.Cells["MaHocSinh"].Value.ToString();
string query = "DELETE tblHocSinh WHERE MaHocSinh = '" + maHocSinh + "'";

try
{
    SingletonDB.ExcuteQuery(query);

    // Refresh database
    dgrvDSHocSinh.DataSource = SingletonDB.GetDataToDataTable("SELECT * FROM tblHocSinh");

    // Cell Click
    DataDisplay();
    MessageBox.Show("Delete successful !");
}
catch (Exception ex)
{
    MessageBox.Show("Error: " + ex.Message);
}
```

Hình 3 - 12: Chức năng xóa học sinh khỏi hệ thống

Login vào hệ thống

```
4 references | 0 changes | 0 authors, 0 changes
public sealed class Logger
{
    private static volatile Logger INSTANCE;

    private static readonly object syncLock = new object();

    1 reference | 0 changes | 0 authors, 0 changes
    private Logger()
    {
    }

    0 references | 0 changes | 0 authors, 0 changes
    public static Logger GetLogger
    {
        get
        {
            lock (syncLock)
            {
                if (INSTANCE == null)
                {
                    INSTANCE = new Logger();
                }
            }

            return INSTANCE;
        }
    }
}
```

Hình 3 - 13: Áp dụng Singleton Pattern vào quy trình đăng nhập hệ thống

Kết nối dữ liệu lên Server

```
private static volatile SqlConnection INSTANCE; // Tránh đụng độ Thread
private static readonly object syncRoot = new object();
private const string connectionString = @"Data Source=THODOAN-PC;Initial C

0 references | 0 changes | 0 authors, 0 changes
private SingletonDB() { }

0 references | 0 changes | 0 authors, 0 changes
public static SqlConnection getInstance
{
    get
    {
        lock(syncRoot)
        {
            if(INSTANCE == null)
            {
                INSTANCE = new SqlConnection(connectionString);
            }
        }

        return INSTANCE;
    }
}
```

Hình 3 - 14: Áp dụng Singleton Pattern vào quy trình kết nối dữ liệu

KẾT LUẬN

Design Pattern là một vấn đề hết sức quan trọng đối với các tổ chức phát triển phần mềm hiện nay. Trong quá trình thực hiện đồ án do thời gian nghiên cứu và kinh nghiệm bản thân còn hạn chế nên một số phần của đồ án nghiên cứu chưa được sâu.

Sau 03 tháng thực hiện nghiên cứu đề tài, dưới sự hướng dẫn tận tình của Tiến sỹ Nguyễn Trịnh Đông, đồ án của em đã đạt được những kết quả sau:

Kết quả đạt được

- Tìm hiểu và nghiên cứu cơ sở lý thuyết của Design Pattern
- Nắm được một số kỹ thuật hay sử dụng và cách sử dụng
- Biết áp dụng Design Pattern vào bài toán đơn giản

Hạn chế

Trong thời gian qua, em đã cố gắng hết sức để tìm hiểu thực hiện đề tài. Tuy nhiên với kinh nghiệm và thời gian hạn chế nên không thể tránh khỏi những thiếu sót trong đồ án. Cụ thể:

- Chưa nghiên cứu sâu vào các kỹ thuật của Design Patterns
- Đồ án mới tập trung vào một bài toán nhỏ nên vẫn chưa toát nên được tầm quan trọng của Design Patterns
- Trình bày thiếu logic, cách diễn đạt còn kém

Hướng phát triển của đề tài trong tương lai

Với mong muốn trở thành lập trình viên phần mềm trong tương lai nên trong thời gian tới em sẽ tiếp tục tìm hiểu và nghiên cứu cũng như hoàn thiện đồ án của mình ở mức cao nhất. Và sau đó sẽ nghiên cứu và áp dụng Design Patterns vào bài toán Quản lý mầm non để thấy được tầm quan trọng của nó. Rồi từ đó rút ra kinh nghiệm của bản thân và cải thiện các kỹ năng cần thiết.

DANH MỤC TÀI LIỆU THAM KHẢO

[1]: https://en.wikipedia.org/wiki/Software_design_pattern

[2]: https://toidicodedao.com/2016/03/01/nhap-mon-design-pattern-phong-cach-kiem-hiep

[3]: https://gpcoder.com/4164-gioi-thieu-design-patterns

[4]: Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm