TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

**KHOA CÔNG NGHỆ THÔNG TIN**



**TIỂU LUẬN GIỮA KỲ MÔN MẪU THIẾT KẾ**

**TÌM HIỂU VỀ MEDIAOR VÀ MEMENTO PATTERN**

*Người hướng dẫn*: ***Thầy Vũ Đình Hồng***

*Người thực hiện*: **NGUYỄN VĂN THỊNH – 52100842**

**ĐỖ MINH ĐĂNG – 52100749**

**NGUYỄN KHÁNH HƯNG - 52100801**

Khoá  **: 25**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2024**

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM

**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

**KHOA CÔNG NGHỆ THÔNG TIN**



**TIỂU LUẬN GIỮA KỲ MÔN MẪU THIẾT KẾ**

**TÌM HIỂU VỀ MEDIAOR VÀ MEMENTO PATTERN**

*Người hướng dẫn****: Thầy Vũ Đình Hồng***

*Người thực hiện****:* NGUYỄN VĂN THỊNH – 52100842**

**ĐỖ MINH ĐĂNG – 52100749**

**NGUYỄN KHÁNH HƯNG - 52100801**

Khoá**:** **25**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2024**

LỜI CẢM ƠN

Lời đầu tiên, nhóm chúng em xin gửi đến Thầy Vũ Đình Hồng những lời tri ân chân thành nhất từ tận đáy lòng về sự hướng dẫn và sự hỗ trợ của Thầy trong suốt thời gian học môn Design Pattern. Thầy không chỉ là một người thầy dạy bài mà còn là một người hướng dẫn tận tâm và đầy tâm huyết. Những giờ học với Thầy không chỉ giúp chúng em nắm vững kiến thức về Design Pattern mà còn mở ra một cánh cửa mới về cách tiếp cận vấn đề và tư duy logic trong lập trình.

Bài báo cáo của nhóm chúng em là thành quả của kiến thức được đúc kết thời gian vừa qua cũng mang một số thực tiễn cao và phù trợ nho nhỏ cho công việc tương lai sau này. Vì thời gian chuẩn bị không đủ nhiều, có thể cách diễn đạt của nhóm chưa được hoàn thiện nhất nên có thể bài báo cáo của nhóm em còn gặp một số sai sót và hạn chế nhất định.

Mong được sự chỉ dẫn và đóng góp ý kiến từ thầy. Từ đó chúng em có thể sửa đổi và rút ra kinh nghiệm cho chính bản thân và ngày một hoàn thiện hơn.

Lời nói cuối cùng, nhóm em xin chân thành cảm ơn thầy!

**ĐỒ ÁN ĐƯỢC HOÀN THÀNH**

**TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**

Tôi xin cam đoan đây là sản phẩm đồ án của riêng chúng tôi và được sự hướng dẫn của Thầy Vũ Đình Hồng. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong đồ án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

**Nếu phát hiện có bất kỳ sự gian lận nào tôi xin hoàn toàn chịu trách nhiệm về nội dung đồ án của mình.** Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

*TP. Hồ Chí Minh, ngày 23 tháng 11 năm 2023*

*Tác giả*

*(ký tên và ghi rõ họ tên)*

*Nguyễn Văn Thịnh*

*Đỗ Minh Đăng*

*Nguyễn Khánh Hưng*

PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

**Phần xác nhận của GV hướng dẫn**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tp. Hồ Chí Minh, ngày tháng năm

(kí và ghi họ tên)

**Phần đánh giá của GV chấm bài**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Tp. Hồ Chí Minh, ngày tháng năm

(kí và ghi họ tên)

MỤC LỤC

[PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN 5](#_Toc164347348)

[DANH MỤC CÁC BẢNG BIỂU, HÌNH VẼ, ĐỒ THỊ 8](#_Toc164347349)

[CHƯƠNG 1 – MEMENTO PATTERN 9](#_Toc164347350)

[1.1 Giới thiệu Memento Pattern 9](#_Toc164347351)

[1.1.1 Khái niệm Memento Pattern 9](#_Toc164347352)

[1.1.2 Mục đích ra đời Memento Pattern 10](#_Toc164347353)

[1.2 Kiến trúc Memento Pattern 13](#_Toc164347354)

[1.2.1 Sơ lược về kiến trúc Memento Pattern 13](#_Toc164347355)

[1.2.2 Chi tiết kiến trúc Memento Pattern 14](#_Toc164347356)

[1.2.3 Cách triển khai Memento Pattern 15](#_Toc164347357)

[1.3 Mối quan hệ của Memento Pattern với các Pattern khác 16](#_Toc164347358)

[1.3.1 Memento Pattern với Command Pattern 16](#_Toc164347359)

[1.3.2 Memento Pattern với State Pattern 16](#_Toc164347360)

[1.3.3 Memento Pattern với Iterator Pattern 17](#_Toc164347361)

[1.3.4 Memento Pattern với Strategy Pattern 17](#_Toc164347362)

[1.3.5 Memento Pattern với Prototype Pattern 17](#_Toc164347363)

[1.3.6 Memento Pattern với Observer Pattern 17](#_Toc164347364)

[1.4 So sánh và nghiên cứu tính năng Memento Pattern 18](#_Toc164347365)

[1.4.1 So sánh với các phương pháp khác về tính năng undo và redo 18](#_Toc164347366)

[1.4.2 Memento Pattern trong các ngôn ngữ và framework khác nhau 19](#_Toc164347367)

[1.4.3 Nghiên cứu về tối ưu hóa hiệu suất của Memento Pattern 20](#_Toc164347368)

[1.5 Ưu & nhược điểm của Memento Pattern 22](#_Toc164347369)

[1.5.1 Ưu điểm 22](#_Toc164347370)

[1.5.2 Nhược điểm 22](#_Toc164347371)

[1.6 Cách sử dụng và Những trường hợp nên sử dụng Mementor Pattern 23](#_Toc164347372)

[1.6.1 Cách sử dụng Mementor Pattern 23](#_Toc164347373)

[1.6.2 Những trường hợp nên sử dụng Mementor Pattern 24](#_Toc164347374)

[1.7 Ứng dụng của Memento Pattern 25](#_Toc164347375)

[1.7.1 Ứng dụng Desktop và ứng dụng web 25](#_Toc164347376)

[1.7.2 Chỉnh sửa văn bản và đồ họa 25](#_Toc164347377)

[1.7.3 Hệ thống giao dịch tài chính 25](#_Toc164347378)

[1.7.4 Ứng dụng trò chơi và giải trí 26](#_Toc164347379)

[1.7.5 Ứng dụng trong Machine Learning và AI 26](#_Toc164347380)

[1.7.6 Hệ thống tích hợp và xử lý sự kiện 26](#_Toc164347381)

[1.8 Ví dụ cụ thể của Memento Pattern 26](#_Toc164347382)

[1.8.1 Sự tương đồng trong thế giới thực của mẫu thiết kế Memento 26](#_Toc164347383)

[1.8.2 Demo nhỏ về Memento Pattern quản lý trạng thái của đối tượng 27](#_Toc164347384)

[1.9 Source code Memento Pattern 32](#_Toc164347385)

[1.9.1 Giới thiệu về đề tài 32](#_Toc164347386)

[1.9.2 Mô hình đề tài 33](#_Toc164347387)

[1.9.3 Kết quả đề tài 40](#_Toc164347388)

[CHƯƠNG 2 – MEDIATOR PATTERN 43](#_Toc164347389)

[2.1 Giới thiệu Mediator Pattern 43](#_Toc164347390)

[2.1.1 Khái niệm Mediator Pattern 43](#_Toc164347391)

[2.1.2 Mục đích ra đời 43](#_Toc164347392)

[2.1.3 Cách triển khai Mediator Pattern 44](#_Toc164347393)

[2.2 Kiến trúc Mediator Pattern 45](#_Toc164347394)

[2.2.1 Sơ lược về kiến trúc Mediator Pattern 45](#_Toc164347395)

[2.2.2 Chi tiết về kiến trúc Mediator Pattern 46](#_Toc164347396)

[2.3 Mối quan hệ của Mediator Pattern với các Pattern khác 47](#_Toc164347397)

[2.3.1 Mediator Pattern với Observer Pattern 47](#_Toc164347398)

[2.3.2 Mediator Pattern với Singleton Pattern 48](#_Toc164347399)

[2.3.3 Mediator Pattern với Facade Pattern 48](#_Toc164347400)

[2.3.4 Mediator Pattern với Command Pattern 48](#_Toc164347401)

[2.3.5 Mediator Pattern với Factory Pattern 48](#_Toc164347402)

[2.4 Mediator Pattern trong các ngôn ngữ và framework khác nhau 48](#_Toc164347403)

[2.4.1 Java 48](#_Toc164347404)

[2.4.2 C# (.NET) 49](#_Toc164347405)

[2.4.3 JavaScript 49](#_Toc164347406)

[2.4.4 Python 49](#_Toc164347407)

[2.4.5 Angular 50](#_Toc164347408)

[2.4.6 ASP.NET Core 50](#_Toc164347409)

[2.4.7 Spring Framework 50](#_Toc164347410)

[2.5 Ưu & nhược điểm của Mediator Pattern 51](#_Toc164347411)

[2.5.1 Ưu điểm 51](#_Toc164347412)

[2.5.2 Nhược điểm 51](#_Toc164347413)

[2.6 Trường hợp sử dụng Mediator Pattern 51](#_Toc164347414)

[2.6.1 Khi nào nên sử dụng Mediator Pattern 51](#_Toc164347415)

[2.6.2 Khi nào không nên sử dụng Mediator Pattern 52](#_Toc164347416)

[2.7 Ứng dụng của Mediator Pattern 52](#_Toc164347417)

[2.7.1 Trong Phát triển Ứng dụng Web 52](#_Toc164347418)

[2.7.2 Trong hệ thống điều khiển và tự động hóa 53](#_Toc164347419)

[2.7.3 Trong ứng dụng mạng xã hội 53](#_Toc164347420)

[2.7.4 Trong ứng dụng trò chơi 54](#_Toc164347421)

[2.8 Source code Mediator Pattern 54](#_Toc164347422)

[2.8.1 Giới thiệu về đề tài 54](#_Toc164347423)

[2.8.2 Mô hình đề tài 56](#_Toc164347424)

[2.8.3 Kết quả đề tài 62](#_Toc164347425)

[TÀI LIỆU THAM KHẢO 63](#_Toc164347426)

DANH MỤC CÁC BẢNG BIỂU, HÌNH VẼ, ĐỒ THỊ

[1. Chức năng Undo/Redo 10](#_Toc164347445)

[2. Vấn đề chức năng Undo/Redo 11](#_Toc164347446)

[3. Giải pháp cho undo/redo 12](#_Toc164347447)

[4. Kiến trúc Memento Pattern 14](#_Toc164347448)

[5. Code của demo nhỏ Memento với chức năng quản lý trạng thái 28](#_Toc164347449)

[6. Originator trong mô hình quản lý trạng thái 29](#_Toc164347450)

[7. Memento trong mô hình quản lý trạng thái 30](#_Toc164347451)

[8. Caretaker trong mô hình quản lý trạng thái 31](#_Toc164347452)

[9. Kết quả thực thi của mô hình 32](#_Toc164347453)

[10. Originator của đề tài quản lý văn bản 34](#_Toc164347454)

[11. Memento của đề tài quản lý văn bản 35](#_Toc164347455)

[12. Caretaker của đề tài quản lý văn bản 37](#_Toc164347456)

[13. Hàm Main của đề tài quản lý văn bản 39](#_Toc164347457)

[14. Kết quả của đề tài undo/redo 42](#_Toc164347458)

[15. Vấn đề Mediator Pattern 43](#_Toc164347459)

[16. Giải pháp cho Mementor Pattern 44](#_Toc164347460)

[17. Kiến trúc Mediator Pattern 46](#_Toc164347461)

[18. Sơ đồ của đề tài mô hình Chat 54](#_Toc164347462)

[19. Mediator của đề tài ứng dụng Chat 56](#_Toc164347463)

[20. ConcreteMediator của đề tài ứng dụng Chat 57](#_Toc164347464)

[21. Colleague của đề tài ứng dụng Chat 58](#_Toc164347465)

[22. ConcreteColleague của đề tài ứng dụng Chat 59](#_Toc164347466)

[23. Lớp Client cho việc thử nghiệm sử dụng mô hình Mediator 61](#_Toc164347467)

CHƯƠNG 1 – MEMENTO PATTERN

**1.1 Giới thiệu Memento Pattern**

***1.1.1 Khái niệm Memento Pattern***

- Phân loại: Nhóm hành vi (Behavior Pattern)

- Memento Pattern là một trong các mẫu thiết kế (design pattern) phổ biến trong lập trình hướng đối tượng (OOP). Mục tiêu chính của Memento Pattern là cho phép lưu trữ và khôi phục trạng thái của một đối tượng mà không tiết lộ chi tiết bên trong của đối tượng đó và mà không vi phạm nguyên tắc của tích hợp hướng đối tượng (encapsulation).

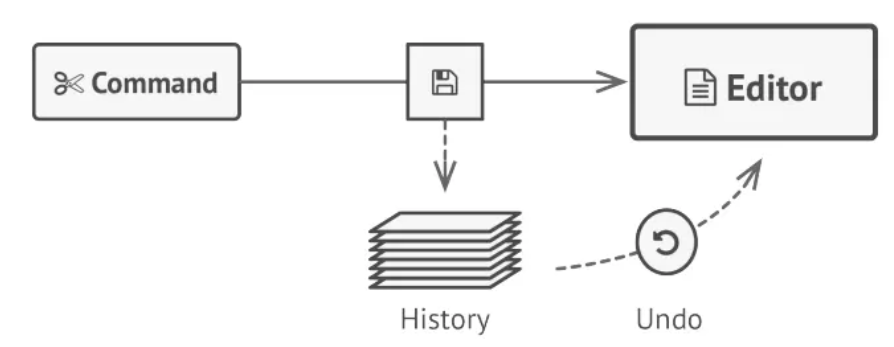
- Mục đích chính của nó là lưu trữ và khôi phục trạng thái của một đối tượng mà không làm thay đổi cấu trúc của nó và không làm tiết lộ chi tiết bên trong của đối tượng đó. Memento Pattern giúp tách biệt trạng thái của đối tượng và chi phí của việc quản lý trạng thái đó.

***1.1.2 Mục đích ra đời Memento Pattern***

***1.1.2.1 Undo/Redo***

***1.1.2.1.1 Giới thiệu về chức năng Undo/Redo***

Memento Pattern cung cấp một giải pháp linh hoạt và hiệu quả cho tính năng hoàn tác và làm lại trong các ứng dụng. Khi người dùng thực hiện các thao tác trên giao diện, như sửa đổi văn bản, vẽ hình ảnh hoặc thao tác trên dữ liệu, Memento Pattern cho phép lưu trữ các trạng thái trước đó của đối tượng. Khi người dùng yêu cầu hoàn tác hoặc làm lại, trạng thái trước đó có thể được khôi phục và áp dụng lại. Điều này cung cấp trải nghiệm người dùng mạnh mẽ và linh hoạt, giúp họ điều chỉnh và hoàn thiện các hành động một cách dễ dàng và linh hoạt hơn.

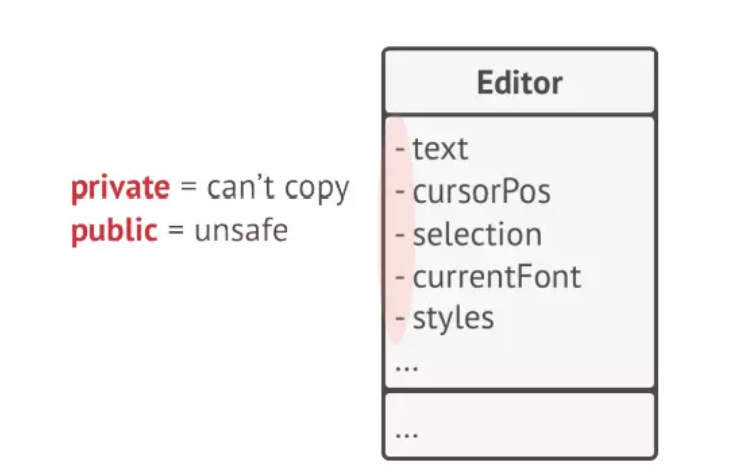
****

**1. Chức năng Undo/Redo**

***1.1.2.1.2 Vấn đề chức năng Undo/Redo***

Tưởng tượng ta đang tạo 1 text editor. Bao gồm các chức năng như chỉnh sửa text, format text, thêm ảnh, v.v.. Để phát triển thêm app, cần cho phép người dùng undo và redo bất kỳ thao tác nào thực hiện trên tệp văn bản. Bằng cách trước khi thực hiện bất kì thao tác nào, app sẽ lưu state tất cả object vào trong một storage (take snapshot, lưu vào history). Sau đó, khi user cần undo 1 thao tác, app lấy state đã được lưu trước đó trong storage và dùng nó để restore state của tất cả object.

Nhưng cái khó ở đây là, làm thế nào để thật sự take snapshot? Ta cần duyệt qua tất cả field của object để lưu nó vào storage. Tuy nhiên, việc đó là không khả thi vì thực tế hầu hết các objects thường giấu phần lớn data trong các trường private.Có vẻ như chỉ để take snapshot, app được đưa vào một tình thế rất gian nan: ta public tất cả các private fields của editor object khiến nó trở nên mong manh và tạo ra 1 class chuyên để copy editor object luôn phải thay đổi mỗi khi editor object thay đổi.

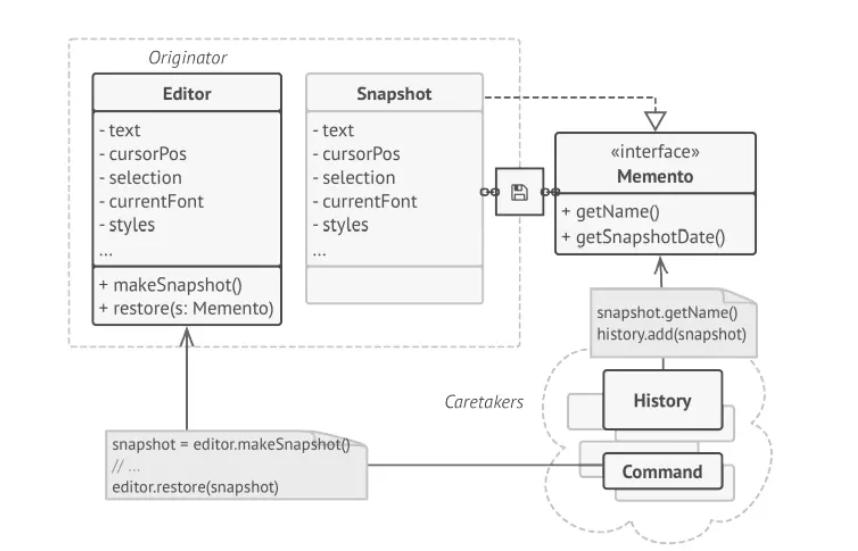
****

**2. Vấn đề chức năng Undo/Redo**

***1.1.2.1.3 Giải pháp***

Memento pattern giao việc tạo ra snapshot cho chính chủ nhân của state đó (originator object). Chính object đó sẽ dễ dàng tạo ra snapshot vì nó có toàn quyền truy cập state của nó.

Memento gợi ý nên lưu state được copy từ object vào một object gọi là memento. Content của memento object không được truy cập từ các object khác ngoại trừ originator object. Các object khác phải giao tiếp với memento thông qua interface bị giới hạn chỉ cho phép lấy metadata của snapshot (metadata - những data về data chứ không phải là data: ngày tạo, tên action, v.v.)..

****

**3. Giải pháp cho undo/redo**

*1.1.2.2 Ứng dụng Checkpointing*

Trong các ứng dụng và hệ thống phức tạp, việc duy trì tính nhất quán của dữ liệu là rất quan trọng. Memento Pattern giúp trong việc tạo các điểm kiểm soát (checkpoint) để lưu trữ các trạng thái trung gian của các đối tượng trong quá trình thực thi. Khi cần thiết, các checkpoint này có thể được sử dụng để khôi phục lại dữ liệu trong trạng thái nhất quán. Điều này giúp đảm bảo tính toàn vẹn và nhất quán của dữ liệu trong các tình huống phức tạp như trong các hệ thống cơ sở dữ liệu, hệ thống giao dịch và hệ thống phân phối.

*1.1.2.3 Tuần tự hóa*

Memento Pattern cung cấp một cơ chế hiệu quả để lưu trữ và truyền dữ liệu qua mạng bằng cách sử dụng các đối tượng Memento. Trạng thái của đối tượng có thể được lưu trữ dưới dạng Memento, và sau đó có thể được chuyển đổi thành dạng tuần tự (serialization) để lưu trữ vào bộ nhớ hoặc truyền qua mạng. Điều này làm cho việc truyền dữ liệu trở nên dễ dàng và hiệu quả hơn, đồng thời giảm thiểu khả năng mất mát dữ liệu trong quá trình truyền tải.

*1.1.2.4 Tính đóng gói*

Memento Pattern giúp duy trì tính đóng gói và bảo vệ trạng thái của đối tượng bằng cách ngăn chặn truy cập trực tiếp vào trạng thái của nó từ bên ngoài. Bằng cách giữ trạng thái của đối tượng trong các đối tượng Memento và cung cấp các phương thức chỉ cho phép truy cập từ Originator, Memento Pattern đảm bảo rằng tính bao quát và tính cá nhân của dữ liệu được bảo vệ.

*1.1.2.5 Dễ Dàng Mở Rộng*

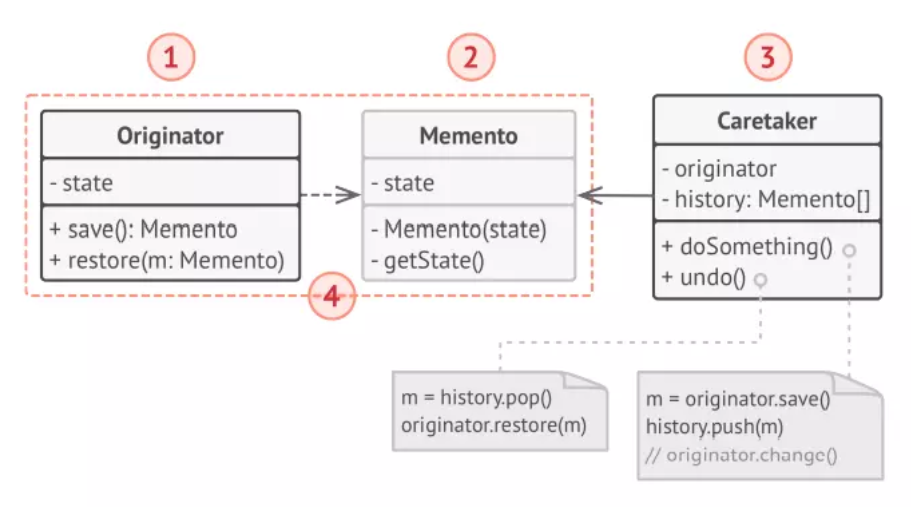
Memento Pattern cung cấp một kiến trúc linh hoạt và dễ mở rộng cho mã nguồn. Bằng cách tách biệt việc lưu trữ và khôi phục trạng thái của đối tượng vào các lớp riêng biệt (Originator và Memento), và bằng cách sử dụng Caretaker để quản lý các Memento, Memento Pattern giúp mã của trở nên dễ bảo trì và mở rộng. Ta có thể thêm các phương thức mới vào Originator hoặc Memento mà không làm thay đổi các phần khác của mã, đồng thời giữ cho tính linh hoạt và dễ mở rộng của mã.

1.2 Kiến trúc Memento Pattern

*1.2.1 Sơ lược về kiến trúc Memento Pattern*

Mục đích chính của nó là lưu trữ và khôi phục trạng thái của một đối tượng mà không làm thay đổi cấu trúc của nó. Mô hình này được sử dụng khi muốn lưu trữ và quản lý các phiên bản trạng thái của một đối tượng để có thể hoàn tác (undo) hoặc làm lại (redo) các thay đổi, hoặc đơn giản là lưu trữ trạng thái của một đối tượng trong các trường hợp nhất định.

Với cách triển khai này, Memento được lồng bên trong Originator. Giúp Originator truy cập private fields và methods của memento. Còn Caretaker bị giới hạn việc truy cập memento, cho phép nó lưu các mementos thành 1 stack nhưng không đụng gì đến các stat



**4. Kiến trúc Memento Pattern**

*1.2.2 Chi tiết kiến trúc Memento Pattern*

*1.2.2.1 Originator (Người khởi tạo)*

- Là class sản xuất ra snapshots từ các state của chính nó, đồng thời restore state từ snapshots khi cần, là đối tượng muốn lưu trữ và khôi phục trạng thái của nó. Originator cung cấp phương thức để tạo ra Memento chứa trạng thái hiện tại của nó và phương thức để khôi phục trạng thái từ một Memento cụ thể.

- Cung cấp phương thức để tạo ra Memento chứa trạng thái hiện tại của nó và phương thức để khôi phục trạng thái từ một Memento cụ thể.

- Thường sẽ có một hoặc nhiều phương thức để thực hiện các thao tác trên trạng thái của nó.

*1.2.2.2 Mementor*

- Là object lưu giá trị, được xem như là một snapshot của Originator, trong thực tiễn nó là immutable class (class không thay đổi được) và truyền data vào 1 lần duy nhất khi construct. Memento cung cấp các phương thức để truy cập và thay đổi trạng thái của nó.

- 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.

- Đối tượng chứa trạng thái của Originator tại một thời điểm cụ thể, nó cung cấp phương thức để lấy (get) và thiết lập (set) trạng thái, nhưng chỉ cho phép Originator thực hiện các thay đổi này. Memento đảm bảo rằng trạng thái của Originator được bảo vệ và không thể truy cập từ bên ngoài.

*1.2.2.3 Caretaker (Người giữ trạng thái)*

Giữ câu trả lời cho các câu hỏi "khi nào" và "vì sao" cho những thời điểm capture lại state của Originator và lúc khôi phục lại state. Caretaker lưu trữ 1 stack các mementos.

Khi Originator cần đi lùi (Undo) về history, Caretaker lấy memento trên cùng của stack và truyền vào restore method của Originator.

Đối tượng này chịu trách nhiệm lưu trữ các Memento. Caretaker không nên biết về cấu trúc nội bộ của Memento; nó chỉ cần lưu trữ và trả về chúng khi được yêu cầu. Caretaker cũng có thể duyệt qua danh sách các Memento đã lưu trữ, hoặc chỉ lưu trữ Memento cuối cùng. Có thể lưu trữ một hoặc nhiều Memento để thực hiện các thao tác hoàn tác và làm lại.

*1.2.3 Cách triển khai Memento Pattern*

- Xác định lớp nào sẽ đóng vai trò là người khởi tạo. Điều quan trọng là phải biết liệu chương trình sử dụng một đối tượng trung tâm thuộc loại này hay nhiều đối tượng nhỏ hơn.

- Tạo lớp Memento, lần lượt khai báo một tập hợp các trường phản ánh các trường được khai báo bên trong lớp khởi tạo.

- Làm cho lớp Memento nên bất biến, chỉ nên chấp nhận dữ liệu một lần thông qua hàm tạo, lớp (class) không nên có setters.

- Lồng Memento vào bên trong trình khởi tạo hoặc hãy trích xuất một giao diện trống từ lớp vật lưu niệm và làm cho tất cả các đối tượng khác sử dụng nó để tham chiếu đến Memento. Ta có thể thêm một số thao tác siêu dữ liệu vào giao diện nhưng không có thao tác nào làm lộ trạng thái của người khởi tạo (originator’s state).

- Thêm một phương thức để tạo Memento cho lớp originator, phải chuyển trạng thái của nó tới vật lưu niệm thông qua một hoặc nhiều đối số của hàm tạo của vật lưu niệm.

- Thêm một phương thức để khôi phục trạng thái của người khởi tạo (originator’s) vào lớp của nó.

- Đối với Caretaker, cho dù nó đại diện cho một đối tượng lệnh, một lịch sử hay một thứ gì đó hoàn toàn khác, nên biết khi nào cần yêu cầu Memento mới từ người khởi tạo, cách lưu trữ chúng và khi nào cần khôi phục người khởi tạo bằng một Memento cụ thể.

- Mối liên kết giữa Caretaker và Originator có thể được chuyển sang lớp Mementor. Trong trường hợp này, mỗi Memento phải được kết nối với người khởi tạo.

1.3 Mối quan hệ của Memento Pattern với các Pattern khác

*1.3.1 Memento Pattern với Command Pattern*

Memento Pattern thường được sử dụng cùng với Command Pattern để hỗ trợ tính năng hoàn tác và làm lại. Trong Command Pattern, các lệnh được biểu diễn dưới dạng các đối tượng có khả năng thực thi một hành động cụ thể. Khi một lệnh được thực thi, Memento Pattern có thể được sử dụng để lưu trữ trạng thái của đối tượng trước khi lệnh được thực hiện, cho phép hoàn tác hành động đó nếu cần thiết.

*1.3.2 Memento Pattern với State Pattern*

Memento Pattern thường được sử dụng cùng với State Pattern để quản lý trạng thái của một đối tượng. Trong State Pattern, đối tượng có thể chuyển đổi giữa các trạng thái khác nhau và thực hiện hành động tương ứng với trạng thái hiện tại của nó. Memento Pattern có thể được sử dụng để lưu trữ và khôi phục các trạng thái trước đó của đối tượng, giúp cho việc chuyển đổi trạng thái trở nên linh hoạt và dễ dàng quản lý.

*1.3.3 Memento Pattern với Iterator Pattern*

Trong một số trường hợp, Memento Pattern có thể được sử dụng cùng với Iterator Pattern để lưu trữ và quản lý trạng thái của các vòng lặp. Trong Iterator Pattern, một đối tượng Iterator được sử dụng để duyệt qua các phần tử của một tập hợp một cách tuần tự. Memento Pattern có thể được sử dụng để lưu trữ trạng thái của vòng lặp tại một điểm nhất định, cho phép người dùng quay lại vị trí trước đó trong vòng lặp nếu cần thiết.

*1.3.4 Memento Pattern với Strategy Pattern*

Memento Pattern có thể được sử dụng cùng với Strategy Pattern để quản lý các chiến lược hoặc cách thức thực hiện một hành động. Trong Strategy Pattern, các chiến lược được biểu diễn dưới dạng các đối tượng riêng biệt, cho phép đối tượng gọi đến các chiến lược khác nhau tùy thuộc vào điều kiện hoặc ngữ cảnh. Memento Pattern có thể được sử dụng để lưu trữ và quản lý các phiên bản trạng thái của các chiến lược, giúp cho việc chuyển đổi giữa các chiến lược trở nên linh hoạt hơn.

*1.3.5 Memento Pattern với Prototype Pattern*

Memento Pattern cũng có mối quan hệ với Prototype Pattern, Prototype Pattern được sử dụng để tạo ra các đối tượng mới bằng cách sao chép các đối tượng đã tồn tại. Memento Pattern có thể được sử dụng để lưu trữ và quản lý trạng thái của các đối tượng, và sau đó, Prototype Pattern có thể được sử dụng để tạo ra các bản sao của đối tượng dựa trên trạng thái đã được lưu trữ.

*1.3.6 Memento Pattern với Observer Pattern*

Memento Pattern cũng có thể liên quan đến Observer Pattern trong một số trường hợp. Trong Observer Pattern, các đối tượng được thông báo khi trạng thái của một đối tượng khác thay đổi. Memento Pattern có thể được sử dụng để lưu trữ và quản lý các phiên bản trạng thái của đối tượng, và sau đó, Observer Pattern có thể được sử dụng để thông báo cho các đối tượng khác về sự thay đổi trong trạng thái của đối tượng.

1.4 So sánh và nghiên cứu tính năng Memento Pattern

*1.4.1 So sánh với các phương pháp khác về tính năng undo và redo*

*1.4.1.1 Memento Pattern*

- Ưu điểm:

+ Memento Pattern tập trung vào việc lưu trữ và quản lý trạng thái của đối tượng, giúp nó trở nên độc lập với cấu trúc lệnh hoặc ghi log.

+ Điều này làm cho Memento Pattern linh hoạt hơn trong việc xử lý các thay đổi trạng thái của đối tượng và tích hợp dễ dàng vào các hệ thống phức tạp.

+ Memento Pattern cho phép lưu trữ nhiều phiên bản trạng thái của đối tượng, từ đó cung cấp khả năng hoàn tác và làm lại đa mức độ.

- Hạn chế:

+ Trong một số trường hợp, việc lưu trữ nhiều phiên bản trạng thái của đối tượng có thể tốn nhiều tài nguyên hơn so với việc lưu trữ lịch sử các hành động đã thực thi như Command Pattern.

+ Memento Pattern có thể trở nên phức tạp trong việc quản lý nếu không được thiết kế và triển khai một cách cẩn thận.

*1.4.1.2 Command Pattern*

- Ưu điểm:

+ Command Pattern tập trung vào việc thực hiện và lưu trữ các hành động hoặc lệnh đã thực thi, giúp dễ dàng hoàn tác và làm lại các thay đổi trên đối tượng.

+ Nó cho phép tách biệt việc thực hiện các hành động và quản lý trạng thái của đối tượng, tăng tính linh hoạt và tái sử dụng mã nguồn.

+ Command Pattern có thể giúp tối ưu hóa việc lưu trữ lịch sử hành động bằng cách chỉ lưu trữ các hành động cần thiết để hoàn tác và làm lại.

- Hạn chế:

+ Command Pattern tập trung vào việc thực hiện các hành động và không lưu trữ trạng thái của đối tượng, điều này có thể gây ra sự phụ thuộc vào cấu trúc hành động cụ thể và khó khăn trong việc quản lý trạng thái đối tượng.

+ Việc lưu trữ lịch sử các hành động có thể tốn nhiều tài nguyên hơn so với việc lưu trữ trạng thái của đối tượng trong Memento Pattern.

*1.4.1.3 Logging*

- Ưu điểm:

+ Logging là một phương pháp đơn giản và tiện lợi để ghi lại các hoạt động và sự kiện quan trọng trong ứng dụng.

+ Nó cho phép lưu trữ các trạng thái hoặc hành động của hệ thống theo thời gian, từ đó giúp xác định và sửa chữa lỗi một cách dễ dàng.

+ Logging có thể được sử dụng song song với Memento Pattern để cung cấp một cơ chế hoàn tác và làm lại phong phú hơn.

- Hạn chế:

+ Logging không tập trung vào việc quản lý trạng thái của đối tượng như Memento Pattern, điều này có thể làm cho quá trình hoàn tác và làm lại trở nên phức tạp hơn.

+ Việc lưu trữ lịch sử các trạng thái hoặc hành động trong log có thể tốn nhiều tài nguyên lưu trữ và có thể gây ra vấn đề về hiệu suất trong các ứng dụng lớn.

*1.4.2 Memento Pattern trong các ngôn ngữ và framework khác nhau*

*1.4.2.1 Java*

Trong Java, việc triển khai Memento Pattern thường đơn giản với các lớp và giao diện tiêu chuẩn của ngôn ngữ, có thể tạo ra một lớp Memento để lưu trữ trạng thái của đối tượng, một lớp Originator để quản lý trạng thái của đối tượng và tạo ra các Memento, và một lớp Caretaker để quản lý các Memento và thực hiện hoàn tác và làm lại.

*1.4.2.2 Python*

Python cũng cung cấp các công cụ và thư viện mạnh mẽ để triển khai Memento Pattern, ta có thể sử dụng tính năng của Python như properties để quản lý trạng thái của đối tượng một cách linh hoạt. Các thư viện như Django hoặc Flask cũng có thể được kết hợp với Memento Pattern để lưu trữ và khôi phục trạng thái của các thành phần trong ứng dụng web.

*1.4.2.3 C++*

Trong C++, có thể sử dụng các tính năng của ngôn ngữ như lớp và tham chiếu để triển khai Memento Pattern. C++ cung cấp sự linh hoạt và hiệu suất cao, đặc biệt là trong việc quản lý bộ nhớ, điều này làm cho Memento Pattern trở nên hiệu quả hơn trong các ứng dụng yêu cầu tốc độ và hiệu suất cao.

*1.4.2.4 .NET Framework*

Trong .NET Framework, có thể sử dụng các tính năng của C# để triển khai Memento Pattern. .NET cung cấp các cơ chế như Serialization để lưu trữ và khôi phục trạng thái của đối tượng một cách dễ dàng. Ta cũng có thể sử dụng các thư viện như Entity Framework trong ứng dụng .NET để quản lý trạng thái của đối tượng.

*1.4.2.5 Spring Framework*

Trong môi trường Java, Spring Framework cung cấp các tính năng và công cụ mạnh mẽ cho việc triển khai Memento Pattern, ta có thể sử dụng các tính năng Dependency Injection và Aspect-Oriented Programming của Spring để quản lý trạng thái của đối tượng và thực hiện hoàn tác và làm lại một cách linh hoạt.

*1.4.3 Nghiên cứu về tối ưu hóa hiệu suất của Memento Pattern*

*1.4.3.1 Xác định yêu cầu hiệu suất*

Trước khi bắt đầu nghiên cứu, quan trọng nhất là phải xác định rõ yêu cầu hiệu suất của ứng dụng. Điều này bao gồm việc xác định các ngưỡng thời gian phản hồi, tải trọng dữ liệu tối đa, và các yêu cầu về tài nguyên hệ thống.

*1.4.3.2 Phân tích và đánh giá hiệu suất hiện tại*

Trước khi áp dụng bất kỳ kỹ thuật tối ưu hóa nào, cần phải hiểu rõ hiệu suất hiện tại của ứng dụng khi sử dụng Memento Pattern. Điều này có thể bao gồm việc đo lường thời gian phản hồi, tài nguyên hệ thống sử dụng (bộ nhớ, CPU), và các chỉ số hiệu suất khác.

*1.4.3.3 Tìm hiểu và áp dụng kỹ thuật tối ưu hóa*

Sau khi xác định yêu cầu hiệu suất và đánh giá hiệu suất hiện tại, tiếp theo là tìm hiểu và áp dụng các kỹ thuật tối ưu hóa. Một số kỹ thuật phổ biến bao gồm:

+ Chỉ lưu trữ trạng thái cần thiết: Thay vì lưu trữ toàn bộ trạng thái của đối tượng, chỉ lưu trữ các trạng thái cần thiết để hoàn tác và làm lại.

+ Nén dữ liệu: Áp dụng các kỹ thuật nén dữ liệu để giảm dung lượng lưu trữ của các Memento, đặc biệt là khi lưu trữ các đối tượng có kích thước lớn.

+ Sử dụng cache: Lưu trữ các trạng thái trung gian trong cache để giảm thời gian truy cập và tăng hiệu suất khi khôi phục trạng thái.

*1.4.3.4 Thực hiện thử nghiệm và đánh giá*

Sau khi áp dụng các kỹ thuật tối ưu hóa, quan trọng là thực hiện các bài kiểm tra và thử nghiệm để đánh giá hiệu quả của chúng. Đo lường lại các chỉ số hiệu suất như thời gian phản hồi và tài nguyên hệ thống sử dụng để đảm bảo rằng hiệu suất đã được cải thiện.

*1.4.3.5 Lặp lại và điều chỉnh*

Quá trình tối ưu hóa hiệu suất không phải là một quy trình một lần duy nhất, mà là một quá trình lặp lại. Sau khi đánh giá hiệu quả của các kỹ thuật tối ưu hóa, cần điều chỉnh và cải tiến chúng để đạt được hiệu suất tối ưu nhất.

*1.4.3.6 Giải quyết các vấn đề tiềm ẩn*

Trong quá trình nghiên cứu và thực hiện tối ưu hóa, có thể phát hiện ra các vấn đề tiềm ẩn hoặc tác động phụ không mong muốn đến hệ thống. Quan trọng là phải giải quyết các vấn đề này một cách kịp thời và hiệu quả để đảm bảo tính ổn định và hiệu suất của ứng dụng.

1.5 Ưu & nhược điểm của Memento Pattern

*1.5.1 Ưu điểm*

- Bảo đảm 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. Giữ cho trạng thái của đối tượng được bảo vệ và không thể truy cập từ bên ngoài, đảm bảo tính bảo mật.

- 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.

- Memento Pattern cung cấp một cơ chế linh hoạt để lưu trữ và khôi phục các trạng thái trước đó của một đối tượng, điều này rất hữu ích trong việc cung cấp khả năng hoàn tác và làm lại trong các ứng dụng.

- Có thể mở rộng Memento Pattern bằng cách thêm các phương thức hoặc thuộc tính mới vào Memento hoặc Originator mà không cần thay đổi cấu trúc của Caretaker hoặc các lớp khác.

- Cung cấp một cơ chế hiệu quả để lưu trữ và truyền dữ liệu qua mạng bằng cách sử dụng các đối tượng Memento.

- Ta có thể tạo ảnh chụp nhanh trạng thái của đối tượng mà không vi phạm tính đóng gói của nó.

- Ta có thể đơn giản hóa mã của người khởi tạo bằng cách cho phép người chăm sóc duy trì lịch sử trạng thái của người khởi tạo.

*1.5.2 Nhược điểm*

- App tiêu thụ nhiều RAM và xử lý nếu clients tạo mementos quá thường xuyên.

- Caretakers phải theo dõi vòng đời của originator để có thể hủy các mementos không dùng nữa.

- Hầu hết các ngôn ngữ hiện đại, hay cụ thể hơn là dynamic programming languages, ví dụ như PHP, Python và Javascript, không thể đảm bảo state bên trong memento được giữ không ai đụng tới.

- Chi phí lưu trữ lớn: Memento Pattern có thể tạo ra một lượng lớn dữ liệu để lưu trữ các trạng thái trước đó của đối tượng, đặc biệt là khi có nhiều thay đổi trạng thái liên tục.

- Trong các ứng dụng lớn, việc quản lý số lượng lớn các Memento có thể trở nên phức tạp, đòi hỏi sự quản lý cẩn thận để tránh lãng phí tài nguyên.

- Khả năng quản lý hiệu suất: Nếu không được quản lý cẩn thận, việc sử dụng Memento Pattern có thể ảnh hưởng đến hiệu suất của ứng dụng, đặc biệt là khi có nhiều trạng thái phức tạp cần lưu trữ.

- Việc triển khai Memento Pattern có thể đòi hỏi một số công sức thiết kế ban đầu, đặc biệt là khi cần tích hợp tính năng hoàn tác và làm lại vào một ứng dụng hiện có.

- Hầu hết các ngôn ngữ lập trình động, như PHP, Python và JavaScript, không thể đảm bảo rằng trạng thái bên trong Memento vẫn nguyên vẹn.

1.6 Cách sử dụng và Những trường hợp nên sử dụng Mementor Pattern

*1.6.1 Cách sử dụng Mementor Pattern*

*1.6.1.1 Khi nào nên sử dụng Memetor Pattern*

- Chức năng hoàn tác: Khi cần triển khai tính năng hoàn tác trong ứng dụng của mình, cho phép người dùng hoàn nguyên các thay đổi đã thực hiện đối với trạng thái của đối tượng.

- Chụp nhanh (snapshot): Khi cần lưu trạng thái của một đối tượng tại nhiều thời điểm khác nhau để hỗ trợ các tính năng như lập phiên bản hoặc điểm kiểm tra.

- Khôi phục giao dịch: Khi cần khôi phục các thay đổi về trạng thái của đối tượng trong trường hợp có lỗi hoặc ngoại lệ, chẳng hạn như trong các giao dịch cơ sở dữ liệu.

- Bộ nhớ đệm: Khi muốn lưu trữ trạng thái của một đối tượng vào bộ nhớ đệm để cải thiện hiệu suất hoặc giảm các tính toán dư thừa.

*1.6.1.2 Khi nào không nên sử dụng Memetor Pattern*

- Trạng thái đối tượng lớn: Nếu trạng thái của đối tượng lớn hoặc phức tạp, việc lưu trữ và quản lý nhiều ảnh chụp nhanh trạng thái của đối tượng có thể tiêu tốn một lượng đáng kể bộ nhớ và tài nguyên xử lý.

- Thay đổi trạng thái thường xuyên: Nếu trạng thái của đối tượng thay đổi thường xuyên và không thể đoán trước, việc lưu trữ và quản lý ảnh chụp nhanh trạng thái của đối tượng có thể trở nên không thực tế hoặc không hiệu quả.

- Đối tượng bất biến: Nếu trạng thái của đối tượng là bất biến hoặc dễ dàng tái tạo lại thì việc sử dụng mẫu Memento để nắm bắt và khôi phục trạng thái của nó có thể mang lại rất ít lợi ích.

- Chi phí chung: Việc giới thiệu mẫu Memento có thể tăng thêm độ phức tạp cho cơ sở mã, đặc biệt nếu ứng dụng không yêu cầu các tính năng như chức năng hoàn tác hoặc khôi phục trạng thái.

*1.6.2 Những trường hợp nên sử dụng Mementor Pattern*

*1.6.2.1 Tính Năng Hoàn Tác và Làm Lại (Undo/Redo)*

Trong các ứng dụng văn bản, đồ họa hoặc chỉnh sửa hình ảnh, tính năng hoàn tác và làm lại là rất quan trọng để người dùng có thể điều chỉnh các hành động của họ một cách linh hoạt. Memento Pattern cho phép lưu trữ các trạng thái trước đó của một đối tượng và khôi phục lại chúng khi cần thiết, giúp hỗ trợ tính năng hoàn tác và làm lại một cách dễ dàng.

*1.6.2.2 Bảo vệ Tính Bao Quát và Tính Cá nhân (Encapsulation)*

Memento Pattern giúp duy trì tính đóng gói (encapsulation) bằng cách ngăn chặn truy cập trực tiếp vào trạng thái của đối tượng từ bên ngoài. Thay vì truy cập trực tiếp vào trạng thái của đối tượng, các thay đổi trạng thái được thực hiện thông qua các phương thức được cung cấp bởi Memento Pattern, giúp bảo vệ tính toàn vẹn của dữ liệu.

*1.6.2.3 Duy Trì Tính Toàn Vẹn của Dữ Liệu*

Trong các hệ thống cơ sở dữ liệu hoặc hệ thống giao dịch, việc duy trì tính toàn vẹn và tính nhất quán của dữ liệu là rất quan trọng. Memento Pattern cho phép lưu trữ và quản lý các phiên bản trạng thái của một đối tượng một cách hiệu quả, giúp duy trì tính toàn vẹn của dữ liệu trong các tình huống phức tạp.

*1.6.2.4 Quản Lý Trạng Thái Phức Tạp*

Trong các đối tượng có trạng thái phức tạp hoặc các đối tượng mà trạng thái thay đổi thường xuyên, việc sử dụng Memento Pattern giúp lưu trữ và quản lý các trạng thái trước đó một cách hiệu quả. Điều này làm cho việc quản lý trạng thái của đối tượng trở nên dễ dàng hơn và giảm thiểu các lỗi liên quan đến việc quản lý trạng thái.

*1.6.2.5 Tối Ưu Hóa Hiệu Suất*

Memento Pattern cung cấp một cơ chế linh hoạt để tối ưu hóa hiệu suất và quản lý tài nguyên. Bằng cách chỉ lưu trữ các trạng thái quan trọng hoặc cần thiết, Memento Pattern giúp giảm thiểu khả năng tiêu tốn tài nguyên không cần thiết và cải thiện hiệu suất của ứng dụng.

1.7 Ứng dụng của Memento Pattern

*1.7.1 Ứng dụng Desktop và ứng dụng web*

Trong các ứng dụng desktop hoặc web, Memento Pattern có thể được sử dụng để cung cấp tính năng hoàn tác và làm lại cho người dùng. Điều này giúp người dùng có thể điều chỉnh hoạt động của họ một cách linh hoạt và tiện lợi.

*1.7.2 Chỉnh sửa văn bản và đồ họa*

Trong các ứng dụng chỉnh sửa văn bản, đồ họa hoặc chỉnh sửa hình ảnh, Memento Pattern có thể được sử dụng để lưu trữ các phiên bản trạng thái trước đó của tài liệu hoặc hình ảnh. Điều này cho phép người dùng hoàn tác và làm lại các thay đổi một cách dễ dàng.

*1.7.3 Hệ thống giao dịch tài chính*

Trong các hệ thống cơ sở dữ liệu hoặc hệ thống giao dịch, Memento Pattern có thể được sử dụng để lưu trữ và quản lý các phiên bản trạng thái của dữ liệu hoặc giao dịch. Điều này giúp đảm bảo tính nhất quán và tính toàn vẹn của dữ liệu trong các tình huống phức tạp.

*1.7.4 Ứng dụng trò chơi và giải trí*

Trong các ứng dụng trò chơi và giải trí, Memento Pattern có thể được sử dụng để lưu trữ và quản lý trạng thái của trò chơi hoặc các tác vụ giải trí. Điều này cho phép người chơi hoặc người dùng giải trí điều chỉnh trạng thái của họ một cách linh hoạt và tiện lợi.

*1.7.5 Ứng dụng trong Machine Learning và AI*

Trong các ứng dụng Machine Learning và AI, Memento Pattern có thể được sử dụng để lưu trữ và quản lý các trạng thái của mô hình máy học hoặc thuật toán., lưu trữ và quản lý trạng thái của mô hình máy học hoặc thuật toán. Điều này giúp các nhà phát triển có thể thử nghiệm và điều chỉnh mô hình của mình một cách dễ dàng, đồng thời giữ cho các trạng thái trung gian của mô hình được bảo toàn.

*1.7.6 Hệ thống tích hợp và xử lý sự kiện*

Trong các hệ thống tích hợp và xử lý sự kiện, Memento Pattern có thể được sử dụng để lưu trữ và quản lý trạng thái của các sự kiện hoặc quy trình tích hợp. Điều này giúp đảm bảo tính nhất quán và tính toàn vẹn của dữ liệu trong quá trình xử lý sự kiện.

1.8 Ví dụ cụ thể của Memento Pattern

*1.8.1 Sự tương đồng trong thế giới thực của mẫu thiết kế Memento*

Originator (Nghệ sĩ):

+ Đây là bạn, nghệ sĩ, chịu trách nhiệm tạo ra và quản lý bức tranh của mình.

Khi bạn đang vẽ, bạn có thể chụp ảnh nhanh (Memento) để ghi lại trạng thái hiện tại của bức tranh.

Memento (Bức ảnh):

+ Một Memento là một bức ảnh chụp bức tranh của bạn tại một thời điểm cụ thể.

+ Tương tự như việc chụp ảnh để ghi lại diện mạo của bức tranh tại thời điểm đó, Memento ghi lại trạng thái của bức tranh.

Caretaker (Người sưu tầm nghệ thuật):

+ Caretaker là như một người sưu tập nghệ thuật, giúp bạn quản lý các bức tranh và Memento của chúng.

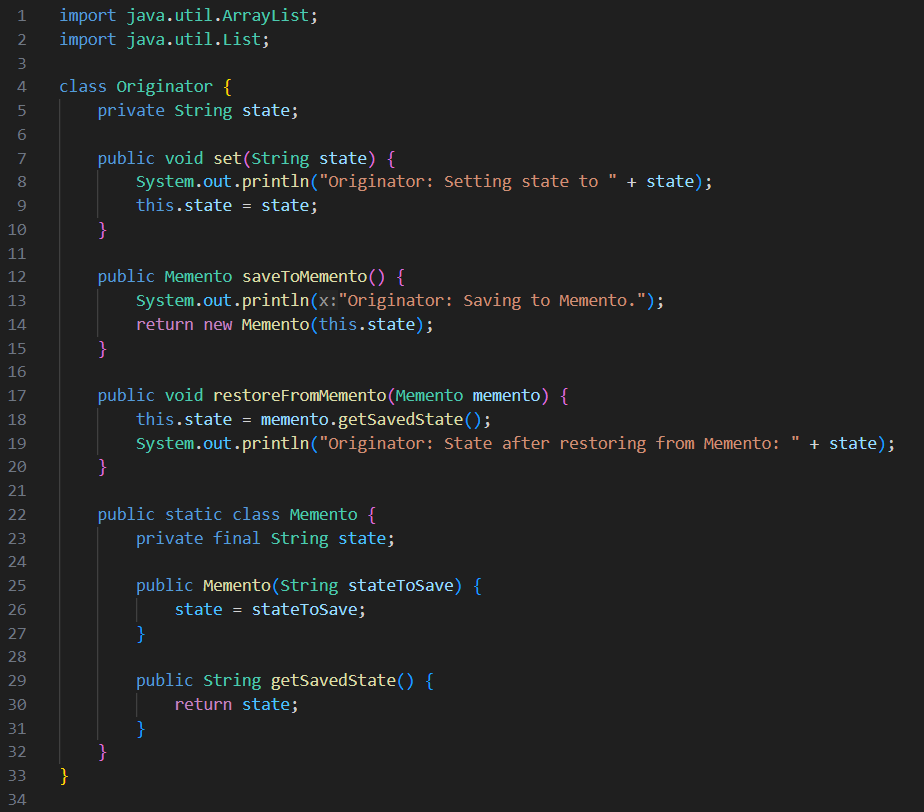
+ Họ không vẽ bức tranh như bạn, nhưng họ theo dõi và lưu trữ Memento.

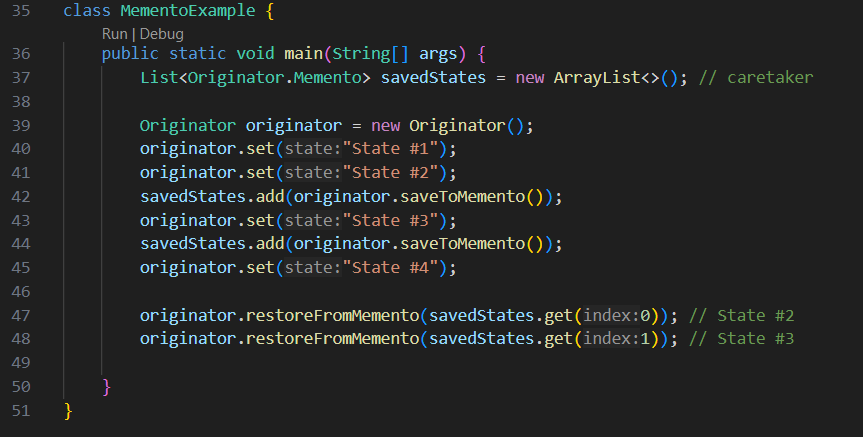
+ Họ có thể gắn nhãn cho mỗi bức ảnh với ngày tháng hoặc mô tả để bạn có thể nhận biết chúng dễ dàng.

+ Khi bạn muốn xem lại phiên bản trước đó của bức tranh, bạn chỉ cần yêu cầu Caretaker lấy Memento tương ứng để xem hoặc khôi phục.

*1.8.2 Demo nhỏ về Memento Pattern quản lý trạng thái của đối tượng*

*1.8.2.1 Code*





**5. Code của demo nhỏ Memento với chức năng quản lý trạng thái**

*1.8.2.2 Giới thiệu mô hình*

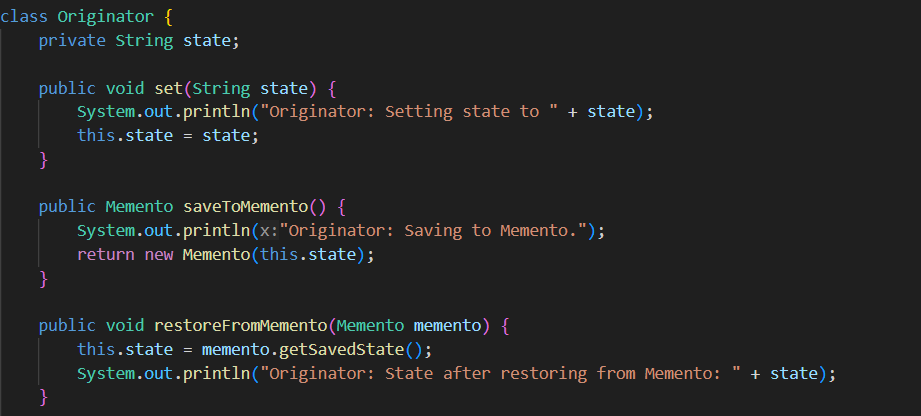
Mô hình trên là một ví dụ về Memento Pattern, một trong những mẫu thiết kế hành vi trong lập trình hướng đối tượng. Memento Pattern cho phép bạn lưu trữ và khôi phục trạng thái của một đối tượng mà không tiết lộ chi tiết cấu trúc nội bộ của nó.

- Các thành phần trong mô hình:

+ Originator (Nguyên tạo):

Đây là lớp chứa trạng thái dùng để lưu trữ hoặc khôi phục(undo/redo), Originator đại diện cho đối tượng có trạng thái cần lưu trữ. Phương thức set được sử dụng để đặt trạng thái mới của Originator, và phương thức saveToMemento được sử dụng để tạo ra một Memento chứa trạng thái hiện tại của Originator.

Phương thức restoreFromMemento được sử dụng để khôi phục trạng thái của Originator từ một Memento đã cho.

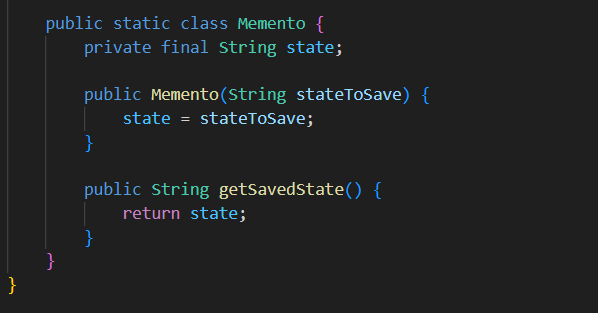


**6. Originator trong mô hình quản lý trạng thái**

+ Memento (Bảo vệ):

Đây là một lớp chứa trạng thái của Originator, Memento đóng vai trò là một bản sao của trạng thái hiện tại của Originator tại một thời điểm cụ thể.

Trong Memento, có một phương thức getSavedState được sử dụng để lấy trạng thái đã lưu.



**7. Memento trong mô hình quản lý trạng thái**

+ Caretaker (Người chăm sóc):

Đây là một lớp hoặc đối tượng chịu trách nhiệm quản lý các Memento, Caretaker duy trì một danh sách các Memento đã lưu trữ của Originator.

Biến savedStates là một danh sách (List) của các đối tượng Memento của Originator, Caretaker cung cấp phương thức để thêm mới Memento vào danh sách (saveDocumentState), hoàn tác trạng thái trước đó (undo), và làm lại trạng thái tiếp theo (redo).



**8. Caretaker trong mô hình quản lý trạng thái**

*1.8.2.3 Cách thức mô hình hoạt động*

- Đầu tiên, Originator thay đổi trạng thái của nó và tạo một Memento để lưu trữ trạng thái đó.

- Memento được chuyển cho Caretaker, nơi nó được lưu trữ trong danh sách.

- Khi cần, Originator có thể khôi phục trạng thái trước đó bằng cách sử dụng Memento đã lưu trữ trong danh sách bởi Caretaker.

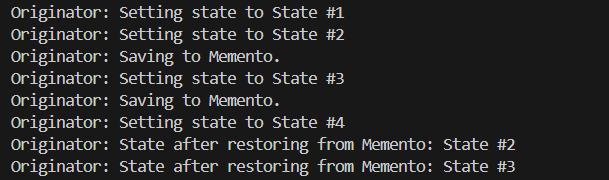
- Trong mô hình, Originator set các trạng thái “State #1”, “State #2”, “State #3” và “State #4”. Phương thức setMemento() dùng để lưu trạng thái “State” cần thiết

- Sau mỗi lần thay đổi trạng thái, ta lưu trạng thái hiện tại của Originator vào một Memento và thêm vào danh sách savedStates.

- Cuối cùng, khôi phục trạng thái của Originator từ các Memento trong danh sách savedStates bằng cách sử dụng phương thức restoreFromMemento.

- savedStates.get(0) trả về Memento chứa trạng thái "State #2". Tương tự, savedStates.get(1) trả về Memento chứa trạng thái "State #3".

- Do khi gọi originator.restoreFromMemento(savedStates.get()) nó sẽ get giá trị trong danh sách và giá được danh sách lần lượt vị trí 0 là “State #2”, vị trí 1 là “State #3”.



**9. Kết quả thực thi của mô hình**

1.9 Source code Memento Pattern

*1.9.1 Giới thiệu về đề tài*

Đề tài: Ứng dụng quản lý tài liệu văn bản

Vấn đề đặt ra:

- Giả sử chúng ta đang phát triển một ứng dụng quản lý tài liệu văn bản, nơi người dùng có thể tạo, chỉnh sửa và lưu trữ các tài liệu. Một tính năng quan trọng mà chúng ta muốn thêm vào ứng dụng là khả năng hoàn tác (undo) và làm lại (redo) các thao tác chỉnh sửa.

Giải pháp:

- Trong trường hợp này, chúng ta có thể sử dụng Memento Pattern để lưu trạng thái của các tài liệu và cho phép người dùng hoàn tác hoặc làm lại các thay đổi. Dưới đây là cách chúng ta có thể triển khai Memento Pattern trong ứng dụng này:

+ Originator (Nguyên tạo): Đây là đối tượng chứa tài liệu văn bản và có khả năng tạo ra Memento để lưu trạng thái hiện tại của tài liệu và khôi phục tài liệu từ một Memento đã lưu trữ. Trong ví dụ này, Originator là một lớp đại diện cho tài liệu văn bản.

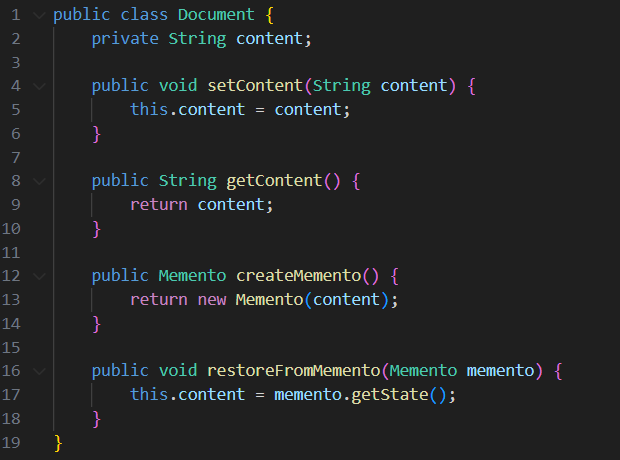
+ Memento (Bảo vệ): Đây là đối tượng lưu trữ trạng thái của tài liệu tại một thời điểm nhất định. Trong ứng dụng này, Memento có thể là một lớp đại diện cho trạng thái của tài liệu văn bản tại thời điểm cụ thể.

+ Caretaker (Người chăm sóc): Đây là đối tượng quản lý các Memento đã tạo ra và cung cấp khả năng hoàn tác (undo) và làm lại (redo) các thao tác chỉnh sửa. Caretaker sẽ duy trì một danh sách các Memento đã tạo ra và cung cấp phương thức để thêm mới, lấy ra và sử dụng các Memento.

Trong đề tài này, triển khai Memento Pattern để cho phép người dùng hoàn tác và làm lại các thao tác chỉnh sửa trên tài liệu văn bản trong ứng dụng quản lý tài liệu. Memento Pattern giúp tách biệt việc lưu trữ trạng thái của tài liệu và quản lý việc hoàn tác và làm lại, làm cho hệ thống trở nên linh hoạt và dễ bảo trì.

*1.9.2 Mô hình đề tài*

*1.9.2.1 Document (Đại diện cho Originator)*

**

**10. Originator của đề tài quản lý văn bản**

Trong đoạn mã trên, lớp Document đại diện cho nguyên tắc mà chúng ta muốn lưu trữ và khôi phục trạng thái của nó bằng Mementor Patern, đoạn mã trên cung cấp các phương thức cơ bản để quản lý trạng thái của một tài liệu và tạo ra các Memento để lưu trữ trạng thái của nó.

- setContent(String content):

+ Phương thức này được sử dụng để đặt nội dung mới cho tài liệu.

+ Khi được gọi, nội dung của tài liệu sẽ được thay đổi thành giá trị được chuyển vào.

- getContent():

+ Phương thức này được sử dụng để lấy nội dung hiện tại của tài liệu.

+ Khi được gọi, phương thức này trả về nội dung hiện tại của tài liệu.

- createMemento():

+ Phương thức này được sử dụng để tạo ra một đối tượng Memento mới từ trạng thái hiện tại của tài liệu.

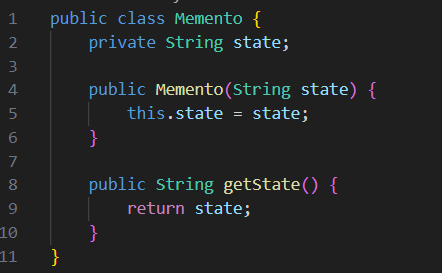
+ Khi được gọi, phương thức này tạo ra một đối tượng Memento mới chứa nội dung hiện tại của tài liệu và trả về nó.

- restoreFromMemento(Memento memento):

+ Phương thức này được sử dụng để khôi phục trạng thái của tài liệu từ một đối tượng Memento đã cho.

+ Khi được gọi, phương thức này sẽ gán nội dung của tài liệu bằng nội dung được lưu trữ trong đối tượng Memento đã cho. Điều này có nghĩa là tài liệu sẽ có trạng thái tương tự như trạng thái đã lưu trữ trong Memento.

*1.9.2.2 Memento (Đại diện cho Memento)*

**

**11. Memento của đề tài quản lý văn bản**

Lớp Memento trong Memento Pattern là nơi lưu trữ trạng thái của một đối tượng mà chúng ta muốn lưu trữ và khôi phục sau này. Trong trường hợp này, nó chứa một trường dữ liệu để lưu trữ trạng thái và cung cấp phương thức để truy cập và nhận trạng thái đó.

- Trường dữ liệu state:

+ Đây là trường dữ liệu để lưu trữ trạng thái của đối tượng.

+ Trong đoạn mã, state là một chuỗi (String) và chứa trạng thái của đối tượng.

- Phương thức khởi tạo Memento(String state):

+ Đây là phương thức khởi tạo của lớp Memento, được sử dụng để tạo ra một đối tượng Memento mới với trạng thái được cung cấp.

+ Khi được gọi, phương thức này tạo ra một đối tượng Memento mới với trạng thái được chuyển vào từ tham số.

- Phương thức getState():

+ Đây là phương thức được sử dụng để truy cập trạng thái của đối tượng.

+ Khi được gọi, phương thức này trả về giá trị của trường dữ liệu state, tức là trạng thái hiện tại của đối tượng.

*1.9.2.3 Editor (Đại diện cho Caretaker)*

**

**12. Caretaker của đề tài quản lý văn bản**

Trong đoạn mã trên, lớp Editor đóng vai trò là Caretaker trong mẫu thiết kế Memento Pattern. Lớp Editor này cung cấp các phương thức để lưu trạng thái của Document, hoàn tác và làm lại các thay đổi trên Document sử dụng Memento đã lưu trữ.

- List<Memento> mementos: Đây là một danh sách chứa các đối tượng Memento, nơi Caretaker lưu trữ các trạng thái của Document khi chúng được lưu lại.

- int currentIndex: Biến này đánh dấu chỉ số hiện tại của trạng thái đã lưu trong danh sách mementos.

- saveDocumentState(Document document): Phương thức này được sử dụng để lưu trạng thái hiện tại của Document.

+ Trước tiên, nó gọi phương thức createMemento() của Document để tạo ra một Memento mới chứa trạng thái hiện tại của Document.

+ Sau đó, Memento này được thêm vào danh sách mementos.

+ Chỉ số hiện tại currentIndex được tăng lên một để chỉ đến vị trí mới nhất trong danh sách mementos.

- undo(Document document): Phương thức này được sử dụng để hoàn tác trạng thái của Document về trạng thái trước đó.

+ Nếu chỉ số hiện tại currentIndex lớn hơn 0 (tức là có ít nhất một trạng thái đã lưu trước đó), phương thức này sẽ giảm chỉ số currentIndex đi một và khôi phục Document từ Memento tại vị trí currentIndex trong danh sách mementos bằng cách gọi phương thức restoreFromMemento() của Document.

- redo(Document document): Phương thức này được sử dụng để làm lại trạng thái của Document.

+ Nếu chỉ số hiện tại currentIndex nhỏ hơn kích thước của danh sách mementos - 1 (tức là có ít nhất một trạng thái đã lưu và có thể làm lại), phương thức này sẽ tăng chỉ số currentIndex lên một và khôi phục Document từ Memento tại vị trí currentIndex trong danh sách memento.

*1.9.2.4 Hàm main*



**13. Hàm Main của đề tài quản lý văn bản**

Lớp Main chứa phương thức main để thực thi chương trình và kiểm tra các chức năng của Memento Pattern. Tóm lại, trong phương thức main, chúng ta thực hiện một loạt các thao tác chỉnh sửa, hoàn tác và làm lại trên Document để kiểm tra và xác nhận tính chính xác của việc quản lý trạng thái bằng Memento Pattern.

- Tạo đối tượng Document và Editor:

+ Đầu tiên, tạo một đối tượng Document để đại diện cho tài liệu cần quản lý trạng thái.

+ Tiếp theo, tạo một đối tượng Editor, đóng vai trò là Caretaker, để quản lý lưu trữ và khôi phục trạng thái của Document.

- Thiết lập và lưu trạng thái ban đầu:

+ Thiết lập nội dung ban đầu của Document bằng cách gọi phương thức setContent.

+ Sau đó lưu trạng thái ban đầu này của Document bằng cách gọi phương thức saveDocumentState của Editor.

- Chỉnh sửa và lưu trạng thái sau mỗi lần chỉnh sửa:

+ Thay đổi nội dung của Document và lưu trạng thái mới sau mỗi lần chỉnh sửa bằng cách gọi setContent và saveDocumentState.

- Hoàn tác (Undo):

+ Thực hiện các lệnh undo bằng cách gọi phương thức undo của Editor.

+ Sau mỗi lần undo, chúng ta in ra nội dung của Document để kiểm tra trạng thái sau mỗi lần hoàn tác.

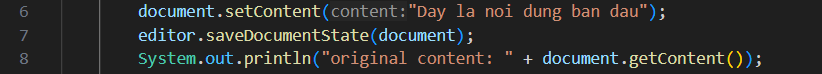
- Làm lại (Redo):

+ Thực hiện các lệnh redo bằng cách gọi phương thức redo của Editor.

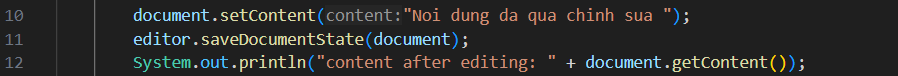
Sau mỗi lần redo, chúng ta in ra nội dung của Document để kiểm tra trạng thái sau mỗi lần làm lại.

*1.9.3 Kết quả đề tài*

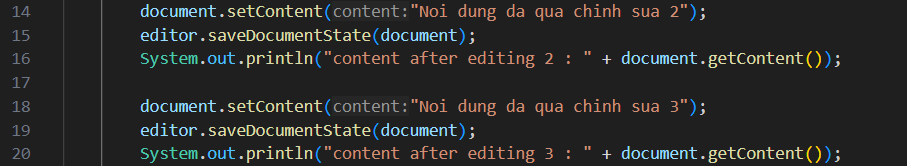
- Ở dòng thứ 6 của class Main sẽ set nội dung ban đầu của Document là : “Day la noi dung ban dau”.



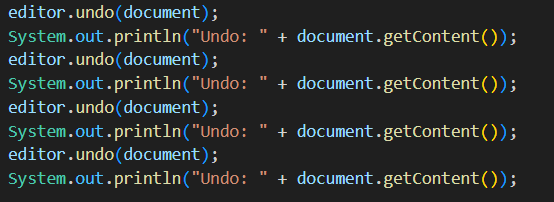
- Dòng thứ 10 của class Main qua chỉnh sửa nội dung lần 1 của Document thành : “Noi dung da qua chinh sua”.



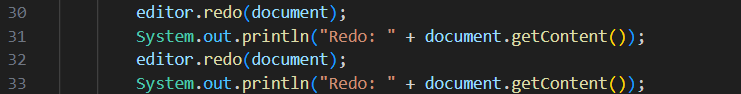
- Tương tự thế dòng 14 và 18 chinh sửa dung thành lần 2 và 3



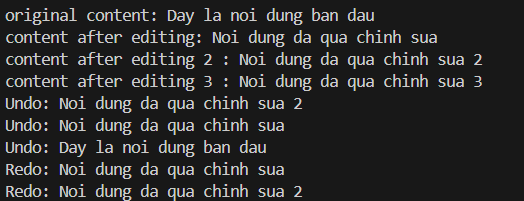
- Với chức năng undo nó sẽ quay lại văn bản trước khi chỉnh sửa 1 bậc. Ví dụ hiện tại nội dung đang là: “Noi dung da qua chinh sua 3”, nó sẽ quay lại nội dung trước đó: “Noi dung da qua chinh sua 2”. Nếu tiếp tục gọi phương thức undo liên tục nó sẽ liên lùi về 1 bậc cho đến khi đến văn bản gốc là “Day la noi dung ban dau”.



- Với phương thức redo nó dùng tiến lên 1 bậc của văn bản trước sau khi undo. Ví dụ ở phương undo đã được sử dụng 3 lần khiến văn lùi về 3 bậc, quay lại văn bản đầu tiên: “Day la noi dung ban dau”. Nếu sau khi tiến 1 bậc với phương thức redo nó sẽ đi đến văn bản: “Noi dung da qua chinh sua”. Tiến 2 bậc sẽ đi đến: “Noi dung da qua chinh sua 2”. Tương tự thực hiện thêm nhiều phương thức redo, chúng ta sẽ đi đến văn bản mới nhất hiện tại



Kết quả thực thi:



**14. Kết quả của đề tài undo/redo**

Giải thích ở kết quả trên:

- Dòng thứ nhất là nội dung văn bản đầu tiên: “Day la noi dung ban dau”

- Tương tự thế dòng 2, 3, 4 là nội dung qua 3 lần chỉnh sửa khác nhau

- Dòng thứ 5 dùng để undo từ nội dùng văn bản dòng thứ 4 về 3

- Dòng thứ 6 dùng để undo từ nội dùng văn bản dòng thứ 3 về 2

- Dòng thứ 7 dùng để undo từ nội dùng văn bản dòng thứ 2 về nội dung ban đầu

- Dòng thứ 9 dùng để redo từ nội dung ban đầu lên văn bản dòng thứ 2

- Dòng thứ 10 dùng để redo từ nội dung văn bản dòng thứ 2 lên văn bản dòng thứ 3

CHƯƠNG 2 – MEDIATOR PATTERN

2.1 Giới thiệu Mediator Pattern

*2.1.1 Khái niệm Mediator Pattern*

- Phân loại: Nhóm hành vi (Behavior Pattern)

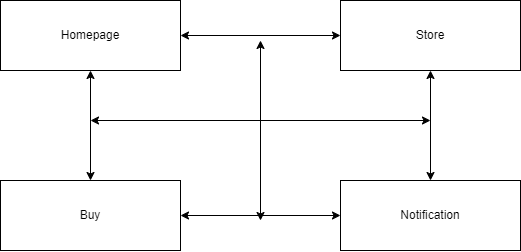
- Mediator Pattern là một mẫu thiết kế hành vi cho phép giảm bớt sự phụ thuộc giữa các đối tượng. Hạn chế liên lạc trực tiếp giữa các đối tượng và buộc chúng chỉ cộng tác thông qua một đối tượng trung gian. Có thể hiểu Mediator : là người trung gian.

- Mediator thúc đẩy sự khớp nối lỏng lẻo (loose coupling), bằng cách ngăn không cho các đối tượng đề cập đến nhau một cách rõ ràng và nó cho phép thay đổi sự tương tác của họ một cách độc lập”.

- Mediator Pattern khá giống như Adapter Pattern nhưng sử dụng trong mục đích khác, 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.

*2.1.2 Mục đích ra đời*

*2.1.2.1 Vấn đề*

**

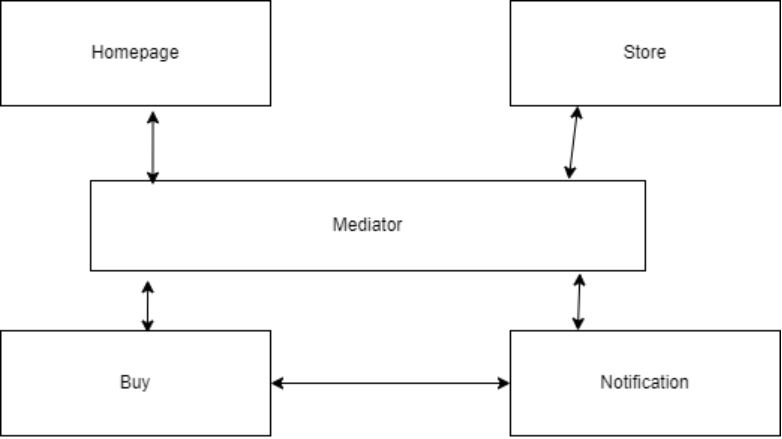
**15. Vấn đề Mediator Pattern**

Giả sử Flow chart demo ta đang có một website với 4 trang. Website cho phép khách hàng duyệt qua kho hàng và đặt mua. Khách hàng có thể đi từ trang này qua trang khác theo đường vẽ trên hình. Ở đây có một vấn đề phát sinh. Khi nào khách muốn từ page này sang page khác thì phải kích hoạt page đó nên phải.

Tại từng trang, phải viết mã để nhận biết. Tại một trang có quá nhiều đường để đi tới trang khác => phát sinh nhiều đoạn code trùng lặp ở nhiều chỗ

***2.1.2.2 Giải pháp***

Bây giờ chúng ta dùng Mediator Pattern để đóng gói các đường dẫn tới trang vào một module duy nhất. Từ bây giờ, từng trang chỉ cần phải thông báo bất cứ sự thay đổi nào cho Mediator, và Mediator tự biết điều hướng trang cần thiết cho khách hàng, như trong hình bên dưới.

****

**16. Giải pháp cho Mementor Pattern**

Chúng ta có thể tạo ra một Mediator với chức năng điều hướng trang. Tại đây chúng ta có thể chỉnh sửa và thay đổi dễ dàng. Đó chính là chức năng của Mediator (Người trung gian).

*2.1.3 Cách triển khai Mediator Pattern*

- Xác định một nhóm các lớp được liên kết chặt chẽ sẽ được hưởng lợi từ việc độc lập hơn (ví dụ: để bảo trì dễ dàng hơn hoặc tái sử dụng các lớp này đơn giản hơn).

- Khai báo giao diện mediator và mô tả giao thức liên lạc mong muốn giữa mediators và các thành phần khác nhau. Trong hầu hết các trường hợp, chỉ cần một phương pháp duy nhất để nhận thông báo từ các thành phần là đủ.

- Giao diện này rất quan trọng khi ta muốn sử dụng lại các lớp thành phần trong các ngữ cảnh khác nhau. Miễn là thành phần này hoạt động với mediator của nó thông qua giao diện chung, ta có thể liên kết thành phần đó với cách triển khai mediator khác.

- Thực hiện class mediator cụ thể, xem xét việc lưu trữ các tham chiếu đến tất cả các thành phần bên trong mediator. Bằng cách này, ta có thể gọi bất kỳ thành phần nào từ các phương thức của người mediator’s .

- Ta có thể tiến xa hơn nữa và yêu cầu mediator chịu trách nhiệm tạo và hủy các đối tượng thành phần. Sau đó, mediator có thể giống một nhà máy hoặc một mặt tiền.

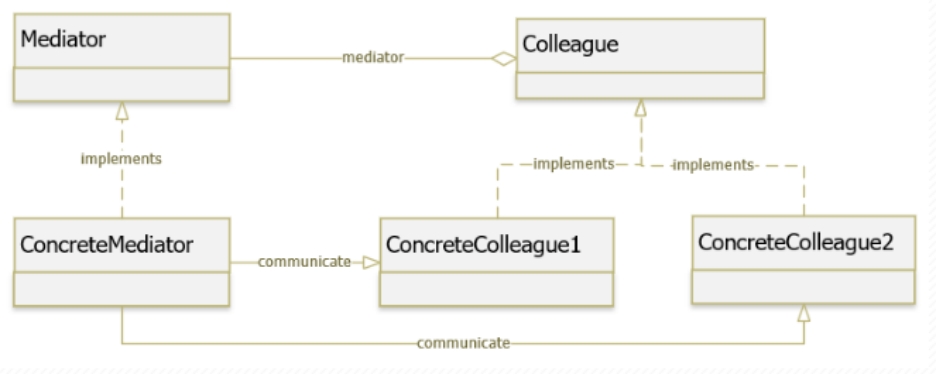
- Các thành phần nên lưu trữ một tham chiếu đến đối tượng mediator. Kết nối thường được thiết lập trong hàm tạo của thành phần, trong đó một đối tượng mediator được truyền dưới dạng đối số.

- Thay đổi code của các thành phần để chúng gọi phương thức thông báo mediator’s thay vì các phương thức trên các thành phần khác.

## **2.2 Kiến trúc Mediator Pattern**

### ***2.2.1 Sơ lược về kiến trúc Mediator Pattern***

Pattern Mediator được sử dụng để giảm sự phụ thuộc giữa các thành phần trong một hệ thống phần mềm bằng cách tạo ra một đối tượng trung gian để điều phối giao tiếp giữa chúng. Bằng cách này, Mediator giúp các thành phần không cần phải biết về nhau cụ thể mà chỉ tương tác thông qua một điểm trung gian. Điều này giúp tăng tính linh hoạt và dễ bảo trì của hệ thống, vì các thành phần có thể thay đổi mà không làm ảnh hưởng đến nhau, và Mediator có thể thay đổi mà không cần sửa đổi nhiều trong các thành phần khác.

****

**17. Kiến trúc Mediator Pattern**

*2.2.2 Chi tiết về kiến trúc Mediator Pattern*

*2.2.2.1 Colleague*

- Colleague là các thành phần trong hệ thống cần giao tiếp với nhau. Mỗi Colleague định nghĩa một giao diện cho việc gửi và nhận thông điệp thông qua Mediator. Thay vì giao tiếp trực tiếp với các thành phần khác, Colleague gửi thông điệp thông qua Mediator, điều này giúp tách biệt logic của chúng và giảm sự phụ thuộc giữa các thành phần.

*2.2.2.2 ConcreteColleague*

- ConcreteColleague cài đặt các phương thức của Colleague, giao tiếp thông qua Mediator khi cần giao tiếp với Colleague khác.

- ConcreteColleague là lớp triển khai của Colleague. Nó không chỉ triển khai các phương thức đã được định nghĩa trong Colleague mà còn thực hiện các nhiệm vụ cụ thể của mình. ConcreteColleague gửi và nhận thông điệp từ các thành phần khác trong hệ thống thông qua Mediator. Nó thực hiện các tác vụ cụ thể của mình và sử dụng Mediator để giao tiếp với các đồng nghiệp khác một cách an toàn và đồng bộ.

*2.2.2.3 Mediator*

- Mediator là một interface, định nghĩa các phương thức để giao tiếp với các Colleague object.

- Mediator là một abstract class, giữ tham chiếu đến Mediator object, đóng vai trò là một trung gian giữa các thành phần trong hệ thống. Nó định nghĩa một giao diện để các thành phần có thể giao tiếp với nhau mà không cần biết về nhau cụ thể. Mediator không giao tiếp trực tiếp với các thành phần khác, thay vào đó, nó cung cấp một cơ chế chung cho các Colleague để gửi và nhận thông điệp. Mediator làm nhiệm vụ điều phối giao tiếp giữa các thành phần một cách an toàn và linh hoạt.

*2.2.2.4 ConcreteMediator*

- ConcreteMediator cài đặt các phương thức của Mediator, biết và quản lý các Colleague object.

- ConcreteMediator là lớp triển khai của Mediator. Nó không chỉ triển khai các phương thức định nghĩa trong Mediator mà còn quản lý các thành phần trong hệ thống. ConcreteMediator có nhiệm vụ chính là xử lý giao tiếp giữa các thành phần thông qua việc sử dụng các phương thức đã được định nghĩa trong Mediator. Nó giữ danh sách các Colleague và chịu trách nhiệm điều phối giao tiếp giữa chúng, đảm bảo rằng các thông điệp được gửi và nhận một cách đúng đắn.

2.3 Mối quan hệ của Mediator Pattern với các Pattern khác

*2.3.1 Mediator Pattern với Observer Pattern*

Pattern Mediator thường được kết hợp với Observer Pattern để xây dựng các hệ thống có khả năng theo dõi và thông báo sự thay đổi trạng thái của các đối tượng. Trong một cấu trúc tích hợp Observer-Mediator, Mediator có thể là một Observer của các đối tượng cụ thể (ConcreteColleague). Khi trạng thái của một ConcreteColleague thay đổi, nó thông báo cho Mediator, sau đó Mediator gửi thông điệp tới các đối tượng khác thông qua giao diện đã được định nghĩa bởi Observer Pattern.

*2.3.2 Mediator Pattern với Singleton Pattern*

Trong một số trường hợp, đối tượng Mediator có thể được triển khai như một Singleton. Điều này đảm bảo rằng chỉ có một đối tượng Mediator duy nhất trong hệ thống, giúp đồng bộ hóa việc giao tiếp giữa các thành phần. Việc triển khai Mediator như một Singleton cũng giúp tiết kiệm tài nguyên và đơn giản hóa việc quản lý.

*2.3.3 Mediator Pattern với Facade Pattern*

Mediator có thể được sử dụng như một Facade cho các thành phần trong hệ thống. Thay vì các thành phần phải tương tác trực tiếp với nhau, chúng có thể tương tác thông qua Mediator, như là một điểm truy cập đơn giản và giao diện chung. Việc sử dụng Mediator như một Facade giúp giảm sự phức tạp và giảm sự phụ thuộc giữa các thành phần trong hệ thống.

*2.3.4 Mediator Pattern với Command Pattern*

Các thông điệp gửi qua Mediator có thể được thực hiện bằng cách sử dụng Command Pattern. Mỗi thông điệp có thể được coi là một lệnh và được đóng gói trong một đối tượng Command. Khi một thông điệp được gửi đến Mediator, Mediator có thể chuyển lệnh đó đến các ConcreteColleague để thực hiện các hành động tương ứng.

*2.3.5 Mediator Pattern với Factory Pattern*

Trong một số trường hợp, các đối tượng Colleague có thể được tạo ra bằng cách sử dụng Factory Pattern. Mediator có thể sử dụng một Factory để tạo các đối tượng Colleague và quản lý việc tạo ra chúng. Điều này giúp tạo ra một cơ chế linh hoạt cho việc tạo ra các đối tượng và giảm sự phụ thuộc vào cách triển khai cụ thể của các đối tượng.

2.4 Mediator Pattern trong các ngôn ngữ và framework khác nhau

*2.4.1 Java*

- Mediator interface sẽ định nghĩa các phương thức cho việc giao tiếp giữa các Colleague.

- ConcreteMediator là lớp triển khai của Mediator, quản lý các Colleague và điều phối giao tiếp giữa chúng.

- Colleague là các thành phần cần giao tiếp với nhau, và chúng gửi thông điệp thông qua Mediator.

- ConcreteColleague là lớp triển khai của Colleague, và nó gửi và nhận thông điệp thông qua Mediator.

- Java cung cấp tính năng Interface và abstract class, giúp dễ dàng triển khai các thành phần trong Mediator Pattern một cách linh hoạt.

*2.4.2 C# (.NET)*

Trong C#, ta cũng có thể sử dụng interface và lớp, tuy nhiên, C# cung cấp tính năng như event và delegate, có thể sử dụng để triển khai một cách linh hoạt hơn.

- Mediator interface có thể được triển khai thông qua các event và delegate, cho phép các - Colleague đăng ký các sự kiện và gửi thông điệp thông qua sự kiện đó.

- ConcreteMediator triển khai các sự kiện và delegate, quản lý các Colleague và điều phối thông điệp giữa chúng.

- Colleague và ConcreteColleague sử dụng sự kiện và delegate để gửi và nhận thông điệp.

*2.4.3 JavaScript*

Trong JavaScript, ta có thể sử dụng các đối tượng và hàm để triển khai Mediator Pattern, tạo một đối tượng Mediator và các Colleague sử dụng các đối tượng và hàm.

- Mediator sẽ duy trì danh sách các Colleague và quản lý việc giao tiếp giữa chúng.

- Các Colleague gửi thông điệp thông qua các hàm của Mediator.

*2.4.4 Python*

Trong Python, ta có thể sử dụng các class và hàm để triển khai Mediator Pattern. Ta có thể tạo các class cho Mediator và Colleague và sử dụng các hàm để giao tiếp giữa chúng.

Python cũng cung cấp các thư viện như asyncio hoặc threading để triển khai các tính năng đa luồng, giúp tạo ra một môi trường đa luồng cho việc triển khai Mediator Pattern trong các ứng dụng đa luồng.

*2.4.5 Angular*

Angular là một framework JavaScript phổ biến được sử dụng cho việc phát triển ứng dụng web đơn trang (SPA). Trong Angular, Mediator Pattern có thể được triển khai thông qua Service. Angular Service là các đối tượng singleton được sử dụng để chia sẻ dữ liệu và logic giữa các thành phần của ứng dụng. Có thể sử dụng một Service như một Mediator để điều phối giao tiếp giữa các thành phần khác nhau của ứng dụng.

*2.4.6 ASP.NET Core*

- ASP.NET Core là một framework phát triển ứng dụng web và dịch vụ web của Microsoft. Trong ASP.NET Core, ta có thể sử dụng Middleware để triển khai Mediator Pattern. Middleware trong ASP.NET Core là các thành phần mà yêu cầu HTTP sẽ đi qua khi chúng được xử lý bởi ứng dụng.

- Có thể tạo một Middleware để điều phối giao tiếp giữa các middleware khác nhau hoặc giữa các thành phần khác nhau của ứng dụng. Middleware này có thể xử lý các yêu cầu và phản hồi, và chuyển tiếp chúng đến các Middleware hoặc thành phần khác.

*2.4.7 Spring Framework*

- Spring Framework là một framework phát triển ứng dụng Java phổ biến, được sử dụng cho phát triển các ứng dụng web, ứng dụng di động và ứng dụng doanh nghiệp. Trong Spring, có thể sử dụng Dependency Injection (DI) và các bean để triển khai Mediator Pattern.

- Ta có thể tạo một bean để đại diện cho Mediator và tiêm nó vào các thành phần khác nhau của ứng dụng bằng cách sử dụng DI. Mediator này có thể điều phối giao tiếp giữa các thành phần và xử lý các yêu cầu của chúng.

2.5 Ưu & nhược điểm của Mediator Pattern

*2.5.1 Ưu điểm*

- Giảm việc gắn kết giữa các component khác nhau trong một chương trình.

- Tăng tính tái sử dụng

- Đảm bảo nguyên tắc Single Responsibility Principle (SRP): chúng ta có thể trích xuất sự liên lạc giữa các component khác nhau vào trong một nơi duy nhất, làm cho nó được bảo trì dễ dàng hơn.

- Đảm bảo nguyên tắc Open/Closed Principle (OCP): chúng ta có thể tạo ra các mediator mới mà không cần thay đổi các component.

- Đơn giản hóa giao tiếp giữa đối tượng, Một Mediator 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.

- Dễ dàng quản lý tập trung

*2.5.2 Nhược điểm*

- Tạo ra một điểm tập trung: Mediator có thể trở thành một điểm tập trung của toàn bộ tương tác trong hệ thống, làm tăng bớt hiệu suất nếu không được thiết kế và triển khai đúng cách.

- Việc hiểu và bảo trì: Việc hiểu và bảo trì có thể khó khăn hơn do nó có thể ẩn các tương tác phức tạp giữa các thành phần.

- Tăng chi phí khởi tạo: Thêm một lớp trung gian có thể làm tăng chi phí khởi tạo và phát triển hệ thống ban đầu.

2.6 Trường hợp sử dụng Mediator Pattern

*2.6.1 Khi nào nên sử dụng Mediator Pattern*

- Khi cần giảm sự phụ thuộc giữa các thành phần: Mediator Pattern giúp giảm sự phụ thuộc giữa các đối tượng trong hệ thống bằng cách tạo ra một lớp trung gian cho việc giao tiếp. Thay vì các đối tượng giao tiếp trực tiếp với nhau, chúng giao tiếp thông qua Mediator. Điều này làm cho các đối tượng trở nên độc lập hơn và dễ dàng mở rộng.

- Khi cần tăng tính linh hoạt và tái sử dụng: Mediator Pattern giúp tăng tính linh hoạt của hệ thống bằng cách tạo ra một cơ chế giao tiếp chung cho các thành phần. Thành phần mới có thể được thêm vào hệ thống mà không cần phải thay đổi các thành phần hiện có. Điều này giúp tăng tính tái sử dụng và giảm thời gian phát triển.

- Khi có nhiều thành phần cần giao tiếp với nhau: Khi có nhiều thành phần cần giao tiếp với nhau và mỗi thành phần cần giao tiếp với nhiều thành phần khác, việc sử dụng Mediator Pattern giúp tạo ra một cơ chế giao tiếp chung cho tất cả các thành phần. Điều này giúp tránh việc các thành phần phải giao tiếp trực tiếp với nhau, làm cho hệ thống trở nên dễ bảo trì và mở rộng.

- Khi cần giảm sự phức tạp của code: Mediator Pattern giúp giảm sự phức tạp của code bằng cách tách biệt logic giao tiếp ra khỏi logic kinh doanh của các thành phần. Điều này làm cho code trở nên dễ đọc, dễ hiểu và dễ bảo trì hơn.

*2.6.2 Khi nào không nên sử dụng Mediator Pattern*

- Khi chỉ có một số ít các thành phần cần giao tiếp với nhau: Nếu chỉ có một số ít các thành phần cần giao tiếp với nhau và không có nhiều sự phụ thuộc giữa chúng, việc sử dụng Mediator Pattern có thể làm tăng sự phức tạp của code mà không mang lại nhiều lợi ích.

- Khi có các giải pháp đơn giản hơn: Nếu có các giải pháp đơn giản hơn như truyền thông điệp trực tiếp giữa các thành phần hoặc sử dụng các pattern khác như Observer Pattern, có thể xem xét việc sử dụng các giải pháp đó thay vì Mediator Pattern.

- Khi không có sự phân cấp rõ ràng giữa các thành phần: Nếu không có một sự phân cấp rõ ràng giữa các thành phần và các thành phần cần giao tiếp với nhau không liên quan đến nhau, việc sử dụng Mediator Pattern có thể làm tăng sự phức tạp của code mà không mang lại lợi ích nhiều.

2.7 Ứng dụng của Mediator Pattern

*2.7.1 Trong Phát triển Ứng dụng Web*

- Mediator Pattern có thể được sử dụng trong phát triển ứng dụng web để quản lý giao tiếp giữa các thành phần của giao diện người dùng. Trong một ứng dụng web, có rất nhiều thành phần như form, danh sách, bảng, nút bấm và các thành phần tương tác khác. Mediator Pattern có thể được áp dụng để tạo ra một cơ chế giao tiếp chung giữa các thành phần này, giúp giảm sự phụ thuộc giữa chúng và làm cho code dễ bảo trì hơn.

- Ví dụ, trong một trang web có một form đăng nhập và một danh sách người dùng, Mediator Pattern có thể được sử dụng để quản lý việc xác thực người dùng sau khi họ đăng nhập. Form đăng nhập có thể gửi yêu cầu xác thực đến Mediator, và Mediator sẽ xác minh thông tin và cập nhật danh sách người dùng. Các thành phần khác trong trang web có thể đăng ký với Mediator để nhận thông báo khi có sự thay đổi trong danh sách người dùng.

*2.7.2 Trong hệ thống điều khiển và tự động hóa*

- Mediator Pattern cũng có thể được áp dụng trong các hệ thống điều khiển và tự động hóa để quản lý giao tiếp giữa các thiết bị và thành phần của hệ thống. Trong một hệ thống điều khiển nhà thông minh, ví dụ, có rất nhiều thiết bị như cảm biến, thiết bị điều khiển và các thành phần khác nhau. Mediator Pattern có thể được sử dụng để tạo ra một cơ chế giao tiếp chung giữa các thiết bị này, giúp giảm sự phụ thuộc giữa chúng và làm cho hệ thống dễ mở rộng hơn.

- Ví dụ, trong một hệ thống điều khiển nhà thông minh, Mediator Pattern có thể được sử dụng để điều phối việc bật/tắt đèn và điều chỉnh nhiệt độ phòng. Các thiết bị cảm biến có thể gửi dữ liệu đến Mediator, và Mediator sẽ quyết định khi nào cần phải bật/tắt đèn hoặc điều chỉnh nhiệt độ dựa trên dữ liệu đó.

*2.7.3 Trong ứng dụng mạng xã hội*

- Trong phát triển ứng dụng mạng xã hội, Mediator Pattern có thể được sử dụng để quản lý giao tiếp giữa các thành phần của hệ thống như người dùng, bài đăng, nhóm và trang cá nhân. Mediator có thể điều phối việc chia sẻ thông tin, tương tác giữa các thành phần và thông báo sự kiện mới đến các thành phần khác.

- Ví dụ, trong một ứng dụng mạng xã hội, Mediator Pattern có thể được sử dụng để quản lý giao tiếp giữa người dùng và bài đăng. Người dùng có thể gửi bình luận hoặc thích một bài đăng, và Mediator sẽ cập nhật thông tin và thông báo sự kiện mới đến tất cả người dùng khác trong mạng xã hội.

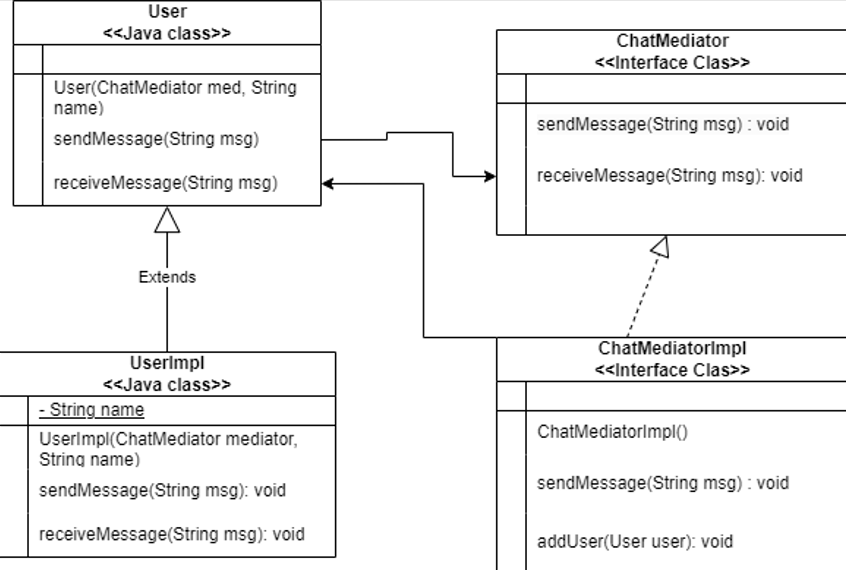
*2.7.4 Trong ứng dụng trò chơi*

- Mediator Pattern cũng có thể được sử dụng trong phát triển trò chơi để quản lý giao tiếp giữa các thành phần của trò chơi như nhân vật, vật phẩm, môi trường và hệ thống. Mediator có thể điều phối các sự kiện trong trò chơi, quản lý trạng thái của trò chơi và tương tác giữa các thành phần khác nhau.

- Ví dụ, trong một trò chơi RPG, Mediator Pattern có thể được sử dụng để quản lý giao tiếp giữa nhân vật và môi trường. Nhân vật có thể gửi yêu cầu di chuyển hoặc tấn công đến Mediator, và Mediator

2.8 Source code Mediator Pattern

*2.8.1 Giới thiệu về đề tài*



**18. Sơ đồ của đề tài mô hình Chat**

Đề tài: Ứng dụng chat

Vấn đề đặt ra:

- Trong quá trình phát triển một ứng dụng chat, chúng ta cần xử lý việc giao tiếp giữa các thành viên trong nhóm chat một cách hiệu quả. Cụ thể, chúng ta muốn triển khai một hệ thống cho phép các người dùng gửi và nhận tin nhắn với nhau thông qua một trung gian, đồng thời không cần phải biết về sự tồn tại của nhau.

Giải pháp:

- Trong trường hợp này, ta có thể sử dụng Pattern Mediator để giải quyết vấn đề trên. Dưới đây là cách chúng ta có thể triển khai Pattern Mediator trong ứng dụng chat:

+ ChatMediator (Trung gian): Đây là giao diện hoặc lớp trừu tượng định nghĩa các phương thức để các thành viên trong nhóm chat có thể giao tiếp với nhau thông qua nó. Trong ví dụ này, interface ChatMediator có thể đại diện cho thành phần này.

+ ConcreteMediator (Trung gian cụ thể): Là lớp cụ thể triển khai giao diện Mediator và quản lý các mối quan hệ giữa các thành viên trong nhóm chat. Nó giữ các tham chiếu đến tất cả các thành viên (Colleague) và hỗ trợ giao tiếp giữa chúng. Trong ví dụ này, lớp ChatMediatorImpl có thể đóng vai trò là ConcreteMediator.

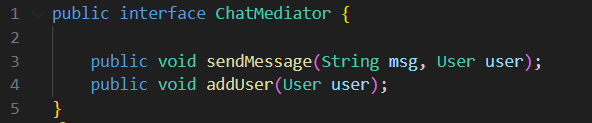
+ Colleague (Thành viên): Đây là giao diện hoặc lớp trừu tượng đại diện cho các thành viên trong nhóm chat cần giao tiếp với nhau thông qua Mediator. Trong ví dụ này, interface User có thể đại diện cho thành phần này.

+ ConcreteColleague (Thành viên cụ thể): Là lớp cụ thể triển khai giao diện Colleague và thực hiện giao tiếp thông qua Mediator. Trong ví dụ này, lớp UserImpl có thể đóng vai trò là ConcreteColleague.

Bằng cách triển khai Pattern Mediator, ta có thể xây dựng một hệ thống chat linh hoạt và dễ bảo trì, trong đó các thành viên có thể giao tiếp với nhau thông qua một trung gian, giúp giảm bớt sự phụ thuộc giữa các thành viên và tăng tính mô-đun hóa của hệ thống.

*2.8.2 Mô hình đề tài*

*2.8.2.1 ChatMediator (Đại diện cho Mediator)*

**

**19. Mediator của đề tài ứng dụng Chat**

ChatMediator (Trung gian trò chuyện) là giao diện (interface) định nghĩa các phương thức để giao tiếp giữa các thành viên của cuộc trò chuyện. Các phương thức bao gồm sendMessage để gửi tin nhắn và addUser để thêm một người dùng mới vào trò chuyện.

*2.8.2.2 ChatMediatorImpl (Đại diện cho ConcreteMediator)*

**

**20. ConcreteMediator của đề tài ứng dụng Chat**

ChatMediatorImpl (Triển khai Trung gian trò chuyện) là lớp cụ thể triển khai giao diện ChatMediator, lưu trữ danh sách các người dùng đang tham gia trò chuyện. Thực hiện phương thức sendMessage để gửi tin nhắn đến tất cả các người dùng trừ người gửi tin nhắn.

Thực hiện phương thức addUser để thêm một người dùng mới vào danh sách.

- Constructor ChatMediatorImpl():

+ Đây là constructor của lớp, được sử dụng để khởi tạo một đối tượng ChatMediatorImpl.

+ Trong constructor này, một danh sách mới users được tạo ra bằng cách khởi tạo một đối tượng ArrayList.

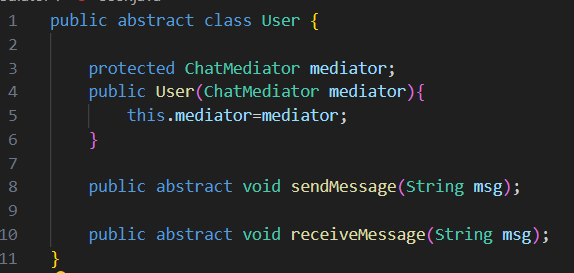
- Phương thức sendMessage(String msg, User user):

+ Sử dụng để gửi tin nhắn từ một người dùng cụ thể (được đại diện bởi đối tượng user) đến tất cả các thành viên khác trong nhóm chat thông qua trung gian.

+ Trong phương thức này, vòng lặp for-each duyệt qua danh sách các người dùng trong users. Mỗi người dùng sẽ nhận được tin nhắn trừ người gửi, đảm bảo rằng người gửi không nhận lại tin nhắn mà họ đã gửi.

- Phương thức addUser(User user): Phương thức này được sử dụng để thêm một người dùng mới vào nhóm chat. Khi một người dùng mới tham gia nhóm, họ sẽ được thêm vào danh sách users, cho phép trung gian gửi tin nhắn đến họ và nhận tin nhắn từ họ.

*2.8.2.3 User (Đại diện cho Colleague)*

**

**21. Colleague của đề tài ứng dụng Chat**

Trong đoạn mã trên, chúng ta có lớp User là một lớp trừu tượng đại diện cho các thành viên trong nhóm chat. Lớp User được thiết kế như một lớp trừu tượng để cung cấp một cơ chế chung cho việc gửi và nhận tin nhắn trong ứng dụng chat. Các lớp con của User sẽ triển khai phương thức sendMessage và receiveMessage để xử lý gửi và nhận tin nhắn cụ thể.

- Constructor User(ChatMediator mediator):

+ Đây là constructor của lớp User, được sử dụng để khởi tạo một đối tượng User với một trung gian cụ thể được truyền vào như là một đối số.

+ Constructor này gán giá trị của trung gian cho thuộc tính mediator.

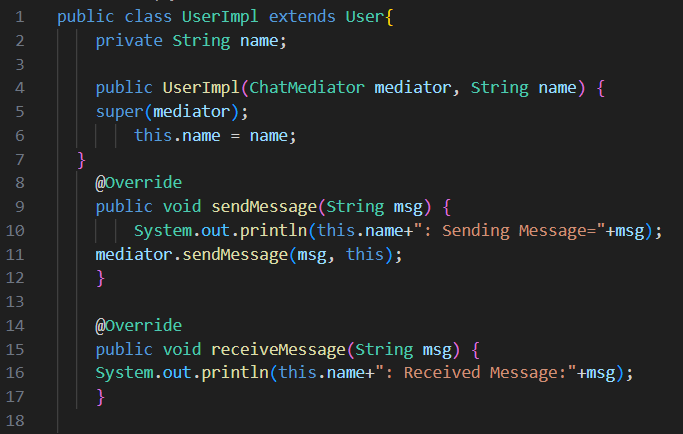
- Phương thức trừu tượng sendMessage(String msg):

+ Phương thức này được khai báo là một phương thức trừu tượng mà các lớp con cần triển khai. Nó đại diện cho việc gửi một tin nhắn từ người dùng hiện tại đến tất cả các thành viên khác trong nhóm chat thông qua trung gian.

- Phương thức trừu tượng receiveMessage(String msg):

+ Tương tự như phương thức sendMessage, phương thức này cũng là một phương thức trừu tượng mà các lớp con cần triển khai. Nó đại diện cho việc nhận một tin nhắn từ một người dùng khác trong nhóm chat thông qua trung gian.

*2.8.2.4 UserImpl (Đại diện cho ConcreteColleague)*

**

**22. ConcreteColleague của đề tài ứng dụng Chat**

Lớp UserImpl triển khai các phương thức sendMessage và receiveMessage để xử lý việc gửi và nhận tin nhắn trong ứng dụng chat. Khi một đối tượng UserImpl gửi tin nhắn, nó sẽ được chuyển tiếp qua trung gian để gửi đến tất cả các thành viên khác trong nhóm chat, và khi một tin nhắn được nhận, nó sẽ được hiển thị với tên của người dùng hiện tại.

- Constructor UserImpl(ChatMediator mediator, String name):

+ Đây là constructor của lớp UserImpl, được sử dụng để khởi tạo một đối tượng UserImpl với một trung gian cụ thể và một tên được truyền vào như là các đối số.

+ Constructor này truyền trung gian và tên cho constructor của lớp cha (User) thông qua super(mediator) và sau đó gán giá trị của tên cho thuộc tính name.

- Phương thức sendMessage(String msg):

+ Phương thức này được định nghĩa để gửi một tin nhắn từ người dùng hiện tại (đối tượng UserImpl) đến tất cả các thành viên khác trong nhóm chat thông qua trung gian.

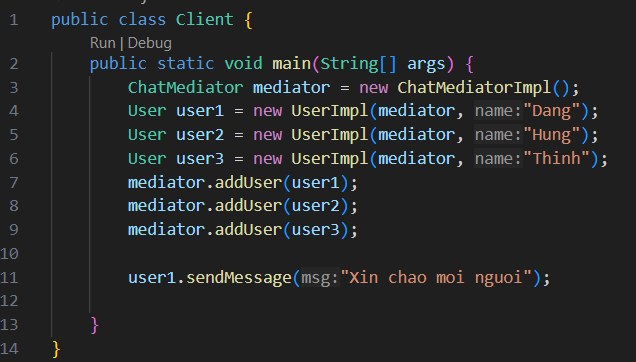
+ Trong phương thức này, tin nhắn được in ra màn hình với tên của người dùng hiện tại và sau đó phương thức sendMessage của trung gian được gọi để chuyển tiếp tin nhắn tới tất cả các thành viên khác.

- Phương thức receiveMessage(String msg):

+ Phương thức này được định nghĩa để nhận một tin nhắn từ một người dùng khác trong nhóm chat thông qua trung gian.

+ Trong phương thức này, tin nhắn được in ra màn hình với tên của người dùng hiện tại, cùng với nội dung của tin nhắn nhận được.

*2.8.2.5 Client*

**

**23. Lớp Client cho việc thử nghiệm sử dụng mô hình Mediator**

Trong đoạn mã trên, chúng ta có lớp Clien được sử dụng để thử nghiệm việc sử dụng mô hình Mediator. Dưới đây là mô tả của các phần trong lớp Client:

- Phương thức main(String[] args): Đây là phương thức chính được thực thi khi chương trình được chạy. Trong phương thức này, chúng ta thực hiện các bước sau:

+ Tạo một trung gian chat (mediator) mới thông qua lớp ChatMediatorImpl.

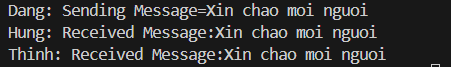
+ Tạo ra ba người dùng (users) mới thông qua lớp UserImpl. Mỗi người dùng được khởi tạo với một trung gian chat (mediator) cụ thể và một tên được chỉ định.

+ Thêm các người dùng đã tạo vào danh sách người dùng của trung gian chat thông qua phương thức addUser.

+ Gửi một tin nhắn từ người dùng đầu tiên (user1) thông qua phương thức sendMessage.

+ Khi chương trình chạy, nó tạo ra một môi trường chat đơn giản với ba người dùng: Dang, Hung và Thinh. Người dùng đầu tiên gửi một tin nhắn chào mừng ("Xin chào mọi người"), và tin nhắn này sẽ được truyền tới tất cả các người dùng khác trong nhóm chat thông qua trung gian.

*2.8.3 Kết quả đề tài*



- Đầu tiên User khởi tạo ra 3 người dùng “Dang”, “Hung”, “Thinh” với phương thức addUser cho ứng dụng Chat.

- User “Dang” đã sử phương thức sendMessage() của mô hình Mediator Pattern với nội dung: “Xin chao moi nguoi”.

- Hai User “Hung” và “Thinh” sử dụng phương thức receiveMessage để nhận tin nhắn từ User “Dang”.

TÀI LIỆU THAM KHẢO

[1] https://refactoring.guru/design-patterns/memento

[2] https://gpcoder.com/4763-huong-dan-java-design-pattern-memento/

[3] https://vi.wikipedia.org/wiki/Memento\_pattern

[4] topdev.vn/blog/huong-dan-java-design-pattern-memento/

[5] https://www.geeksforgeeks.org/memento-design-pattern/

[6] https://refactoring.guru/design-patterns/mediator

[7]<https://cafedev.vn/tu-hoc-design-pattern-gioi-thieu-ve-mediator-design-pattern-va-code-vi-du/>