

TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP.HCM
KHOA CÔNG NGHỆ THÔNG TIN



HCMUTE

BÁO CÁO CUỐI KÌ
TÌM HIỂU VỀ DESIGN PATTERN
TRONG JAVA

Giảng viên hướng dẫn: T.S Huỳnh Xuân Phụng

Lớp: 212DEPA330879_04CLC

Thành viên thực hiện:

Nguyễn Thanh Minh Đức 19110017

Lê Vũ Minh Hoànng 19119181

TP. Hồ Chí Minh, tháng 05 năm 2022

LỜI CẢM ƠN



Để hoàn thành tốt đề tài và bài báo cáo này, chúng em xin gửi lời cảm ơn chân thành đến giảng viên, tiến sĩ Huỳnh Xuân Phụng, người đã trực tiếp hỗ trợ chúng em trong suốt quá trình làm đề tài. Chúng em cảm ơn thầy đã đưa ra những lời khuyên từ kinh nghiệm thực tiễn của mình để định hướng cho chúng em đi đúng với yêu cầu của đề tài đã chọn, luôn giải đáp thắc mắc và đưa ra những góp ý, chỉnh sửa kịp thời giúp chúng em khắc phục nhược điểm và hoàn thành tốt cũng như đúng thời hạn đã đề ra.

Chúng em cũng xin gửi lời cảm ơn chân thành các quý thầy cô trong khoa Đào tạo Chất Lượng Cao nói chung và ngành Công Nghệ Thông Tin nói riêng đã tận tình truyền đạt những kiến thức cần thiết giúp chúng em có nền tảng để làm nên đề tài này, đã tạo điều kiện để chúng em có thể tìm hiểu và thực hiện tốt đề tài. Cùng với đó, chúng em xin được gửi cảm ơn đến các bạn cùng khóa đã cung cấp nhiều thông tin và kiến thức hữu ích giúp chúng em có thể hoàn thiện hơn đề tài của mình

Đề tài và bài báo cáo được chúng em thực hiện trong khoảng thời gian ngắn, với những kiến thức còn hạn chế cùng nhiều hạn chế khác về mặt kỹ thuật và kinh nghiệm trong việc thực hiện một dự án phần mềm. Do đó, trong quá trình làm nên đề tài có những thiếu sót là điều không thể tránh khỏi nên chúng em rất mong nhận được những ý kiến đóng góp quý báu của các quý thầy cô để kiến thức của chúng em được hoàn thiện hơn và chúng em có thể làm tốt hơn nữa trong những lần sau. Chúng em xin chân thành cảm ơn.

Cuối lời, chúng em kính chúc quý thầy luôn dồi dào sức khỏe và thành công hơn nữa trong sự nghiệp trồng người. Một lần nữa nhóm chúng em xin chân thành cảm ơn.

TP. Hồ Chí Minh, tháng 5 năm 2022

Sinh viên thực hiện

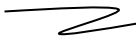

Nguyễn Thanh Minh Đức - 19110017

Lê Vũ Minh Hoàng - 19119181

ĐỒ ÁN MÔN HỌC MẪU THIẾT KẾ PHẦN MỀM

HỌC KÌ II, NĂM HỌC 2019 – 2023

- **Tên đề tài:** TÌM HIỂU VỀ DESIGN PATTERN TRONG JAVA
- **Thời gian thực hiện:** Từ: 26/03/2022 Đến: 5/6/2022
- **Giáo viên hướng dẫn:** TS. Huỳnh Xuân Phụng
- **Yêu cầu của đề tài:**
 1. SOLID Design principle in Java
 2. Pattern:
 - Builder
 - Adapter
 - Composite
 - Facade
 - Command
 - Memento
 - Visitor
 - Strategy
 3. Project minh họa
- **Lớp:** 212DEPA330879_04CLC
- **Sinh viên thực hiện:**

| STT | HỌ TÊN | MSSV | KÝ TÊN | THAM GIA (%) |
|-----|-----------------------|----------|--|--------------|
| 1 | Nguyễn Thanh Minh Đức | 19110017 | Đức  | 100% |
| 2 | Lê Vũ Minh Hoàng | 19119181 | Hoàng  | 100% |

Ghi chú:

- Tỷ lệ % = 100%
- Trưởng nhóm: Nguyễn Thanh Minh Đức

NHẬN XÉT

ĐIỂM SỐ

| TIÊU CHÍ | NỘI DUNG | TRÌNH BÀY | TỔNG |
|----------|----------|-----------|------|
| ĐIỂM | | | |

NHẬN XÉT CỦA GIÁO VIÊN

*BB*DD*

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

TP. Hồ Chí Minh, tháng 5 năm 2022

Giáo viên chấm điểm

Huỳnh Xuân Phụng

MỤC LỤC

| | |
|---|----|
| MỤC LỤC | 4 |
| MỞ ĐẦU | 7 |
| DANH MỤC HÌNH VẼ VÀ BẢNG BIỂU | 10 |
| DANH MỤC TỪ VIẾT TẮT | 12 |
| CHƯƠNG 1: KIẾN THỨC CƠ BẢN | 13 |
| 1. VẤN ĐỀ TRONG THIẾT KẾ PHẦN MỀM HƯỚNG ĐỐI TƯỢNG | 13 |
| 2. LỊCH SỬ HÌNH THÀNH DESIGN PATTERN | 13 |
| 3. KHÁI NIỆM | 14 |
| 4. ĐẶC ĐIỂM CHUNG | 15 |
| 5. ƯU NHƯỢC ĐIỂM | 16 |
| 5.1. Ưu điểm | 16 |
| 5.2. Nhược điểm | 16 |
| 6. PHÂN LOẠI DESIGN PATTERN | 17 |
| 6.1. Nhóm Creational | 18 |
| 6.2. Nhóm Structural | 19 |
| 6.3. Nhóm Behavioral | 20 |
| 7. TẠM KẾT | 22 |
| CHƯƠNG 2: CÁC KỸ THUẬT CỦA DESIGN PATTERN | 23 |
| 1. SOLID | 23 |
| 1.1. Khái niệm | 23 |
| 1.2. Năm nguyên tắc SOLID | 23 |
| <i>1.2.1. Nguyên tắc đơn nhiệm (Single Responsibility Principle)</i> | 23 |
| <i>1.2.2. Nguyên tắc đóng mở (The Open-Closed Principle)</i> | 25 |
| <i>1.2.3. Nguyên tắc phân vùng Liskov (The Liskov Substitution Principle)</i> | 27 |
| <i>1.2.4. Nguyên tắc phân tách giao diện (Interface Segregation Principle)</i> ... | 28 |
| <i>1.2.5. Nguyên tắc đảo ngược phụ thuộc (Dependency Inversion Principle)</i> 31 | |

| | |
|---|----|
| 1.3. Tạm kết | 33 |
| 2. CÁC MẪU THIẾT KẾ PHẦN MỀM | 33 |
| 2.1. Builder Pattern | 33 |
| 2.1.1. Khái niệm | 33 |
| 2.1.2. Cách cài đặt | 34 |
| 2.1.3. Ứng dụng | 35 |
| 2.1.4. Ưu/nhược điểm | 35 |
| 2.2. Adapter Pattern | 35 |
| 2.2.1. Khái niệm | 35 |
| 2.2.2. Cách cài đặt | 35 |
| 2.2.3. Ứng dụng | 37 |
| 2.2.4. Ưu/nhược điểm | 37 |
| 2.3. Composite Pattern | 38 |
| 2.3.1. Khái niệm | 38 |
| 2.3.2. Cách cài đặt | 38 |
| 2.3.3. Ứng dụng | 39 |
| 2.3.4. Ưu/nhược điểm | 39 |
| 2.4. Façade Pattern | 39 |
| 2.4.1. Khái niệm | 40 |
| 2.4.2. Cách cài đặt | 40 |
| 2.4.3. Ứng dụng | 40 |
| 2.4.4. Ưu/nhược điểm | 41 |
| 2.5. Command Pattern | 41 |
| 2.5.1. Khái niệm | 41 |
| 2.5.2. Cách cài đặt | 42 |
| 2.5.3. Ứng dụng | 43 |
| 2.5.4. Ưu/nhược điểm | 43 |

| | |
|---------------------------------------|------------------------------|
| 2.6. Memento Pattern | 43 |
| 2.6.1. Khái niệm | 43 |
| 2.6.2. Cách cài đặt | 43 |
| 2.6.3. Ứng dụng | 44 |
| 2.6.4. Ưu/nhược điểm | 44 |
| 2.7. Visitor Pattern | 45 |
| 2.7.1. Khái niệm | 45 |
| 2.7.2. Cách cài đặt | 46 |
| 2.7.3. Ứng dụng | 47 |
| 2.7.4. Ưu/nhược điểm | 47 |
| 2.8. Strategy Pattern | 47 |
| 2.8.1. Khái niệm | 47 |
| 2.8.2. Cách cài đặt | 48 |
| 2.8.3. Ứng dụng | 48 |
| 2.8.4. Ưu/nhược điểm | 48 |
| 2.9. Kết luận | 49 |
| CHƯƠNG 3: PROJECT MINH HỌA | 50 |
| 3.1. GIỚI THIỆU | 50 |
| 3.2. HỆ THỐNG QUẢN LÝ SẢN PHẨM | 50 |
| 3.3. CHƯƠNG TRÌNH THỰC NGHIỆM | 50 |
| 3.3.1. Giao diện chương trình | 50 |
| 3.3.2. Chức năng | 55 |
| 3.3.3. Cơ Sở Dữ Liệu | Error! Bookmark not defined. |
| CHƯƠNG 4: TỔNG KẾT | 59 |
| TÀI LIỆU THAM KHẢO | 60 |

MỞ ĐẦU

1. LỜI 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 “Design Pattern trong Java” Trong quá trình thực hiện đồ án, do còn hạn chế về thời gian, kinh nghiệm thực tế và dịch bệnh COVID, nhóm chúng em mong nhận được những góp ý chân thành từ thầy 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.

2. KẾ HOẠCH THỰC HIỆN

2.1. Kế hoạch thực hiện

| Tuần | Công Việc Thực Hiện | Ngày Bđ | Ngày Kt | KQ |
|------|---------------------|------------|------------|----|
| 1 | Tìm hiểu SOLID | 28/03/2022 | 03/04/2022 | ✓ |
| 2 | Tìm hiểu Builder | 04/04/2022 | 10/04/2022 | ✓ |
| 3 | Tìm hiểu Adapter | 11/04/2022 | 17/04/2022 | ✓ |
| 4 | Tìm hiểu Composite | 18/04/2022 | 20/04/2022 | ✓ |
| 5 | Tìm hiểu Façade | 21/04/2022 | 24/04/2022 | ✓ |

| | | | | |
|----|-----------------------------|------------|------------|---|
| 6 | Tìm hiểu Command | 25/04/2022 | 28/04/2022 | ✓ |
| 7 | Tìm hiểu Memento | 29/04/2022 | 01/05/2022 | ✓ |
| 8 | Tìm hiểu Visitor | 02/05/2022 | 08/05/2022 | ✓ |
| 9 | Tìm hiểu Strategy | 09/05/2022 | 15/05/2022 | ✓ |
| 10 | Xây dựng project minh họa | 16/05/2022 | 21/05/2022 | |
| 11 | Hoàn tất project và báo cáo | 22/5/2022 | 22/5/2022 | |

2.2. Phân công nhiệm vụ

| TT | Tên sinh viên | Mô tả công việc | Đóng góp |
|----|-----------------------|--|----------|
| 1 | Nguyễn Thanh Minh Đức | <ul style="list-style-type: none"> - Lập kế hoạch thực hiện, phân chia công việc các thành viên trong nhóm, và lên ý tưởng Project. - Phụ trách tìm hiểu Builder, Adapter, Composite, Command. - Phụ trách tổng hợp nội dung, code và làm báo cáo | 100% |
| 2 | Lê Vũ Minh Hoàng | <ul style="list-style-type: none"> - Phụ trách tìm hiểu SOLID, Façade, Memento, Visitor, Strategy. - Phụ trách xây dựng Project. - Phụ trách báo cáo phần thực hiện. | 100% |

3. BỐ CỤC BÀI BÁO CÁO

Đồ án được tổ chức làm 6 phần như sau:

- Mở đầu: Trình bày rõ lý do chọn đề tài, mục tiêu nghiên cứu đồ án kế hoạch thực hiện 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 Pattern*. Chương này trình bày chi tiết về các kỹ thuật cũng như cách xây dựng mẫu trong thiết kế phần mềm.
- Chương 3: *Project minh họa*. Chương này trình bày chủ yếu về phân tích thiết kế hệ thống hướng đối tượng và áp dụng Design Pattern vào bài toán.
- Tổng kết: 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.
- Tài liệu tham khảo: Phần này đưa ra những đường dẫn website và sách tham khảo mà nhóm đã xem và áp dụng vào bài báo cáo.

DANH MỤC HÌNH VẼ VÀ BẢNG BIỂU

| | |
|--|----|
| Hình 1 Quy tắc thiết kế hướng đối tượng | 17 |
| Hình 2 Mối quan hệ giữa 23 Design Patterns | 18 |
| Hình 3 Các mẫu Design Pattern trong nhóm Creational | 18 |
| Hình 4 Các mẫu Design Pattern trong nhóm Structural | 19 |
| Hình 5 Các mẫu Design Pattern trong nhóm Behavioral | 20 |
| Hình 6 Minh họa nguyên tắc đơn nhiệm (SRP)..... | 23 |
| Hình 7 Class vi phạm nguyên tắc SRP | 24 |
| Hình 8 Áp dụng SRP vào thuật toán..... | 25 |
| Hình 9 Minh họa nguyên tắc đóng mở (OCP)..... | 25 |
| Hình 10 Class vi phạm nguyên tắc OCP..... | 26 |
| Hình 11 Áp dụng OCP vào thuật toán | 27 |
| Hình 12 Minh họa nguyên tắc phân vùng (LSP) | 27 |
| Hình 13 Minh họa LSP trong thuật toán..... | 28 |
| Hình 14 Minh họa nguyên tắc phân tách giao diện (ISP)..... | 29 |
| Hình 15 Minh họa ISP trong thuật toán..... | 30 |
| Hình 16 Áp dụng ISP vào thuật toán | 30 |
| Hình 17 Minh họa nguyên tắc đảo ngược phụ thuộc (DIP)..... | 31 |
| Hình 18 Ví dụ thuật toán chưa áp dụng DIP..... | 32 |
| Hình 19 Áp dụng DIP vào thuật toán | 33 |
| Hình 20 Sơ đồ UML mô tả Builder Pattern..... | 34 |
| Hình 21 Sơ đồ UML mô tả Object Pattern | 36 |
| Hình 22 Sơ đồ UML mô tả Class Pattern | 37 |
| Hình 23 Sơ đồ UML mô tả Composite Pattern..... | 39 |
| Hình 24 Sơ đồ UML mô tả Facade Pattern..... | 40 |
| Hình 25 Sơ đồ UML mô tả Command Pattern | 42 |
| Hình 26 Sơ đồ UML mô tả Memento Pattern..... | 44 |
| Hình 27 Sơ đồ UML mô tả Visitor Pattern..... | 46 |
| Hình 28 Sơ đồ UML mô tả Strategy Pattern..... | 48 |
| Hình 29 Login vào hệ thống | 50 |
| Hình 30 Đăng ký tài khoản | 51 |
| Hình 31 Giao diện chính quản lý sản phẩm..... | 51 |

| | |
|--|----|
| Hình 32 Giao diện quản lý sản phẩm..... | 52 |
| Hình 33 Giao diện quản lý người dùng | 52 |
| Hình 34 Giao diện quản lý loại sản phẩm..... | 53 |
| Hình 35 Giao diện quản lý quyền người dùng..... | 53 |
| Hình 36 Giao diện quản lý nhân viên | 54 |
| Hình 37 Đăng xuất phần mềm | 54 |
| Hình 38 Lấy dữ liệu sản phẩm từ CSDL | 55 |
| Hình 39 Chức năng thêm sản phẩm vào hệ thống | 55 |
| Hình 40 Chức năng sửa sản phẩm trong hệ thống | 56 |
| Hình 41 Chức năng xóa sản phẩm khỏi hệ thống | 56 |
| Hình 42 Áp dụng Builder Pattern vào quy trình kết nối dữ liệu..... | 56 |
| Hình 43 Chức năng đăng nhập vào hệ thống..... | 57 |
| Hình 44 Chức năng kiểm tra quyền | 57 |
| Hình 45 Biểu đồ Class Diagram | 58 |

DANH MỤC TỪ VIẾT TẮT

| TT | 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 | SRP | Single Responsibility Principle | Nguyên tắc đơn nhiệm |
| 7 | ORP | Open-Closed Principle | Nguyên lý đóng mở |
| 8 | LSP | Liskov Substitution Principle | Nguyên lý thay thế |
| 9 | ISP | Interface Segregation Principle | Nguyên lý phân tách |
| 10 | DIP | Dependency Inversion Principle | Nguyên tắc đảo ngược phụ thuộc |

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. 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.

2. LỊCH SỬ HÌNH THÀNH 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 Vlissides, 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]

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 bao 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.*

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, các 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 mối 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.*
- **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 rõ 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.

5. ƯU NHƯỢC ĐIỂM

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.

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.

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 Patterns chủ yếu được dựa theo những quy tắc sau đây về thiết kế hướng đối tượng.

| By Purpose | | | | |
|------------|--------|---|--|---|
| | | Creational | Structural | Behavioral |
| By Scope | Class | <ul style="list-style-type: none"> Factory Method | <ul style="list-style-type: none"> Adapter (class) | <ul style="list-style-type: none"> Interpreter Template Method |
| | Object | <ul style="list-style-type: none"> Abstract Factory Builder Prototype Singleton | <ul style="list-style-type: none"> Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy | <ul style="list-style-type: none"> Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor |

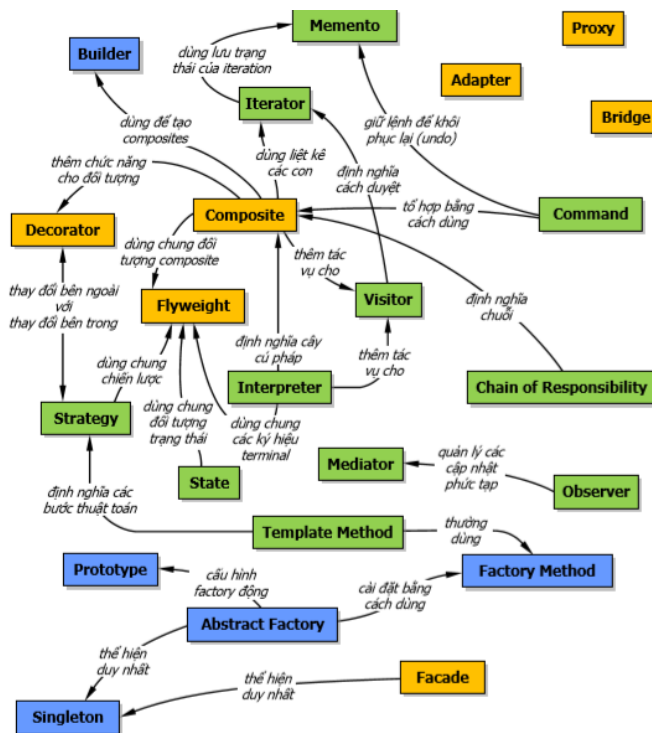
Hình 1 Quy tắc thiết kế hướng đối tượng

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

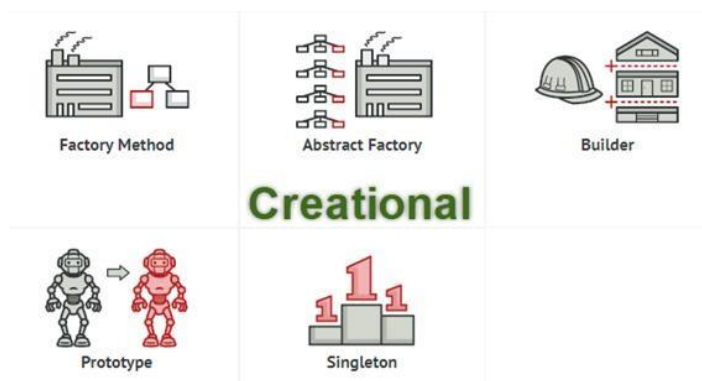
method new. Điều này giúp cho chương trình trở nên mềm dẻo hơn trong việc quyết định object nào cần được tạo ra trong những tình huống được đưa ra. [3]

- **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 2 Mối quan hệ giữa 23 Design Patterns

6.1. Nhóm Creational



Hình 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.

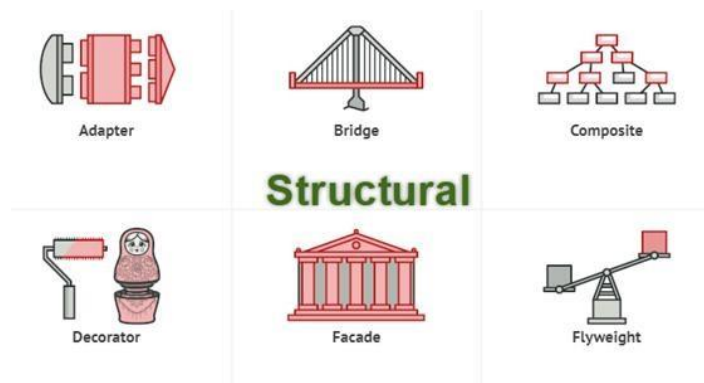
Builder:

- 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.

6.2. Nhóm Structural



Hình 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 \Rightarrow khả năng tổng quát hoá trong code của client \Rightarrow 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.

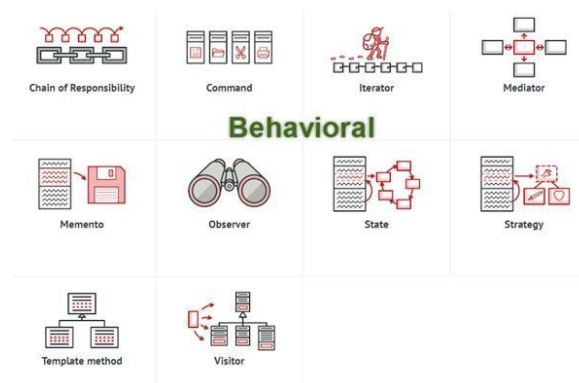
Flyweight:

- 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 được gọi là proxy.

6.3. Nhóm Behavioral



Hình 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 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.

7. TẠM KẾT

Chương này trình bày những khái niệm cơ bản cũng như ưu nhược điểm của Design Pattern. Nó 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

1. SOLID

1.1. Khái niệm

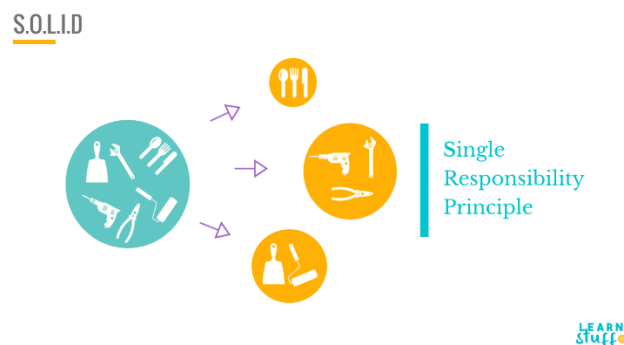
Trong OOP có 4 tính chất đặc biệt: trừu tượng, đóng gói, đa hình và kế thừa. Hầu hết lập trình viên đều đã biết các tính chất này của OOP, nhưng cách thức để phối hợp các tính chất này với nhau để tăng hiệu quả của ứng dụng thì không phải ai cũng nắm được. Một trong những chỉ dẫn để giúp chúng ta sử dụng được OOP hiệu quả hơn đó là nguyên tắc SOLID. Về cơ bản, SOLID là một bộ 5 chỉ dẫn đã được nhắc tới từ lâu bởi các nhà phát triển phần mềm giúp cho developer viết ra những đoạn code dễ đọc, dễ hiểu, dễ maintain, được đưa ra bởi Bob Martin và Michael Feathers

SOLID là viết tắt của 5 chữ cái đầu trong 5 nguyên tắc

- Single responsibility principle (SRP)
- Open/Closed principle (OCP)
- Liskov substitution principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)

1.2. Năm nguyên tắc SOLID

1.2.1. Nguyên tắc đơn nhiệm (Single Responsibility Principle)



Hình 6 Minh họa nguyên tắc đơn nhiệm (SRP)

Một class chỉ nên thực hiện một công việc. Nói cho dễ hiểu thì một class chỉ nên thực hiện một công việc, thay vì thực hiện nhiều việc trong một class thì chúng ta có thể cho mỗi class thực hiện một công việc.

Trong cuộc sống cũng vậy, nếu ta làm 2 hoặc nhiều công việc một lúc sẽ không thể có hiệu suất cao được. Như ví dụ phía dưới trong Bloggeer phải thực hiện 2 công việc là đọc và viết nhưng sau này chúng ta cần thêm nhiều công

việc hơn nữa thì sao? Mỗi lần muốn thay đổi ta lại phải vào Blogger để sửa chỉ cần một nhầm lẫn ta sẽ có hậu quả nghiêm trọng. Và để đảm bảo an toàn ta tách 2 công việc này 2 class khác nhau.

Với việc chia nhỏ ra ta thấy ta có thể dễ dàng gọi đến lớp tương ứng với từng công việc, nó cũng dễ hơn khi maintain code và không phải sửa ở lớp chính quá nhiều, các đối tượng đã được tách biệt hoàn toàn về nhiệm vụ.

Nguyên lý chung: Mỗi lớp chỉ nên đảm nhiệm một nhiệm vụ duy nhất

Lý do:

- Dễ quản lí mã nguồn
- Các lớp tập trung vào nhiệm vụ của mình
- Giảm tính phụ thuộc giữa các thành phần
- Có thể phát triển đồng thời các lớp độc lập với nhau
- Dễ dàng mở rộng
- Dễ dàng bảo trì

Demo:

```
public class DBHelper {

    public Connection openConnection() {};

    public void saveUser(User user) {};

    public List<Product> getProducts() {};

    public void closeConnection() {};

}
```

Hình 7 Class vi phạm nguyên tắc SRP

Bức ảnh trên là một ví dụ về việc vi phạm lỗi SRP vì class DBHelper ôm đồm tất cả các xử lý liên quan đến CSDL

- Trách nhiệm thứ nhất dùng để mở kết nối với CSDL
- Trách nhiệm thứ hai dùng để lưu trữ User
- Trách nhiệm thứ ba dùng để lấy sản phẩm từ User
- Trách nhiệm thứ tư dùng để đóng kết nối với CSDL

Theo đúng nguyên tắc này, ta phải tách class này ra làm nhiều class riêng, mỗi class chỉ làm một nhiệm vụ duy nhất. Tuy số lượng class nhiều

hơn những việc sửa chữa sẽ đơn giản hơn, dễ dàng tái sử dụng hơn, class ngắn hơn nên cũng ít bug hơn.

Vậy chúng ta nên tách thành class trên thành các class xử lý công việc riêng kiểu như sau:

```
public class DBConnection {

    public Connection openConnection() {};

    public void closeConnection() {};

}

public class UserRepository {

    public void saveUser(User user) {};

}

public class ProductRepository {

    public List<Product> getProducts() {};

}
```

Hình 8 Áp dụng SRP vào thuật toán

Với cách làm này, khi yêu cầu ứng dụng kết nối với CSDL sẽ không còn thực hiện nhiều trách nhiệm khác nhau: lấy dữ liệu từ Database, Validate, thông báo, ghi log, xử lý dữ liệu... Khi chỉ cần ta thay đổi cách lấy dữ liệu DB, thay đổi cách validate... ta sẽ chỉ phải sửa đổi một class. Rất dễ dàng khi bảo trì, nâng cấp, sửa lỗi, thử nghiệm...

1.2.2. Nguyên tắc đóng mở (The Open-Closed Principle)



Hình 9 Minh họa nguyên tắc đóng mở (OCP)

Hãy thử hình dung bạn có một chiếc máy ảnh chuyên dụng, bạn đã có một ống kính kèm theo máy để chụp phong cảnh hoa lá nhưng bây giờ bạn muốn biến máy ảnh này thành “chuyên chụp chân dung” và chúng ta sẽ mở rộng bộ máy ảnh ra bằng cách dùng các ống kính khác nhau, chứ chúng ta không nên đem ống kính đang dùng đi sửa lại, tương tự máy ảnh của bạn có đèn flash nhưng công suất yếu, bạn muốn đèn sáng hơn thì hãy ráp thêm một cái flash rời, không nên mở cái flash bên trong máy ra để thay bóng đèn công suất cao hơn, điều đó có thể làm được nhưng nó sẽ khá là ngu ngốc.

Nguyên lý chung:

- **Hạn chế sửa đổi:** Ta không nên chỉnh sửa source code của một module hoặc class có sẵn, vì sẽ ảnh hưởng tới tính đúng đắn của chương trình
- **Ưu tiên mở rộng:** Khi cần thêm tính năng mới, ta nên kế thừa và mở rộng các module/class có sẵn thành các module con lớn hơn. Các module/class con vừa có các đặc tính của lớp cha (đã được kiểm chứng đúng đắn), vừa được bổ sung tính năng mới phù hợp với yêu cầu.

Demo:

```
class Blogger {
    private string name;
    private int post;
    int paysalary(){
        if(this->post == 3)
            return salary;
        if(this->post == 4)
            return salary * 1.2;
        if(this->post == 5)
            return salary * 1.5;
    }
}
```

Hình 10 Class vi phạm nguyên tắc OCP

Theo cách làm trên hoàn toàn đúng. Tuy nhiên, nếu bạn thiết kế chương trình như thế này thì thực sự có nhiều điểm không hợp lý lắm, nếu chúng ta lại có thêm 1 kiểu nhuận bút nữa thì sao, khi đó chúng ta lại phải vào sửa

lại hàm để đáp ứng được nhu cầu mới hay sao? Code mới lúc đó sẽ ảnh hưởng tới code cũ, như vậy có khả năng là sẽ làm hỏng luôn code cũ...

Rõ ràng là chúng ta nên có một phương pháp an toàn và thân thiện hơn.

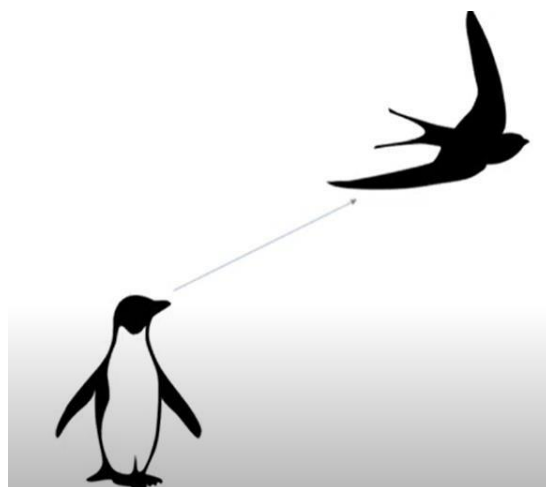
```
class Blogger {
    private string name;
    private int post;
    int paysalary(){
        return salary;
    }
}

class Postedfourposts extends Blogger{
    int paysalary(){
        return salary * 1.2;
    }
}
```

Hình 11 Áp dụng OCP vào thuật toán

Có thể thấy rằng, cách thiết kế này làm cho lớp Blogger trở nên: **ĐÓNG** với mọi sự thay đổi bên trong, nhưng luôn **MỞ** cho sự kế thừa để mở rộng sau này. Trong tương lai, khi nhu cầu mở rộng chương trình xuất hiện, có thêm nhiều đối tượng nữa cần xử lý thì chúng ta chỉ cần thêm lớp mới là sẽ giải quyết được vấn đề, trong khi vẫn đảm bảo được chương trình có sẵn không bị ảnh hưởng, nhờ đó mà hạn chế được phạm vi test, giúp giảm chi phí phát triển. Đó cũng là một trong những lợi ích ở khía cạnh dễ bảo trì sản phẩm

1.2.3. Nguyên tắc phân vùng Liskov (The Liskov Substition Principle)



Hình 12 Minh họa nguyên tắc phân vùng (LSP)

Bức ảnh trên cho chúng ta thấy rằng tuy cả hai đối tượng cùng là chim nhưng một loại có thể bay còn một loại không thể bay, điều này nó cũng giống như nguyên tắc Liskov.

Nguyên lý chung: Các lớp con có thể được sử dụng thay thế cho các lớp cha, nghĩa là nếu S là một lớp con của T, thì các đối tượng của lớp T có thể được thay thế bằng các đối tượng của lớp S mà không làm ảnh hưởng tới bất cứ hành vi nào của hệ thống

Lí do: Tránh các sai sót khi mở rộng thiết kế.

Demo: Chương trình để mô tả các loài chim bay được nhưng chim cánh cụt không bay được. Vì vậy khi viết đến hàm chim cánh cụt thì khi gọi hàm bay của chim cánh cụt, ta sẽ quăng NoFlyException

```
public class Bird {
    public virtual void Fly() { Console.WriteLine("Fly"); }
}
public class Chim_ung : Bird {
    public override void Fly() { Console.WriteLine("Eagle Fly"); }
}

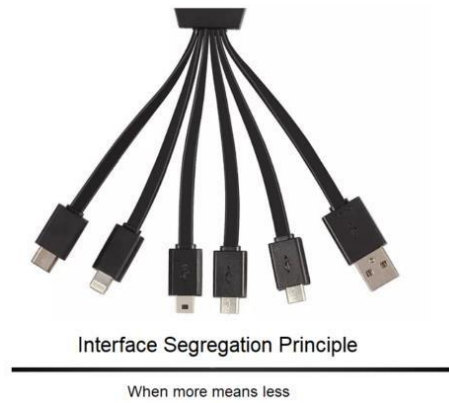
public class Chim_canh_cut : Bird {
    public override void Fly() { throw new NoFlyException(); }
}
```

Hình 13 Minh họa LSP trong thuật toán

Tuy nhiên, quay lại vòng lặp for ở hàm main, nếu như trong danh sách các con chim đó mà có một con chim cánh cụt thì sao? Chương trình mình sẽ quăng **Exception** vì chương trình của chúng ta đã vi phạm **LSP**.

Để có thể giải quyết vấn đề này chúng ta sẽ tách class chim cánh cụt ra một interface riêng.

1.2.4. Nguyên tắc phân tách giao diện (Interface Segregation Principle)



Hình 14 Minh họa nguyên tắc phân tách giao diện (ISP)

Bức ảnh cho chúng ta thấy nếu người dùng mua dây sạc này về chỉ để sạc iphone thì dường như họ đã bỏ ngổ 5 cáp sạc khác, điều này là vô cùng lãng phí, nó cũng giống với việc bạn tạo ra một interface với 5 method mà class thực thi interface chỉ dùng 1 method.

Chúng ta sẽ cùng đến nguyên tắc thứ 4 Interface segregation Principle: Nguyên tắc phân tách Interface.

Nguyên lý chung: Không nên bắt buộc phải triển khai một interface nếu không cần đến nó, cũng không nên bắt buộc phải phụ thuộc vào các phương thức mà không cần đến chúng

Nghĩa là: Các Interface chỉ nên chứa các phương thức cần thiết vừa đủ với mục đích của nó. Nên tách các Interface thành nhiều Interface nếu các phương thức của chúng không liên quan chặt chẽ đến nhau.

Demo: Khi học đại học những năm đầu có thể sẽ học chung một số môn. Nhưng sau một thời gian các môn học của mỗi người sẽ khác nhau và tùy thuộc vào ngành mỗi sinh viên, nhưng tất cả đều học, do vậy dùng interface là hợp lí. Lúc thiết kế, chúng ta sẽ nghĩ tới việc thiết kế một interface là Study có các function học, do chỉ có duy nhất một sự khác biệt nên bạn nhét hết tất cả môn học vào cùng một interface này luôn

```

interface Study
{
    function studyEnglish();
    function studyProgrammingLanguage();
}
class NormalStudent implements Study
{
    function studyEnglish(){
    }
    function studyProgrammingLanguage(){
        return NULL;
    }
}
class InformationTechnologyStudents implements Study
{
    function studyEnglish(){
    }
    function studyProgrammingLanguage(){
    }
}

```

Hình 15 Minh họa ISP trong thuật toán

Tuy thiết kế chương trình như vậy vẫn sẽ chạy bình thường nhưng khi ta muốn thêm một số môn học mới. Chúng ta nhận ra rằng, bởi vì sự khác biệt ngày càng nhiều, rất có thể sau này sẽ phát sinh nhiều môn học riêng cho từng hệ sinh viên nữa, do đó chúng ta nên có cách giải quyết triệt để hơn, bởi vì nếu cứ viết chung vào interface Study thì sẽ phát sinh **việc phải implement nhiều hàm không cần thiết**. Do vậy, chúng ta quyết định tách việc học thành các interface cụ thể khác nhau: phần học chung, phần học riêng.

```

interface Study
{
    function studyEnglish();
    function studyMath();
}

interface InformationInfTechnologyStudy
{
    function studyProgrammingLanguage();
}

interface EconomicsStudy
{
    function studyPhilosophy();
}

```

Hình 16 Áp dụng ISP vào thuật toán

Với thiết kế này, chúng ta không còn phải lo lắng tới việc phải implement những hàm không cần thiết, cũng sẽ dễ dàng kiểm soát được việc mở rộng hơn. Trong tương lai, nếu phát sinh thêm nhiều môn học riêng việc triển khai cũng dễ dàng và tường minh hơn rất nhiều

1.2.5. Nguyên tắc đảo ngược phụ thuộc (*Dependency Inversion Principle*)



Hình 17 Minh họa nguyên tắc đảo ngược phụ thuộc (DIP)

Chúng ta thấy 2 loại bóng đèn đó là bóng đèn tròn và bóng đèn dạng xoắn 2 bóng đèn lại phụ thuộc vào đuôi đèn tròn, khi bóng đèn tròn bị hư chúng ta chỉ cần thay bóng đèn tròn bằng bóng đèn dạng xoắn mà không hề ảnh hưởng đến kết nối điện hay hệ thống của bóng đèn vì chúng đã có một lớp trung gian kết nối 2 thứ đó là đuôi đèn tròn.

Để giúp việc bảo trì các hệ thống một tốt hơn chúng ta cùng đi đến nguyên tắc số 5 nguyên tắc Đảo ngược Phụ thuộc.

Nguyên lý chung:

- Các module cấp cao không nên phụ thuộc vào các module cấp thấp. Cả 2 nên phụ thuộc vào abstraction.
- Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. (Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation).

Ưu nhược điểm:

| Ưu điểm | Nhược điểm |
|--|---|
| <ul style="list-style-type: none"> ▪ Giảm sự kết dính giữa các module ▪ Code dễ bảo trì, dễ thay thế module ▪ Rất dễ test và viết Unit Test | <ul style="list-style-type: none"> ▪ Khái niệm DI khá “khó hiểu”, các developer mới sẽ gặp khó khăn khi học ▪ Sự dụng Interface nên đôi khi sẽ khó de-bug, do không biết chính xác module nào được gọi ▪ Làm tăng độ phức tạp của code |

Demo: Khi ta thiết kế một chương trình gửi thông báo từ email đến User.

```

public class EmailSender
{
    public void SendEmail()
    {
        //Send email
    }
}

public class Notification
{
    private EmailSender _email;
    public Notification()
    {
        _email = new EmailSender();
    }

    public void Send()
    {
        _email.SendEmail();
    }
}
    
```

Hình 18 Ví dụ thuật toán chưa áp dụng DIP

Nhưng bây giờ khách hàng muốn gửi cả SMS và Email đến cho user, bạn phải làm sao?

Tạo một lớp SMSSender và chỉnh sửa class Notificaiton. Như vậy bạn vi phạm một lúc hai nguyên tắc, Dependency Inversion về sự đảo ngược phụ thuộc và Open-Close Principle. Software đóng cho việc thay đổi nhưng mở cho việc mở rộng.

Để thỏa mãn hai nguyên tắc trên, bạn phải Refactoring code theo chiều hướng giảm sự phụ thuộc cứng bằng cách tạo ra một interface ISender dùng chung giữa hai class EmailSender và SMSSender.

```

public class EmailSender
{
    public void SendEmail()
    {
        //Send email
    }
}

public class Notification
{
    private EmailSender _email;
    public Notification()
    {
        _email = new EmailSender();
    }

    public void Send()
    {
        _email.SendEmail();
    }
}

```

Hình 19 Áp dụng DIP vào thuật toán

Giờ đây class Notification phụ thuộc mềm vào interface ISender, nếu khách hàng yêu cầu thêm một phương thức để chuyển tin nhắn ta có thể thêm vào dễ dàng bằng cách sử dụng interface ISender.

1.3. Tạm kết

Trên đây nhóm chúng em đã trình bày 5 nguyên tắc trong OOP. Áp dụng những nguyên tắc này sẽ đưa chúng ta tới một cảnh giới mới của việc thiết kế phần mềm, khi tuân theo những nguyên tắc này chương trình của chúng ta bây giờ đã linh động, rành mạch hơn, dễ bảo trì hơn, dễ mở rộng hơn, có tính kế thừa cao... Tất nhiên vẫn còn một số nguyên tắc khác nhưng SOLID là 5 nguyên tắc quan trọng nhất.

2. CÁC MẪU THIẾT KẾ PHẦN MỀM

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. Builder Pattern

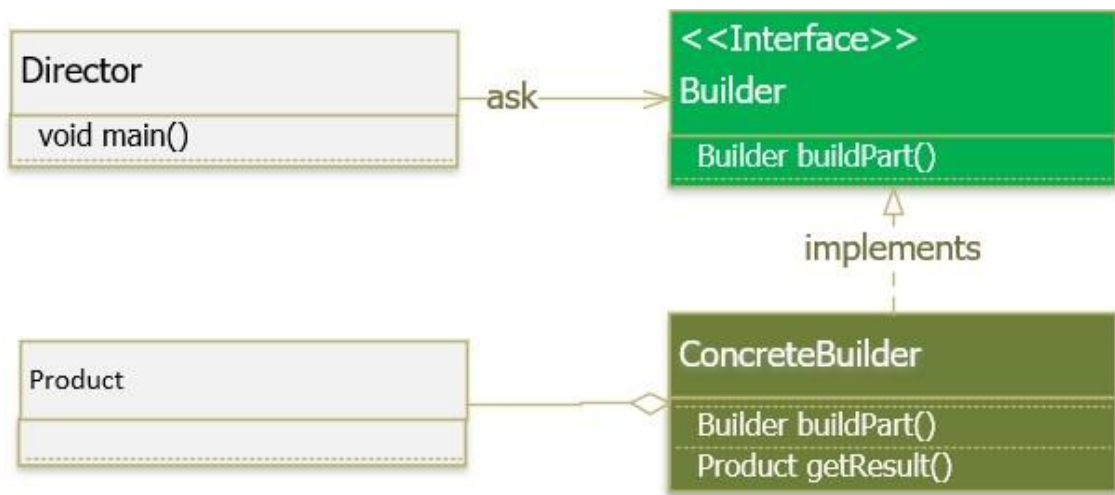
2.1.1. Khái niệm

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.*

2.1.2. Cách cài đặt



Hình 20 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 thể hiện mà nó đã tạo ra trước đó .
- **Director / Client:** Là nơi sẽ gọi tới Builder để tạo ra đối tượng.

2.1.3. Ứng dụng

- Tạo một đối tượng phức tạp: có nhiều thuộc tính (nhiều hơn 4) và một số bắt buộc (required), 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.

2.1.4. Ưu/nhược điểm

2.1.4.1. Ưu điểm

Hỗ trợ, loại bớt việc phải viết nhiều constructor. Code dễ đọc, dễ bảo trì hơn khi số lượng thuộc tính (property) bắt buộc để tạo một object từ 4 hoặc 5 property. 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.

2.1.4.2. Nhược điểm

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.

2.2. Adapter Pattern

2.2.1. Khái niệm

Adapter Pattern cho phép các interface (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.

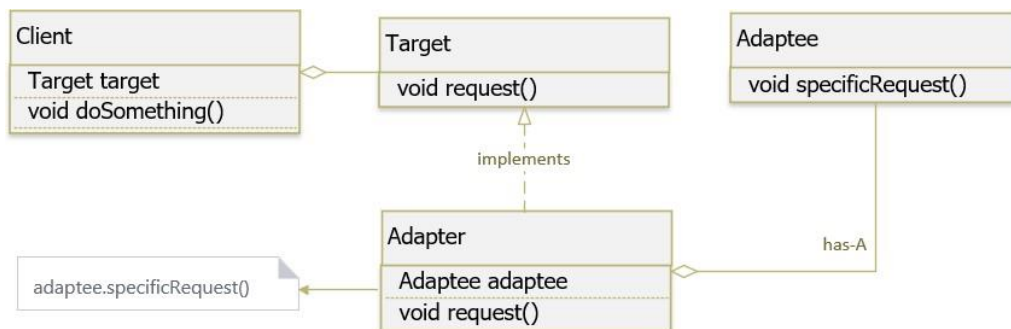
2.2.2. Cách cài đặt

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.

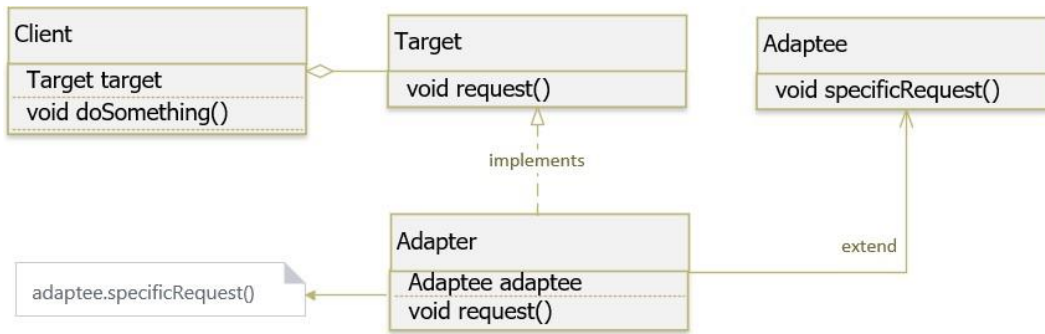
Có hai cách để thực hiện Adapter Pattern dựa theo cách cài đặt (implement):

- **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 21 Sơ đồ UML mô tả 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 22 Sơ đồ UML mô tả 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.

2.2.3. Ứng dụng

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.

2.2.4. Ưu/nhược điểm

2.2.4.1. Ưu điểm

- 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 có một số khuyết điểm nhỏ sau:
 - Tất cả các yêu cầu được chuyển tiếp, do đó sẽ 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.

2.2.4.2. Nhược điểm

- Tất cả yêu cầu phải được chuyển tiếp thông qua adapter, làm tăng thêm một ít chi phí
- Độ phức tạp của code nhìn chung tăng lên vì phải thêm interface và lớp.
- Vì không phải lúc nào ta cũng có thể thích nghi các method của các interface khác nhau với nhau, nên exception có thể xảy ra. Vấn đề này có thể tránh được nếu client cẩn thận hơn và adapter có tài liệu hướng dẫn rõ ràng.

2.3. Composite Pattern

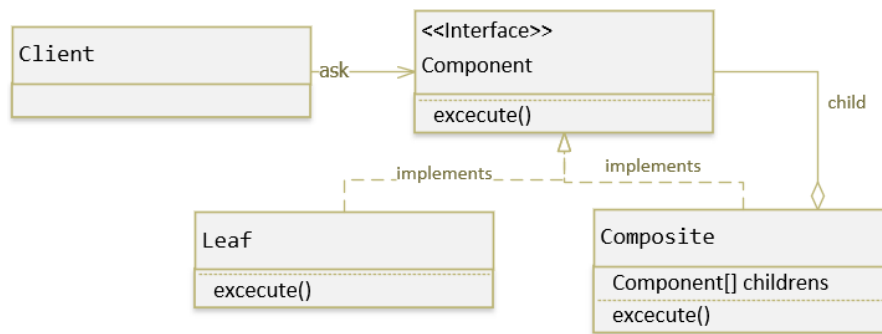
2.3.1. Khái niệm

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

2.3.2. Cách cài đặt

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 23 Sơ đồ UML mô tả Composite Pattern

2.3.3. Ứng dụng

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.

2.3.4. Ưu/nhược điểm

2.3.4.1. Ưu điểm

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

2.3.4.2. Nhược điểm

Có thể khó cung cấp một giao diện chung cho các lớp có chức năng khác nhau quá nhiều. Trong một số trường hợp nhất định, cần phải tổng quát hóa quá mức giao diện thành phần, khiến nó khó hiểu hơn

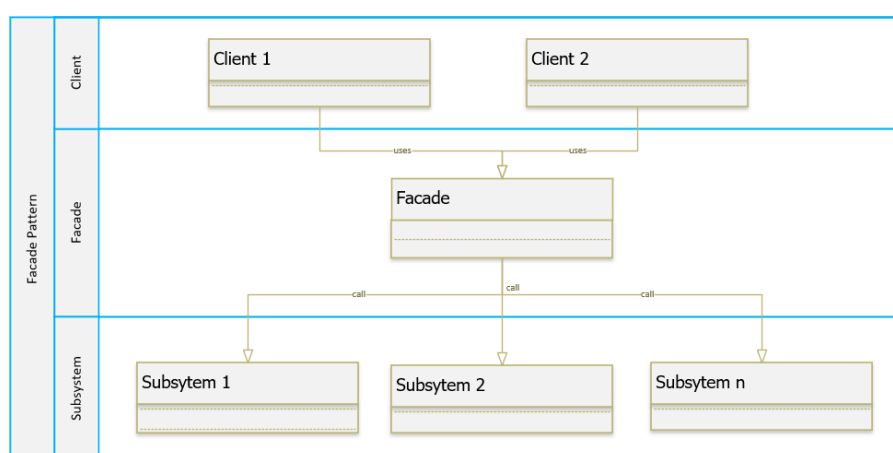
2.4. Façade Pattern

2.4.1. Khái niệm

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.

2.4.2. Cách cài đặt



Hình 24 Sơ đồ UML mô tả Facade 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.

2.4.3. Ứng dụng

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.

2.4.4. Ưu/nhược điểm

2.4.4.1. Ưu điểm

- Giúp cho hệ thống của bạn trở nên đơn giản hơn trong việc sử dụng và trong việc hiểu nó, vì 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.

2.4.4.2. Nhược điểm

- Class Facade của bạn có thể trở lên quá lớn, làm quá nhiều nhiệm vụ với nhiều hàm chức năng trong nó.
- Dễ bị phá vỡ các quy tắc trong SOLID.
- Việc sử dụng Facade cho các hệ thống đơn giản, không quá phức tạp trở nên dư thừa.

2.5. Command Pattern

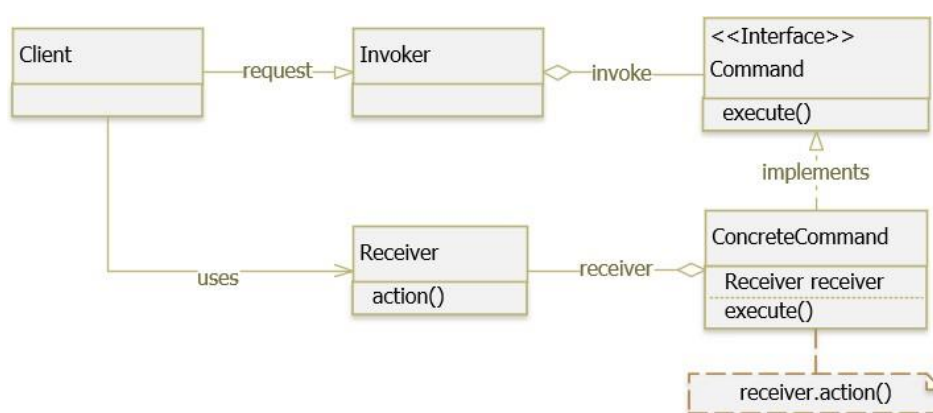
2.5.1. Khái niệm

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.

2.5.2. Cách cài đặt



Hình 25 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.

2.5.3. Ứng dụng

- 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.

2.5.4. Ưu/nhược điểm

2.5.4.1. Ưu điểm

- 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.

2.5.4.2. Nhược điểm

Mã có thể trở nên phức tạp hơn vì bạn đang giới thiệu một lớp hoàn toàn mới giữa sender và receiver.

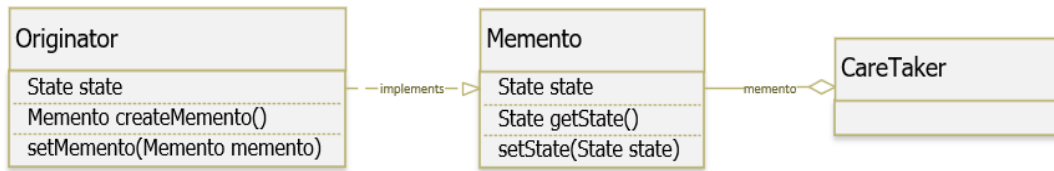
2.6. Memento Pattern

2.6.1. Khái niệm

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.

2.6.2. Cách cài đặt



Hình 26 Sơ đồ UML mô tả Memento Pattern

Các thành phần tham gia trong Memento Pattern:

- **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.

2.6.3. Ứng dụng

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.

2.6.4. Ưu/nhược điểm

2.6.4.1. Ưu điểm

- 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.

2.6.4.2. Nhược điểm

- Ứng dụng tiêu thụ nhiều RAM và xử lý nếu clients tạo ra 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.

2.7. Visitor Pattern

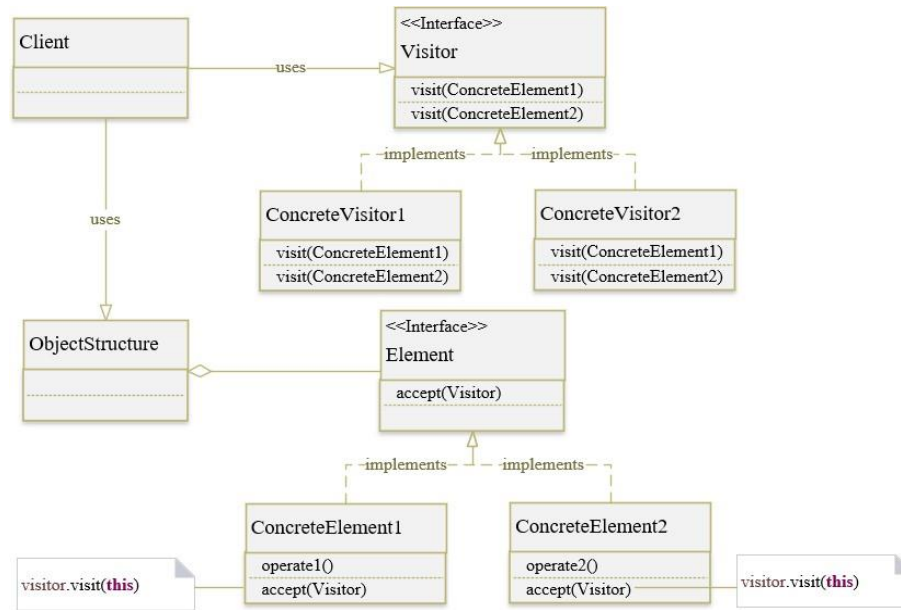
2.7.1. Khái niệm

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**.

2.7.2. Cách cài đặt



Hình 27 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.

2.7.3. Ứng dụng

- 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.

2.7.4. Ưu/nhược điểm

2.7.4.1. Ưu điểm

- 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.

2.7.4.2. Nhược điểm

Hạn chế lớn nhất của Visitor Pattern là chúng ta cần phải biết kiểu trả về của phương thức visit() tại thời điểm thiết kế nếu không chúng ta phải thay đổi interface và tất cả các cài đặt của nó. Như trong ví dụ trên, nếu chúng ta muốn thêm một loại sách khác (chẳng hạn OthersBook), chúng ta phải thêm phương thức visit(OthersBook) trong interface và sửa đổi tất cả các cài đặt tương ứng của Visitor

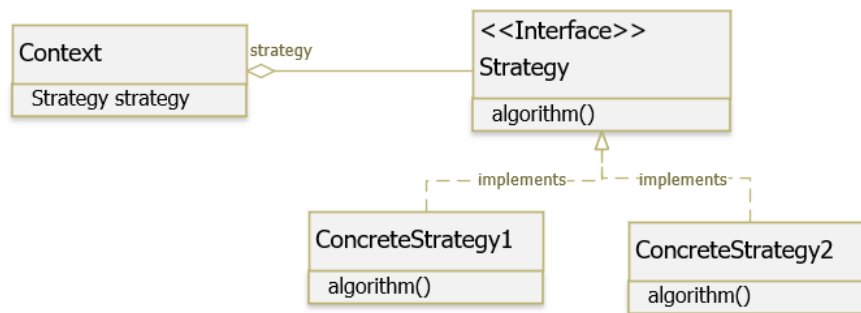
2.8. Strategy Pattern

2.8.1. Khái niệm

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.

2.8.2. Cách cài đặt



Hình 28 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.

2.8.3. Ứng dụng

- 3.3.1. Khi muốn có thể thay đổi các thuật toán được sử dụng bên trong một đối tượng tại thời điểm run-time.
- 3.3.2. Khi có một đoạn mã dễ thay đổi, và muốn tách chúng ra khỏi chương trình chính để dễ dàng bảo trì.
- 3.3.3. Tránh sự rắc rối, khi phải hiện thực một chức năng nào đó qua quá nhiều lớp con.
- 3.3.4. Cần che dấu sự phức tạp, cấu trúc bên trong của thuật toán.

2.8.4. Ưu/nhược điểm

2.8.4.1. Ưu điểm

- 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.

2.8.4.2. Nhược điểm

- Ứng dụng phải nhận thức được tất cả các chiến lược để chọn đúng chiến lược(strategy) cho tình huống phù hợp.
- Context và các lớp Strategy thường giao tiếp thông qua giao diện được chỉ định bởi lớp cơ sở Strategy trừu tượng. Lớp cơ sở chiến lược phải hiển thị giao diện cho tất cả các hành vi được yêu cầu, mà một số lớp Chiến lược cụ thể có thể không triển khai.
- Trong hầu hết các trường hợp, ứng dụng định cấu hình Context với đối tượng Strategy được yêu cầu. Do đó, ứng dụng cần tạo và duy trì hai đối tượng thay cho một.

2.9. Kết luận

Trong chương này, bài báo cáo 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: PROJECT MINH HỌA

3.1. GIỚI THIỆU

Design Pattern được ứng dụng nhiều trong phát triển phần mềm. Cách tiếp cận này đã đem lại nhiều lợi ích như phát triển nhanh, tái sử dụng, đảm bảo tính đúng đắn của phần mềm. Trong chương 3, bài báo cáo trình bày cách áp dụng Design Pattern trong phát triển phần mềm.

Do giới hạn của bài báo cáo, 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 bài báo cáo 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. HỆ THỐNG QUẢN LÝ SẢN PHẨM

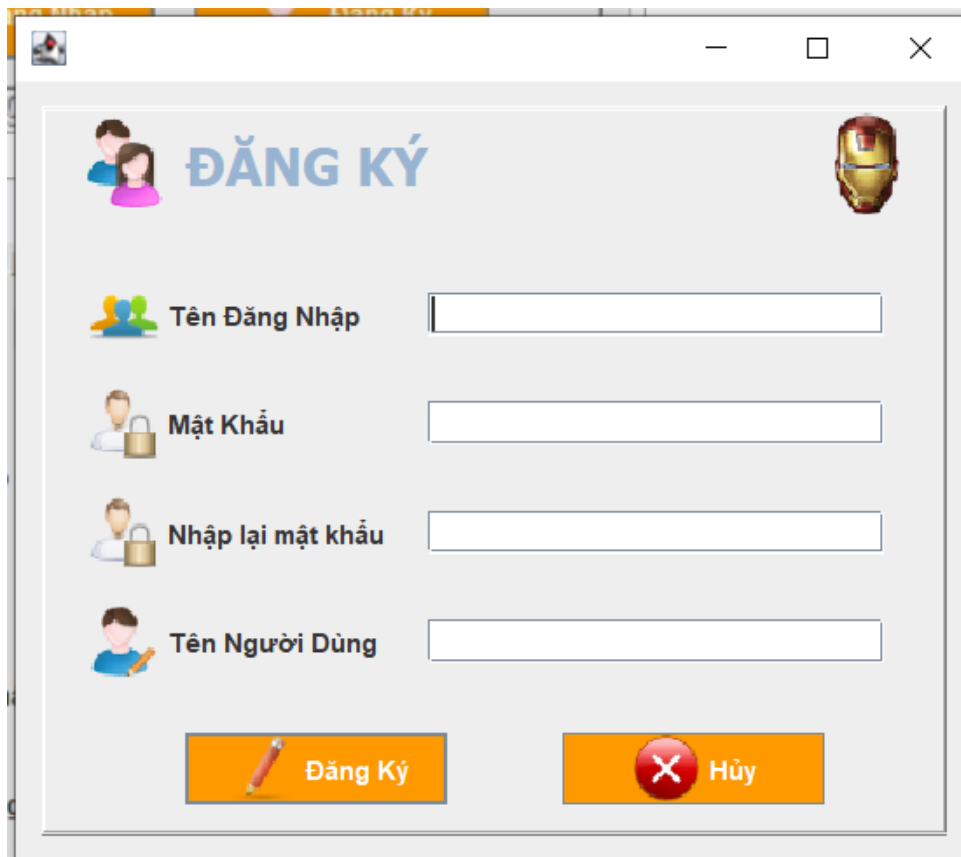
Ngày nay, dịch vụ bán hàng đang phát triển theo quy mô lớn. Số lượng sản phẩm đang ngày càng tăng, dẫn đến việc không thể quản lý tất cả được, dẫn đến sai sót. Nếu vẫn quản lý theo kiểu thủ công sẽ rất mất nhiều thời gian lẫn công sức, hiệu quả thì không được cao. Thậm chí sẽ bị nơi khác chiếm mất phần 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. CHƯƠNG TRÌNH THỰC NGHIỆM

3.3.1. Giao diện chương trình



Hình 29 Login vào hệ thống



Hình 30 Đăng ký tài khoản



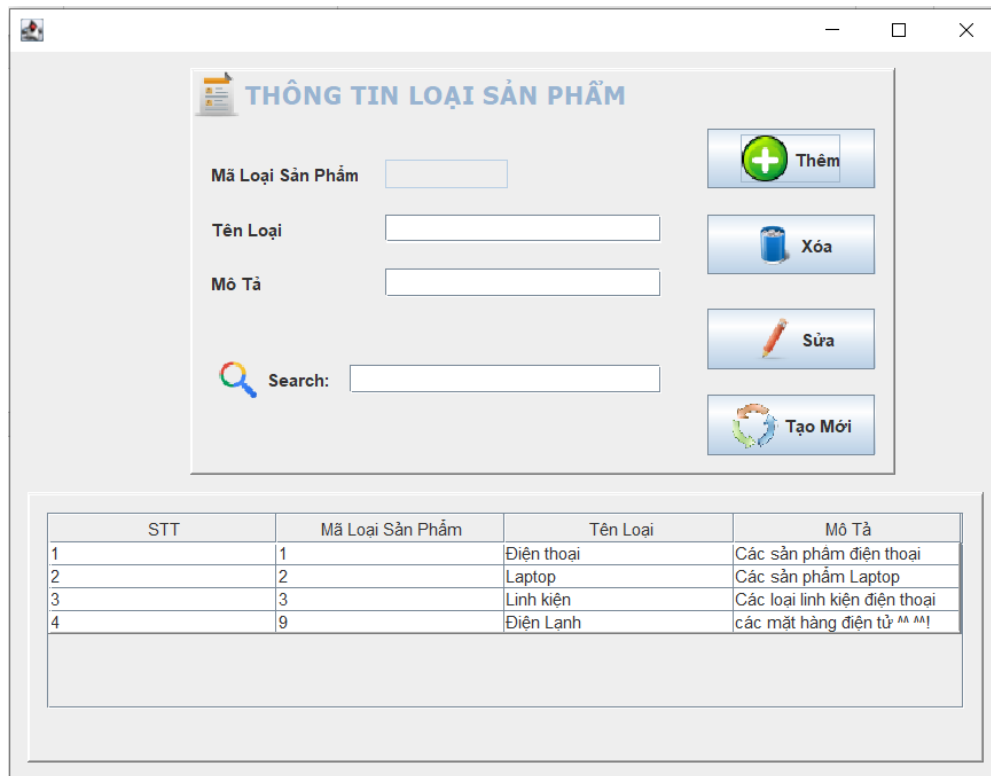
Hình 31 Giao diện chính quản lý sản phẩm

| STT | ID | Mã Sản P... | Tên Sản P... | Số Lượng | Đơn Vị Tính | Giá Nhập | Giá Bán | Mã Loại S... | Mô Tả |
|-----|----|-------------|----------------|----------|-------------|----------|----------|--------------|-------|
| 1 | 1 | DT01 | Zenfone 2 | 10 | Cái | 7000000 | 9000000 | Điện thoại | |
| 2 | 2 | DT02 | Zenfone 9 | 200 | Chiếc | 4000000 | 8000000 | Điện Lanh | |
| 3 | 4 | LT02 | Dell Inspir... | 10 | Cái | 11000000 | 13000000 | Laptop | |
| 4 | 5 | LK01 | Bao da IP... | 29 | Cái | 10000 | 50000 | Linh kiện | |
| 5 | 6 | LK02 | Miếng dán ... | 50 | Miếng | 20000 | 60000 | Điện thoại | |
| 6 | 8 | LT01 | Acer 511 | 9 | Cái | 17000000 | 19000000 | Laptop | |

Hình 32 Giao diện quản lý sản phẩm

| Mã Người Dùng | Tài Khoản | Mật Khẩu | Tên Người Dùng | Quyền |
|---------------|-----------|----------|--------------------|-----------|
| 1 | admin | admin | Nguyễn Thanh Mi... | ADMIN |
| 2 | tuannc | 123123 | Nguyễn Ngọc Tuấn | IT |
| 3 | danhtp | 123456 | Trần Phước Danh | ADMIN |
| 4 | taolhpk | 123123 | Lê Hữu Tạo | Nhân Viên |
| 5 | tuan1994 | 123456 | Nguyễn Tuấn | Nhân Viên |

Hình 33 Giao diện quản lý người dùng



THÔNG TIN LOẠI SẢN PHẨM

Mã Loại Sản Phẩm:

Tên Loại:

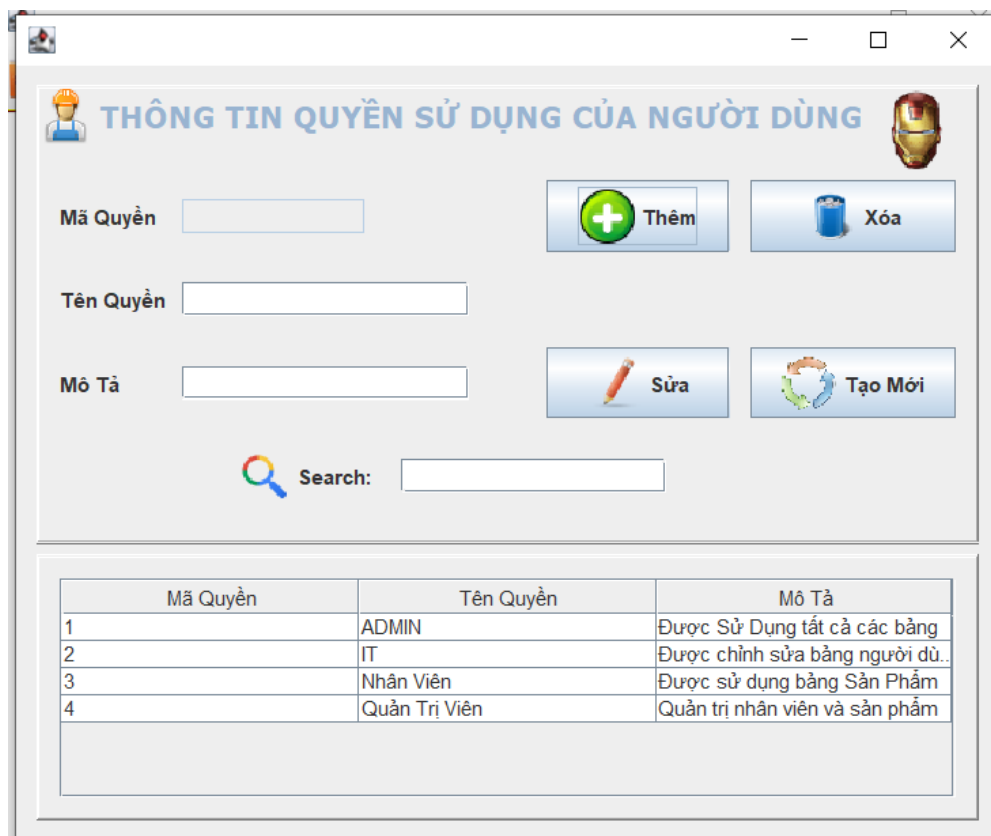
Mô Tả:

Search:

Buttons: Thêm, Xóa, Sửa, Tạo Mới

| STT | Mã Loại Sản Phẩm | Tên Loại | Mô Tả |
|-----|------------------|------------|-------------------------------|
| 1 | 1 | Điện thoại | Các sản phẩm điện thoại |
| 2 | 2 | Laptop | Các sản phẩm Laptop |
| 3 | 3 | Linh kiện | Các loại linh kiện điện thoại |
| 4 | 9 | Điện Lạnh | các mặt hàng điện tử AA AA! |

Hình 34 Giao diện quản lý loại sản phẩm



THÔNG TIN QUYỀN SỬ DỤNG CỦA NGƯỜI DÙNG

Mã Quyền:

Tên Quyền:

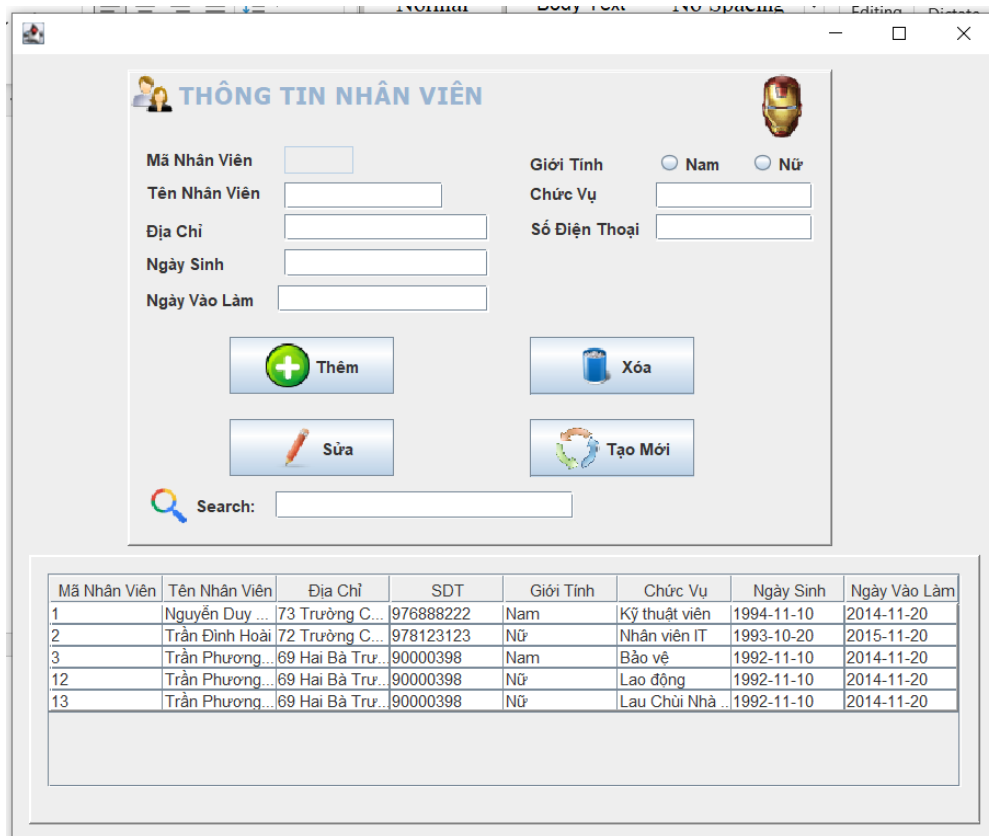
Mô Tả:

Search:

Buttons: Thêm, Xóa, Sửa, Tạo Mới

| Mã Quyền | Tên Quyền | Mô Tả |
|----------|---------------|--------------------------------|
| 1 | ADMIN | Được Sử Dụng tất cả các bảng |
| 2 | IT | Được chỉnh sửa bằng người dùng |
| 3 | Nhân Viên | Được sử dụng bằng Sản Phẩm |
| 4 | Quản Trị Viên | Quản trị nhân viên và sản phẩm |

Hình 35 Giao diện quản lý quyền người dùng



THÔNG TIN NHÂN VIÊN

Mã Nhân Viên:

Tên Nhân Viên:

Địa Chỉ:



Ngày Sinh:



Ngày Vào Làm:

Giới Tính: ☐ Nam ☐ Nữ

Chức Vụ:

Số Điện Thoại:

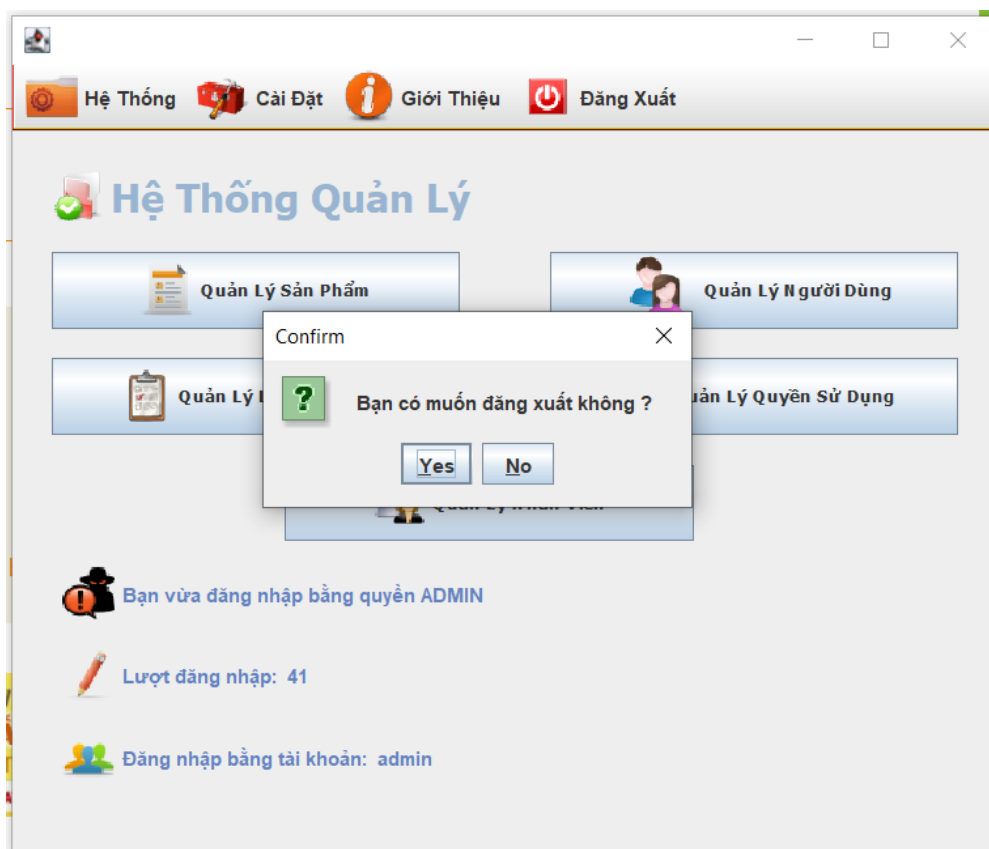
 Thêm  Xóa

 Sửa  Tạo Mới

Search:

| Mã Nhân Viên | Tên Nhân Viên | Địa Chỉ | SDT | Giới Tính | Chức Vụ | Ngày Sinh | Ngày Vào Làm |
|--------------|----------------|------------------|-----------|-----------|-----------------|------------|--------------|
| 1 | Nguyễn Duy ... | 73 Trường C... | 976888222 | Nam | Kỹ thuật viên | 1994-11-10 | 2014-11-20 |
| 2 | Trần Đình Hoài | 72 Trường C... | 978123123 | Nữ | Nhân viên IT | 1993-10-20 | 2015-11-20 |
| 3 | Trần Phương... | 69 Hai Bà Tru... | 90000398 | Nam | Bảo vệ | 1992-11-10 | 2014-11-20 |
| 12 | Trần Phương... | 69 Hai Bà Tru... | 90000398 | Nữ | Lao động | 1992-11-10 | 2014-11-20 |
| 13 | Trần Phương... | 69 Hai Bà Tru... | 90000398 | Nữ | Lau Chùi Nhà .. | 1992-11-10 | 2014-11-20 |

Hình 36 Giao diện quản lý nhân viên



Hình 37 Đăng xuất phần mềm

3.3.2. Chức năng

- *Lấy dữ liệu sản phẩm từ cơ sở dữ liệu*

```
private void LayDuLieu(){
    String SQL = "select * from SanPham";
    ResultSet rs = Main.connection.ExcuteQueryGetTable(SQL);
    Object [] obj = new Object[]{"STT", "ID", "Mã Sản Phẩm", "Tên Sản Phẩm", "Số Lượng",
        "Đơn Vị Tính", "Giá Nhập", "Giá Bán", "Mã Loại Sản Phẩm", "Mô Tả"};
    DefaultTableModel tableModel = new DefaultTableModel(obj, 0);
    tblSanPham.setModel(tableModel);
    try{
        while(rs.next()){
            Object [] item = new Object[10];
            item[0] = tblSanPham.getRowCount() + 1;
            item[1] = rs.getInt("IDSanPham");
            item[2] = rs.getString("MaSP");
            item[3] = rs.getString("TenSP");
            item[4] = rs.getInt("SoLuong");
            item[5] = rs.getString("DonViTinh");
            item[6] = rs.getInt("GiaNhap");
            item[7] = rs.getInt("GiaBan");
            item[8] = layTen(rs.getString("MaLoaiSP"));
            item[9] = rs.getString("MoTa");
            tableModel.addRow(item);
        }
    } catch (SQLException e) {
        System.out.println("nó đâu rồi");
    }
}

private void LayLoaiSanPham(){
    String SQL = "select * from LoaiSanPham";
    ResultSet rs = Main.connection.ExcuteQueryGetTable(SQL);
    DefaultComboBoxModel cbbModel = new DefaultComboBoxModel();
    try{
        while(rs.next()){
            DisplayComboBoxModel item = new DisplayComboBoxModel(rs.getString("TenLoai"), rs.getString("MaLoaiSP"));
            cbbModel.addElement(item);
        }
        cbbMaLoaiSanPham.setModel(cbbModel);
    } catch (SQLException e) {
        System.out.println("ò đây");
    }
}
```

Hình 38 Lấy dữ liệu sản phẩm từ CSDL

- *Thêm dữ liệu sản phẩm vào hệ thống*

```
private void btnThemActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    Main.playSound("soundbutton.wav");
    String maSP, tenSP, soLuong, donViTinh, giaNhap, giaBan, maLoai;
    maSP = txtMaSanPham.getText();
    tenSP = txtTenSanPham.getText();
    soLuong = txtSoLuong.getText();
    donViTinh = txtDonViTinh.getText();
    giaNhap = txtGiaNhap.getText();
    giaBan = txtGiaBan.getText();
    //
    Object [] obj = cbbMaLoaiSanPham.getSelectedObjects();
    DisplayComboBoxModel item = (DisplayComboBoxModel) obj[0];
    maLoai = item.DisplayValue.toString();

    String SQL = "insert into SanPham(MaSP,TenSP,SoLuong,DonViTinh,GiaNhap,GiaBan,MaLoaiSP) "
        + " values ('"+maSP+"',N'"+tenSP+"', '"+soLuong+"',N'"+donViTinh+"', "
        + " '"+giaNhap+"', '"+giaBan+"', '"+maLoai+"')";
    if("".equals(maSP) && "".equals(tenSP) && "".equals(donViTinh) && "".equals(giaNhap)){
        Main.thongBao("Bạn chưa nhập đầy đủ dữ liệu", "Thông Báo", 1);
    }
    else{
        Main.connection.ExcuteQueryUpdateDB(SQL);
    }
    LayDuLieu();
}
```

Hình 39 Chức năng thêm sản phẩm vào hệ thống

- *Sửa dữ liệu sản phẩm trong hệ thống*


```
private void btnSuaActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    Main.playSound("soundbutton.wav");
    String idSP, maSP, tenSP, soLuong, donViTinh, giaNhap, giaBan, maLoaiSP;
    idSP = txtIDSanPham.getText();
    maSP = txtMaSanPham.getText();
    tenSP = txtTenSanPham.getText();
    soLuong = txtSoLuong.getText();
    donViTinh = txtDonViTinh.getText();
    giaNhap = txtGiaNhap.getText();
    giaBan = txtGiaBan.getText();
    //
    Object [] obj = cbbMaLoaiSanPham.getSelectedObjects();
    DisplayComboBoxModel item = (DisplayComboBoxModel) obj[0];
    maLoaiSP = item.DisplayValue.toString();
    String SQL = "update SanPham set "
        + "MaSP = '"+maSP+"', TenSP = N'"+tenSP+"', "
        + "SoLuong = '"+soLuong+"', DonViTinh = N'"+donViTinh+"', "
        + "GiaNhap = '"+giaNhap+"', GiaBan = '"+giaBan+"', MaLoaiSP = '"+maLoaiSP+" "
        + "where IDSanPham = "+idSP;
    Main.connection.ExcuteQueryUpdateDB(SQL);
    LayDuLieu();
}
```

Hình 40 Chức năng sửa sản phẩm trong hệ thống

- Xóa dữ liệu sản phẩm trong hệ thống

```
private void btnXoaActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    Main.playSound("soundbutton.wav");
    int[] index = tblSanPham.getSelectedRows();
    for(int i = 0; i < index.length; i++){
        String maCanXoa = tblSanPham.getValueAt(index[i], 1).toString();
        String SQL = "delete from SanPham where IDSanPham = "+maCanXoa;
        Main.connection.ExcuteQueryUpdateDB(SQL);
    }
    LayDuLieu();
}
```

Hình 41 Chức năng xóa sản phẩm khỏi hệ thống

- Kết nối dữ liệu lên Server

```
static Connection conn = null;
public dsConnectDatabase() {
    try {
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        conn = DriverManager.getConnection("jdbc:sqlserver://LAPTOP-49D67JGL\\MINHDUC:1433;"
            + "databaseName=QLSanPham;user=sa;password=123456"
            + "intergratedSecurity=true;encrypt=true;trustServerCertificate=true");

        if(conn != null){
            System.out.println("Kết nối CSDL SQL Server thành công!");
        }
        else{
            System.out.println("Kết nối CSDL SQL Server thất bại");
        }
    }
    catch (ClassNotFoundException | SQLException ex) {
        System.out.println(ex.toString());
    }
}
```

Hình 42 Áp dụng Builder Pattern vào quy trình kết nối dữ liệu

- Đăng nhập vào hệ thống

```
private void btnDangNhapActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    Main.playSound("soundbutton.wav");
    // Hiện thị lượt đăng nhập:
    String SQL = "update LuotDangNhap set SoLuotDangNhap = SoLuotDangNhap + 1";
    Main.connection.ExcuteQueryUpdateDB(SQL);
    String SQLLayGiaTri = "select * from LuotDangNhap";
    ResultSet rss = Main.connection.ExcuteQueryGetTable(SQLLayGiaTri);
    try{
        while(rss.next()){
            Main.LuotDangNhap = rss.getInt("SoLuotDangNhap");
        }
    }
    catch (SQLException e) {
        System.out.println(e.toString());
    }
    // Hiện thị tên đăng nhập:
    Main.hienThiTenNguoiDung = txtTenDangNhap.getText();
    //-----
    String tenDangNhap = txtTenDangNhap.getText().trim();
    String matKhai = String.valueOf(txtMatKhai.getPassword()).trim();
    if(kiemTra(tenDangNhap,matKhai)){
        try{
            Thread.sleep(2000);
        }
        catch (InterruptedException ex) {
            System.out.println(ex.toString());
        }
        Main.thongBao("Đăng Nhập Thành Công", "Thông Báo", 1);
        frmQuanLy frmql = new frmQuanLy();
        frmql.show();
        this.dispose();
    }
    else if("").equals(txtTenDangNhap.getText())){
        Main.thongBao("Tên đăng nhập không được bỏ trống", "Thông Báo", 1);
        txtTenDangNhap.requestFocus();
    }
    else if("").equals(txtMatKhai.getText())){
        Main.thongBao("Mật khẩu không được bỏ trống", "Thông Báo", 1);
        txtMatKhai.requestFocus();
    }
    else{
        Main.thongBao("Bạn nhập sai tài khoản hoặc mật khẩu", "Thông Báo", 1);
    }
}
}
```

Hình 43 Chức năng đăng nhập vào hệ thống

- Kiểm tra quyền người dùng khi đăng nhập

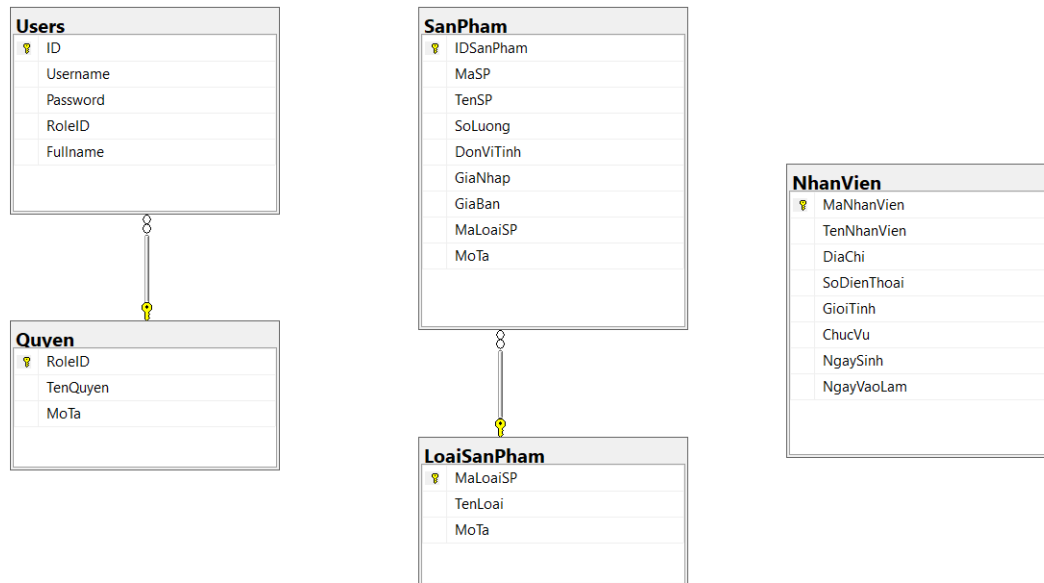
```
public static int Quyen = 0;

private boolean kiemTra(String tenDangNhap, String matKhai){
    boolean ketQua = false;
    String cauTruyVan = "select * from Users where Username = '"+tenDangNhap+"' "
        + "and Password = '"+matKhai+"'";
    ResultSet rs = Main.connection.ExcuteQueryGetTable(cauTruyVan);
    try{
        if(rs.next()){
            ketQua = true;
            Quyen = rs.getInt("RoleID");
        }
    }
    catch (SQLException e) {
        System.out.println("Lỗi Đăng Nhập");
    }
    return ketQua;
}
```

Hình 44 Chức năng kiểm tra quyền

3.3.3. Mô tả các nghiệp vụ

3.3.3.1. Mô hình quan hệ



Hình 45 Biểu đồ Class Diagram

3.3.3.2. Mô tả các nghiệp vụ

| STT | Tên Table | Mục Đích |
|-----|-------------|--|
| 1 | SanPham | Chứa thông tin sản phẩm. |
| 2 | LoaiSanPham | Chứa thông tin cá các loại sản phẩm |
| 3 | NhanVien | Chứa thông tin nhân viên và phân quyền |
| 4 | Users | Chứa thông tin tài khoản của các User |
| 5 | Quyen | Chứa thông tin quyền hạn |

CHƯƠNG 4: TỔNG KẾT

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 Huỳnh Xuân Phụng, 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, chúng 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.

TÀI LIỆU THAM KHẢO

- [1]. <https://shorturl.at/tADFK>
- [2]. <https://tuanitpro.com/design-pattern-la-gi>
- [3]. <https://blog.duyet.net/2015/02/oop-design-patterns-la-gi>
- [4]. <https://refactoring.guru/design-patterns/catalog>
- [5]. <https://developer.ibm.com/languages/java/>
- [6]. <https://www.journaldev.com/1827/java-design-patterns-example-tutorial>
- [7]. <http://surl.li/bvpgk>
- [8]. <http://kenhsinhvien.net/forum/topic/...ke-he-thong-thong-tin-150-bai-mau/>
- [9]. <http://svth.net/forum/threads/25334/>
- [10]. <http://www.voer.edu.vn/giao-trinh/khoa-hoc-va-cong-nghe/>