Đại học Quốc gia
Thành phố Hồ Chí Minh

Trường Đại học Khoa học Tự nhiên

khoa công nghệ thông tin

Báo cáo đồ án OOP Video Project

CSC10003 – Phương Pháp Lập Trình Hướng Đối Tượng

Ngô Nguyễn Thế Khoa 23127065
Phi Anh Khôi 23127073
Khưu Ngọc Ý Vy 23127145
Trần Phụng Đình 23127527

Ngày 31 tháng 12 năm 2024

Mục lục

1 Thông tin nhóm

- **Môn:** Phương pháp lập trình hướng đối tượng
- **Lớp học phần:** 23CLC09
- **Giảng viên hướng dẫn:** Bùi Tiến Lên
- **Thành viên:**

STT	Họ và tên	MSSV	Email
1	Ngô Nguyễn Thế Khoa	23127065	nntkhoa23@clc.fitus.edu.vn
2	Phi Anh Khôi	23127073	pakhoi23@clc.fitus.edu.vn
3	Khưu Ngọc Ý Vy	23127145	knyvy23@clc.fitus.edu.vn
4	Trần Phụng Đình	23127527	tpdinh23@clc.fitus.edu.vn

- **Công cụ hỗ trợ:**
  - **Git**, **GitHub**: Quản lý mã nguồn
  - **Google Docs**: Soạn thảo tài liệu
  - **StarUML**: Vẽ UML
  - **CapCut**: Dựng video
  - **ChatGPT**, **Gemini**: Sinh ý tưởng dự án, phân tích yêu cầu và lựa chọn Design Pattern phù hợp
  - **Natural Readers**: Chuyển văn bản thành giọng nói tiếng Việt
  - **Visual Studio Code**: Soạn thảo mã nguồn (C++ và Latex)

2 Nội dung

2.1 Thông tin chung về đồ án:

- **Tên đồ án:** Design Pattern Video Project
- **Môi trường lập trình:** Visual Studio Code (Windows)
- **Ngôn ngữ lập trình:** C++
- **Thư viện sử dụng:**
  - **raylib:** Thư viện đồ họa
  - **uuid_v4:** Thư viện tạo 'UUIDv4'
  - **stb_image_write**: Thư viện tạo ảnh từ dữ liệu
  - **qrcode_gen**: Thư viện tạo mã QR từ dữ liệu

3 Bảng phân công việc

STT	Công việc	Người thực hiện
1	Dùng AI sinh ý tưởng và lựa chọn Design Pattern phù hợp	Khoa, Khôi, Vy, Đình
2	Tìm hiểu, làm slide và viết report về 'Factory Method'	Khôi
3	Tìm hiểu, làm slide và viết report về 'Decorator'	Vy
4	Tìm hiểu, làm slide và viết report về 'Chain of Responsibility'	Đình
5	Làm demo application, vẽ UML và hoàn thiện report	Khoa
6	Làm video	Khoa, Khôi, Vy, Đình

4 Ý tưởng dự án và lựa chọn Design Pattern

4.1 Ý tưởng dự án

- **Tên dự án:** E-commerce Application
- **Mô tả:** Ứng dụng mô phỏng một trang web bán hàng trực tuyến, bao gồm các chức năng cơ bản như:
  - Xem danh sách sản phẩm
  - Thêm và xóa sản phẩm khỏi giỏ hàng
  - Xem giỏ hàng, chỉnh sửa số lượng sản phẩm
  - Áp dụng mã giảm giá
  - Điền thông tin giao hàng (địa chỉ, số điện thoại)
  - Chọn đơn vị vận chuyển
  - Chọn phương thức thanh toán (COD, Credit Card, PayPal, Stripe,...)
  - Thanh toán đơn hàng
  - Xem lịch sử đơn hàng
- **Mục tiêu:** Áp dụng **Design Pattern** vào ứng dụng để tăng tính linh hoạt, dễ bảo trì và mở rộng.
- **Note:** Ý tưởng dự án được sinh từ ChatGPT và có thể xem ở đây.

4.2 Lựa chọn Design Pattern

Phân tích vấn đề

- **Yêu cầu:** Cần xây dựng một ứng dụng E-commerce linh hoạt, dễ bảo trì và mở rộng.
- **Vấn đề:**
  - Làm sao để tích hợp đa dạng các hình thức thanh toán và cổng thanh toán mà tránh việc sửa đổi mã nguồn nhiều và chỉ cần bổ sung thêm khi cần?
  - Với một đơn hàng cơ bản, làm sao để bổ sung thêm đơn vị vận chuyển hay quà tặng mà không trực tiếp sửa đổi đơn hàng cơ bản?
  - Giả sử quy trình xử lí đơn hàng có thể thay đổi, làm sao để thay đổi (thêm, sửa, xóa) quy trình mà không ảnh hưởng đến các bước khác?
- **Giải pháp:** Cần lựa chọn và sử dụng các Design Pattern phù hợp để giải quyết những vấn đề trên.

Các Design Pattern phù hợp

- **Tính linh hoạt và mở rộng**
  - **Thêm mới phương thức thanh toán:** Với *Factory Method*, việc thêm mới một cổng thanh toán (ví dụ: PayPal, Stripe) chỉ cần tạo thêm một lớp con của lớp Payment Gateway và cập nhật logic trong Factory. Điều này giúp tránh sửa đổi nhiều phần mã nguồn khi cần bổ sung tính năng mới.
  - **Thêm tính năng bổ sung cho đơn hàng:** *Decorator Pattern* cho phép thêm các tính năng bổ sung cho đơn hàng (ví dụ: gói quà tặng, giao hàng nhanh, bảo hiểm) mà không cần sửa đổi trực tiếp lớp `Order` cơ bản. Điều này giúp ứng dụng dễ dàng tùy chỉnh và đáp ứng nhu cầu đa dạng của khách hàng.
  - **Thay đổi quy trình xử lý đơn hàng:** *Chain of Responsibility Pattern* giúp dễ dàng thay đổi hoặc thêm mới các bước xử lý đơn hàng (ví dụ: kiểm tra tồn kho, xác thực địa chỉ) mà không ảnh hưởng đến các bước khác. Điều này tăng tính linh hoạt trong quản lý và tối ưu hóa quy trình kinh doanh.
- **Dễ bảo trì và sửa chữa**
  - **Giảm sự phụ thuộc giữa các thành phần:** Các Design Pattern giúp giảm sự liên kết chặt chẽ giữa các thành phần trong ứng dụng, giúp dễ dàng bảo trì và sửa chữa các phần riêng lẻ mà không ảnh hưởng đến toàn bộ hệ thống.
  - **Tái sử dụng mã nguồn:** Các Pattern khuyến khích việc viết mã nguồn một cách trừu tượng và tái sử dụng được, giúp tiết kiệm thời gian và công sức phát triển.
  - **Dễ dàng kiểm thử:** Các Pattern thường giúp tách biệt các chức năng, giúp việc kiểm thử đơn vị và tích hợp trở nên dễ dàng hơn.
- **Nâng cao hiệu suất**
  - **Tái sử dụng đối tượng:** *Factory Method* giúp tránh việc tạo ra nhiều đối tượng giống nhau, giúp tiết kiệm tài nguyên hệ thống.
  - **Tối ưu hóa quá trình xử lý:** *Chain of Responsibility Pattern* giúp tối ưu hóa quá trình xử lý đơn hàng bằng cách chỉ thực hiện các bước cần thiết.

5 Chi tiết về các Design Pattern được sử dụng

5.1 Factory Method

Factory Method là một mẫu thiết kế sáng tạo (creational design pattern) cung cấp một giao diện để tạo đối tượng trong lớp cha (superclass), nhưng cho phép các lớp con (subclasses) thay đổi loại đối tượng sẽ được tạo ra.

Vấn đề

- Ban đầu, hệ thống chỉ hỗ trợ thanh toán qua Credit Card và hoạt động hiệu quả làm tăng doanh thu. Tuy nhiên, khi số lượng khách hàng tăng càng tăng, cần đa dạng hóa hình thức thanh toán (PayPal, Stripe, v.v).
- Việc này gặp khó khăn vì mã nguồn cũ chỉ tập trung vào Credit Card, dẫn đến việc hệ thống trở nên phức tạp và khó quản lý khi thêm các hình thức thanh toán mới.

Mục đích

Sử dụng *Factory Method* để tách biệt quá trình tạo đối tượng khỏi logic sử dụng đối tượng, giúp cho code trở nên linh hoạt, dễ bảo trì và mở rộng hơn. Cụ thể:

- **Ẩn đi quá trình tạo đối tượng:** Thay vì khởi tạo đối tượng trực tiếp bằng từ khóa `new`, chúng ta sẽ gọi một phương thức `factory` để tạo ra đối tượng. Điều này giúp ẩn đi các chi tiết phức tạp của quá trình tạo đối tượng, làm cho code trở nên sạch sẽ và dễ đọc hơn.
- **Tăng tính linh hoạt:** *Factory Method* cho phép chúng ta thay đổi loại đối tượng được tạo ra mà không cần sửa đổi nhiều phần code. Điều này đặc biệt hữu ích khi cần thêm các loại đối tượng mới vào hệ thống.
- **Tuân thủ nguyên tắc Open-Closed:** Hệ thống có thể mở rộng để thêm các loại đối tượng mới mà không cần sửa đổi code hiện có.
- **Tăng khả năng kiểm soát:** Chúng ta có thể thêm các logic kiểm tra, xác thực hoặc cấu hình vào quá trình tạo đối tượng thông qua `factory method`.

Giải pháp

Cài đặt phương thức ảo 'Factory': Thay vì tạo mới loại hình thanh toán một cách trực tiếp, giờ đây chúng ta sẽ tạo mới thông qua các phương thức ảo trên.

- Điều này cho phép những loại hình thanh toán mới có thể được thêm vào hệ thống và điều chỉnh độc lập ở những lớp con thay vì phải điều chỉnh lại toàn bộ code của hệ thống.
- Tuy nhiên ở lớp con cần phải tuân thủ trả về đầy đủ các kiểu trả về của các `factory method` đã được khai báo ở lớp cha.

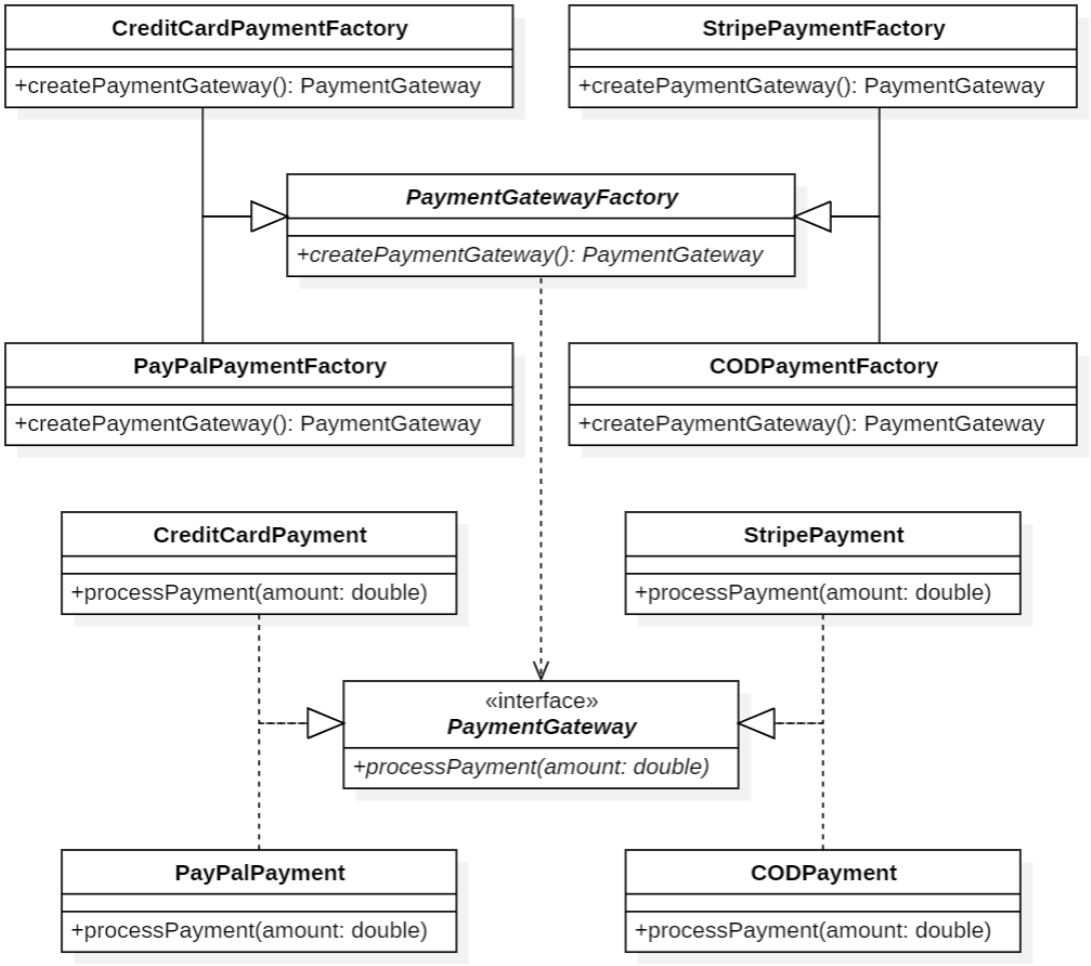
Cấu trúc

1. **PaymentGateway:** Là giao diện (interface) xác định một hợp đồng chung cho tất cả các phương thức thanh toán. Phương thức `processPayment(amount: double)` đại diện cho việc xử lý một giao dịch thanh toán với một số tiền nhất định.

2. **CreditCardPayment, StripePayment, PayPalPayment, CODPayment:** Đây là các lớp cụ thể thực hiện giao diện PaymentGateway, mỗi lớp đại diện cho một phương thức thanh toán khác nhau (thẻ tín dụng, Stripe, PayPal, thanh toán khi giao hàng).

3. **PaymentGatewayFactory:** Đây là một lớp trừu tượng (abstract class) đại diện cho một nhà máy tạo ra các đối tượng `PaymentGateway`. Phương thức `createPaymentGateway()` là phương thức trừu tượng, nghĩa là các lớp con phải cài đặt lại phương thức này để tạo ra các đối tượng cụ thể.

4. **CreditCardPaymentFactory, StripePaymentFactory, PayPalPaymentFactory, CODPaymentFactory:** Đây là các lớp con của `PaymentGatewayFactory`, mỗi lớp sẽ thực hiện phương thức `createPaymentGateway()` để tạo ra một đối tượng `PaymentGateway` cụ thể tương ứng.



Hình 1: Factory Method UML Class Diagram

Hoạt động

- **Tạo đối tượng:** Khi cần tạo một đối tượng `PaymentGateway` cụ thể, thay vì khởi tạo trực tiếp bằng từ khóa `new`, chúng ta sẽ sử dụng một factory tương ứng. Ví dụ, để tạo một đối tượng thanh toán bằng thẻ tín dụng, ta sẽ gọi phương thức `createPaymentGateway()` của `CreditCardPaymentFactory`.
- **Đa hình:** Nhờ cơ chế đa hình, cùng một phương thức `processPayment()` có thể thực hiện các hành vi khác nhau tùy thuộc vào đối tượng cụ thể được gọi. Ví dụ, khi gọi `processPayment()` trên một đối tượng `CreditCardPayment`, hệ thống sẽ thực hiện các logic xử lý thanh toán bằng thẻ tín dụng.
- **Mở rộng:** Nếu muốn thêm một phương thức thanh toán mới, ta chỉ cần tạo một lớp con mới của `PaymentGateway` và một lớp factory tương ứng, không cần sửa đổi các lớp đã có.

Khả năng ứng dụng

- **Hệ thống thanh toán:** Giúp quản lý nhiều phương thức thanh toán khác nhau một cách linh hoạt.
- **Factory:** Tạo ra các sản phẩm khác nhau dựa trên các thông số đầu vào.
- **Xây dựng giao diện người dùng:** Tạo ra các đối tượng giao diện khác nhau dựa trên các cấu hình.

Ưu nhược điểm

- **Ưu điểm**
  - **Tách biệt quá trình tạo đối tượng:** Quá trình tạo đối tượng được tách biệt khỏi logic sử dụng đối tượng, giúp code rõ ràng hơn và dễ bảo trì.
  - **Tăng tính linh hoạt:** Dễ dàng thêm các phương thức thanh toán mới mà không ảnh hưởng đến phần còn lại của hệ thống.
  - **Ẩn đi các chi tiết triển khai:** Người dùng chỉ cần biết giao diện PaymentGateway, không cần quan tâm đến cách các phương thức thanh toán được thực hiện bên trong.
  - **Dễ kiểm thử:** Mỗi factory có thể được kiểm thử độc lập.
- **Nhược điểm**
  - Mã có thể trở nên phức tạp hơn vì cần tạo thêm nhiều lớp con mới để triển khai mẫu thiết kế.
  - Việc có quá nhiều lớp factory có thể làm cho hệ thống trở nên khó quản lý và khó bảo trì (maintain).

Mã nguồn

```
// Abstract Payment Gateway
class PaymentGateway {
 public:
  virtual ~PaymentGateway() = default;
  virtual pair<bool, string> processPayment(const double&) = 0;
};

// Concrete Payment Gateways
class CreditCardPayment : public PaymentGateway {
 public:
  pair<bool, string> processPayment(const double& amount) override {
    return {true, "Processed Credit Card payment of $" +
                  utils::toStringWithPrecision(amount, 2) + "\n"};
  }
};

class PayPalPayment : public PaymentGateway {
 public:
  pair<bool, string> processPayment(const double& amount) override {
    return {true, "Processed Paypal payment of $" +
                  utils::toStringWithPrecision(amount, 2) + "\n"};
  }
};

class StripePayment : public PaymentGateway {
 public:
  pair<bool, string> processPayment(const double& amount) override {
    return {true, "Processed Stripe payment of $" +
                  utils::toStringWithPrecision(amount, 2) + "\n"};
  }
};

class CODPayment : public PaymentGateway {
 public:
  pair<bool, string> processPayment(const double& amount) override {
    return {true, "Processed COD payment of $" +
                  utils::toStringWithPrecision(amount, 2) + "\n"};
  }
};

// Abstract Factory
class PaymentGatewayFactory {
 public:
  virtual ~PaymentGatewayFactory() = default;
  virtual unique_ptr<PaymentGateway> createPaymentGateway() const = 0;
};
```

```
47
48  // Concrete Factories
49  class CreditCardPaymentFactory : public PaymentGatewayFactory {
50   public:
51    unique_ptr<PaymentGateway> createPaymentGateway() const override {
52      return make_unique<CreditCardPayment>();
53    }
54  };
55
56  class PayPalPaymentFactory : public PaymentGatewayFactory {
57   public:
58    unique_ptr<PaymentGateway> createPaymentGateway() const override {
59      return make_unique<PayPalPayment>();
60    }
61  };
62
63  class StripePaymentFactory : public PaymentGatewayFactory {
64   public:
65    unique_ptr<PaymentGateway> createPaymentGateway() const override {
66      return make_unique<StripePayment>();
67    }
68  };
69
70  class CODPaymentFactory : public PaymentGatewayFactory {
71   public:
72    unique_ptr<PaymentGateway> createPaymentGateway() const override {
73      return make_unique<CODPayment>();
74    }
75  };
```

5.2 Decorator

Decorator là một mẫu thiết kế cấu trúc cho phép mở rộng hoặc thêm chức năng mới cho một đối tượng động mà không làm thay đổi trực tiếp lớp của nó. 'Decorator' hoạt động bằng cách bọc (wrap) đối tượng gốc, sau đó tùy chỉnh hoặc mở rộng hành vi của nó.

Vấn đề

- Hãy tưởng tượng bạn đang làm việc với một hệ thống xử lý đơn hàng thương mại điện tử. Ban đầu, hệ thống này rất đơn giản: mỗi đơn hàng chỉ chứa danh sách các sản phẩm và thông tin thanh toán cơ bản. Mọi thứ đều theo một quy trình mặc định, từ thanh toán đến vận chuyển tiêu chuẩn, không có tùy chọn bổ sung nào khác.
- Mọi thứ vẫn vận hành suôn sẻ cho tới khi khách hàng đưa ra các yêu cầu mới:
  - *'Tôi muốn đơn hàng được giao hỏa tốc vì tôi cần nó ngay bây giờ'*
  - *'Sản phẩm này là một món quà, hãy đóng gói thật đẹp giúp tôi'*
  - *'Hàng của tôi dễ vỡ, tôi muốn thêm bảo hiểm để đảm bảo an toàn'*
  - ...
- Bạn cố gắng cập nhật hệ thống để đáp ứng các yêu cầu này bằng cách thêm các thuộc tính mới vào lớp đơn hàng (như *isGiftWrap, isExpressDelivery, hasInsurance...*). Tuy nhiên điều này nhanh chóng trở nên phức tạp bởi vì:
  - Mỗi yêu cầu bổ sung dẫn đến thêm nhiều điều kiện *if-else* vào mã nguồn.
  - Các phương thức xử lý đơn hàng trở nên cồng kềnh và khó bảo trì.
  - Khi số lượng dịch vụ bổ sung tăng lên, việc tổ hợp các tùy chọn trở thành một cơn ác mộng.
- Lúc này, bạn nhận ra cần một cách tiếp cận tốt hơn để mở rộng chức năng của đơn hàng mà không làm phức tạp thêm cấu trúc lớp gốc.

Mục đích

- Để giải quyết vấn đề được đặt ra, bạn bắt đầu nghĩ đến kế thừa (Inheritance) để thay đổi lớp ban đầu. Tuy nhiên kế thừa lại có một số hạn chế như sau:
  - **Tính chất tĩnh:** Hành vi được xác định tại thời điểm biên dịch, khó thay đổi khi chạy.
  - **Không hỗ trợ đa kế thừa:** Nhiều ngôn ngữ lập trình không cho phép đa kế thừa, hạn chế khả năng mở rộng.
- Bạn tiếp tục tìm đến những giải pháp khác là tập hợp (Aggregation) hoặc thành phần (Composition) để khắc phục nhược điểm của kế thừa:
  - Cho phép đối tượng tham chiếu và ủy quyền nhiệm vụ cho các đối tượng khác.
  - Thay đổi hành vi linh hoạt tại thời điểm chạy.

- Hai nguyên tắc này nền tảng cho nhiều mẫu thiết kế, đặc biệt là `Decorator Pattern`.

Giải pháp

1. **Cơ chế của Decorator**
   Decorator Pattern mở rộng hành vi bằng cách 'bọc' đối tượng trong các lớp mở rộng, mỗi lớp thêm hoặc thay đổi chức năng mà không ảnh hưởng đến đối tượng gốc.

   - **Tham chiếu đối tượng gốc:** Các lớp Decorator chứa tham chiếu đến đối tượng cần bọc (có thể là đối tượng cơ bản hoặc một Decorator khác).
   - **Giao diện thống nhất:** Decorator và đối tượng gốc tuân theo cùng một giao diện hoặc lớp trừu tượng. Điều này giúp Decorator có thể thay thế đối tượng gốc mà không làm thay đổi logic tổng thể.
   - **Thêm hành vi từng bước:** Decorator thực hiện logic riêng trước hoặc sau khi gọi phương thức của đối tượng bọc.
   - **Kết hợp các tính năng:**
     - Đơn hàng cơ bản: sử dụng *BasicOrder*.
     - Đơn hàng cần gói quà và vận chuyển nhanh: sử dụng *GiftWrapDecorator* và *ExpressDeliveryDecorator* bọc quanh *BasicOrder*.

2. **Ứng dụng vào hệ thống xử lý đơn hàng**

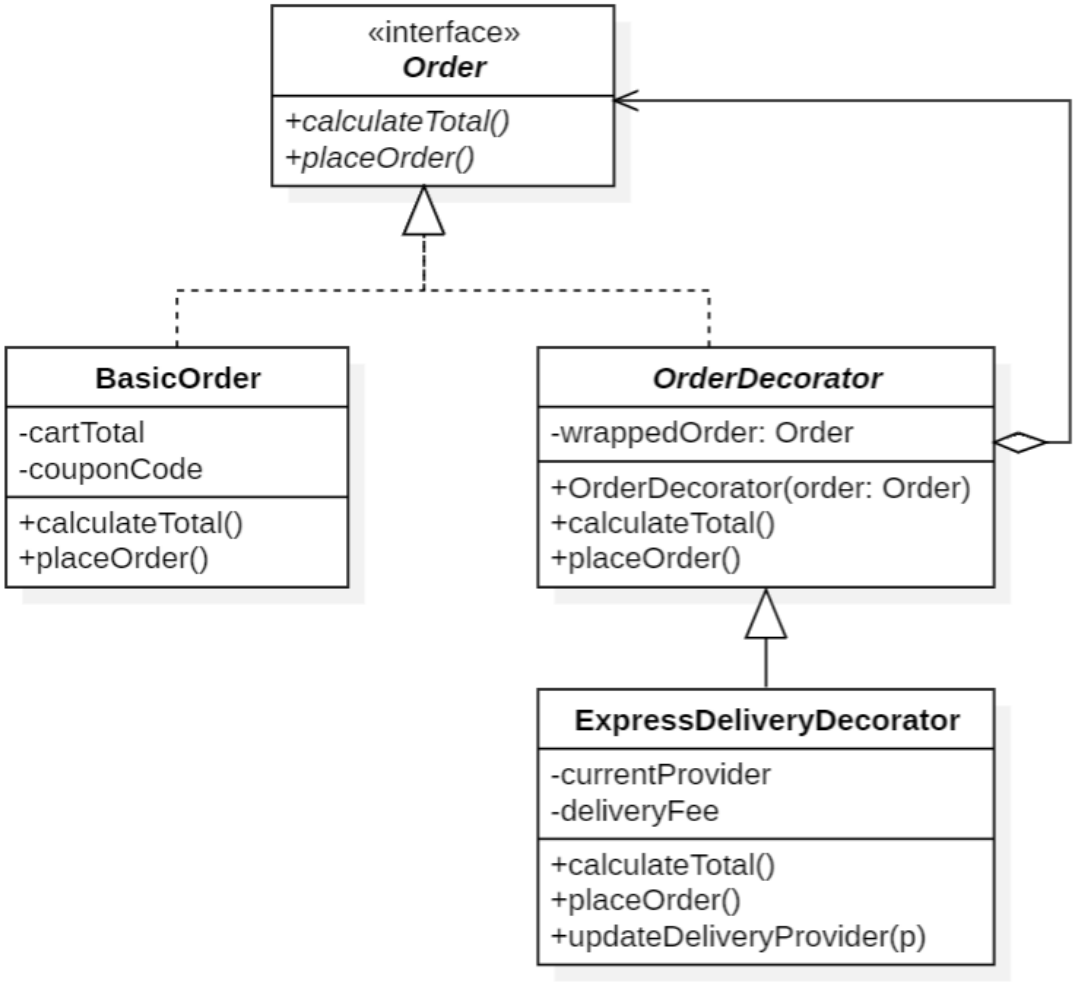
   - **Đối tượng cơ bản:** *BasicOrder* là đơn hàng tiêu chuẩn, chỉ chứa thông tin sản phẩm, chi phí cơ bản và mã giảm giá.
   - **Thêm Decorator theo yêu cầu:**
     - *GiftWrapDecorator*: Thêm phí và mô tả gói quà.
     - *ExpressDeliveryDecorator*: Thêm phí và đơn vị giao hàng nhanh.
   - **Xử lý tuần tự:** Khi gọi phương thức như `calculateTotal()`, mỗi lớp Decorator sẽ thêm logic riêng, sau đó chuyển lời gọi tới lớp tiếp theo, tạo nên một luồng xử lý linh hoạt và có tổ chức.

Cấu trúc

1. **Order:** Đây là một giao diện (interface) xác định các hành vi cơ bản của một đơn hàng, bao gồm tính toán tổng tiền và đặt hàng.
2. **BasicOrder:** Lớp cụ thể thực hiện giao diện `Order`, đại diện cho một đơn hàng cơ bản. Lớp này có các thuộc tính như tổng giá trị giỏ hàng, mã giảm giá và các phương thức để tính toán tổng tiền và đặt hàng.
3. **OrderDecorator:** Đây là một lớp trừu tượng (abstract class) đóng vai trò là lớp cơ sở cho các lớp 'decorator'. Lớp này có một thuộc tính tham chiếu đến một đối tượng `Order` khác (`wrappedOrder`) và các phương thức để tính toán tổng tiền và

đặt hàng. Các phương thức này thường sẽ gọi đến các phương thức tương ứng của `wrappedOrder` và thực hiện thêm các logic của các 'decorator' khác.

4. **ExpressDeliveryDecorator:** Lớp cụ thể kế thừa từ `OrderDecorator`, đại diện cho một lớp 'decorator' thêm chức năng giao hàng nhanh cho đơn hàng. Lớp này có các thuộc tính như nhà cung cấp giao hàng, phí giao hàng và các phương thức để tính toán tổng tiền bao gồm cả phí giao hàng và cập nhật nhà cung cấp giao hàng.



Hình 2: Decorator Pattern UML Class Diagram

Hoạt động

- **Tạo đối tượng BasicOrder:** Đầu tiên, chúng ta tạo một đối tượng `BasicOrder` để đại diện cho một đơn hàng cơ bản.
- **Decorator:** Sau đó, chúng ta có thể tạo các đối tượng `OrderDecorator` để 'trang trí' cho đối tượng `BasicOrder`. Ví dụ, chúng ta tạo một đối tượng `ExpressDeliveryDecorator` và truyền đối tượng `BasicOrder` vào trong constructor của nó.
- **Gọi phương thức:** Khi gọi các phương thức của đối tượng được trang trí (ví dụ: `calculateTotal()`), các phương thức này sẽ gọi đến các phương thức tương ứng của đối tượng được gói bên trong (BasicOrder) và thực hiện thêm các logic trang

trí. Trong trường hợp của `ExpressDeliveryDecorator`, phương thức `calculateTotal()` sẽ tính toán tổng tiền của đơn hàng cơ bản và cộng thêm phí giao hàng.

Khả năng ứng dụng

- **Tạo ra các loại đơn hàng đa dạng:**
  - **BasicOrder:** Đơn hàng tiêu chuẩn, chỉ chứa thông tin sản phẩm, chi phí cơ bản và mã giảm giá.
  - **ExpressDeliveryDecorator(BasicOrder):** Đơn hàng với giao hàng nhanh, có thêm phí và đơn vị giao hàng.
- **Linh hoạt và mở rộng:** Để thêm một tính năng mới (ví dụ: gói quà), chỉ cần tạo một lớp decorator mới kế thừa từ OrderDecorator và triển khai logic thêm tính năng đó.
- **Tách biệt mối quan tâm:** Mỗi lớp decorator chỉ tập trung vào một tính năng cụ thể, giúp code dễ đọc, dễ bảo trì và dễ kiểm thử hơn.

Ưu nhược điểm

1. **Ưu điểm**
   - **Linh hoạt:** Dễ dàng thêm các tính năng mới cho một đối tượng mà không cần thay đổi lớp gốc.
   - **Mở rộng:** Có thể kết hợp nhiều lớp trang trí để tạo ra các hành vi phức tạp.
   - **Tái sử dụng:** Các lớp trang trí có thể được tái sử dụng cho nhiều loại đối tượng khác nhau.
2. **Nhược điểm**
   - Việc loại bỏ một lớp Decorator cụ thể trong ngăn xếp không đơn giản.
   - Hành vi của Decorator phụ thuộc vào thứ tự áp dụng.
   - Việc có cấu trúc lớp phức tạp và khó khăn trong việc xác định hành vi của đối tượng khiến việc đọc hiểu mã nguồn trở nên khó khăn.

Mã nguồn

```
// Abstract Order
class Order {
 public:
  virtual ~Order() = default;
  virtual Price calculateTotal() const = 0;
  virtual pair<bool, vector<string>> placeOrder() = 0;
};

```

```
// Concrete Order: Basic Order
class BasicOrder : public Order {
 private:
  Price cartTotal;
  string couponCode = "";

 public:
  BasicOrder() = default;
  BasicOrder(Price, const string&);

  const Price& getCartTotal() const { return cartTotal; };
  const string& getCouponCode() const { return couponCode; };

  void setCartTotal(const Price&);
  void setCouponCode(const string& _);

  Price calculateTotal() const override;
  pair<bool, vector<string>> placeOrder() override;
};

// Abstract Decorator
class OrderDecorator : public Order {
 protected:
  shared_ptr<Order> wrappedOrder;

 public:
  OrderDecorator() = delete;
  OrderDecorator(shared_ptr<Order> order) : wrappedOrder(move(order)) {}

  Order* unwrap() override {
    return wrappedOrder ? wrappedOrder->unwrap() : this;
  }
};

// Concrete Decorator: Express Delivery
class ExpressDeliveryDecorator : public OrderDecorator {
 public:
  inline static map<string, Price> availableProviders = {
      {"J&T Express", Price(23'15, 2)},
      {"Viettel Post", Price(25'35, 2)},
      {"Ninja Van", Price(21'55, 2)}
  };

 private:
  string currentProvider;
  Price expressFee;

 public:
  ExpressDeliveryDecorator(shared_ptr<Order>);
```

```
58
59      void updateDeliveryProvider(const string&);
60
61      const string& getCurrentDeliveryProvider() const;
62      const Price& getExpressFee() const;
63
64      Price calculateTotal() const override;
65      pair<bool, vector<string>> placeOrder() override;
66  };
```

5.3 Chain of Responsibility

Chain of Responsibility là một mẫu thiết kế hành vi (behavioral design pattern) cho phép bạn truyền yêu cầu qua một chuỗi các bộ xử lý (handlers). Khi nhận được một yêu cầu, mỗi bộ xử lý sẽ quyết định hoặc xử lý yêu cầu đó hoặc chuyển nó cho bộ xử lý tiếp theo trong chuỗi.

Vấn đề

- **Xử lý yêu cầu phức tạp:** Khi một yêu cầu cần được xử lý qua nhiều giai đoạn khác nhau, việc quản lý luồng xử lý và các điều kiện chuyển tiếp giữa các giai đoạn có thể trở nên phức tạp.
- **Mở rộng hệ thống:** Khi cần thêm hoặc xóa các giai đoạn xử lý, việc sửa đổi toàn bộ hệ thống có thể gây ra nhiều rắc rối.
- **Tách biệt mối quan tâm:** Các logic xử lý khác nhau được phân tán vào các lớp khác nhau, giúp code dễ đọc và bảo trì hơn.

Mục đích

- **Tách biệt các bước xử lý:** Mỗi bước xử lý được đóng gói trong một đối tượng riêng biệt (handler).
- **Xử lý theo chuỗi:** Các handler được liên kết thành một chuỗi, yêu cầu sẽ được truyền qua chuỗi cho đến khi được xử lý hoặc đạt đến cuối chuỗi.
- **Linh hoạt:** Dễ dàng thêm, xóa hoặc thay đổi thứ tự các handler mà không ảnh hưởng đến các phần còn lại của hệ thống.

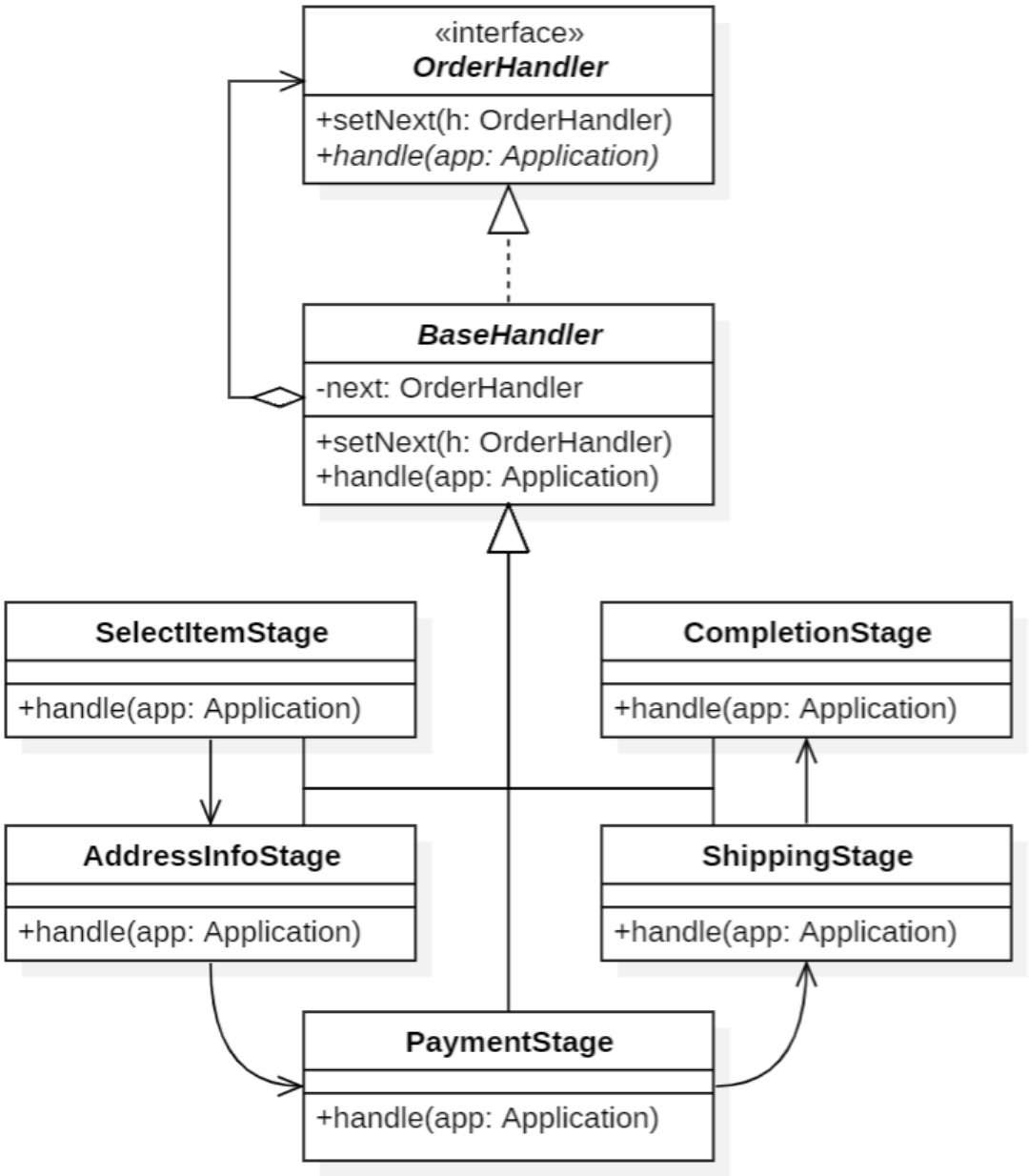
Giải pháp

- Giống như nhiều mẫu thiết kế hành vi khác, *Chain of Responsibility* dựa trên việc biến đổi các hành vi cụ thể thành các đối tượng độc lập gọi là *handlers*.
- Mỗi công đoạn nên được trích xuất thành một lớp riêng biệt với một phương thức duy nhất thực hiện hành động cụ thể theo từng công đoạn. Yêu cầu, cùng với dữ liệu của nó, được truyền vào phương thức này như một đối số
- Mỗi handler được liên kết có một trường để lưu trữ tham chiếu đến handler tiếp theo trong chuỗi. Ngoài việc xử lý yêu cầu, các handlers còn truyền yêu cầu tiếp tục dọc theo chuỗi. Yêu cầu di chuyển dọc theo chuỗi cho đến khi tất cả các handlers đều có cơ hội xử lý nó.

Cấu trúc

- **OrderHandler:** Giao diện chung cho tất cả các handler, định nghĩa hai phương thức chính:
  - `setNext`: Thiết lập handler tiếp theo trong chuỗi.
  - `handle`: Xử lý yêu cầu.

- **BaseHandler:** Lớp trừu tượng triển khai giao diện `OrderHandler`, cung cấp một thuộc tính `next` để lưu trữ handler tiếp theo.
- **SelectItemStage, AddressInfoStage, PaymentStage, ShippingStage, CompletionStage:** Các lớp cụ thể kế thừa từ `BaseHandler`, đại diện cho các giai đoạn xử lý khác nhau trong một đơn hàng. Mỗi giai đoạn sẽ thực hiện một nhiệm vụ cụ thể và quyết định có chuyển yêu cầu đến giai đoạn tiếp theo hay không.



Hình 3: Chain of Responsibility UML Class Diagram

Hoạt động

- **Tạo chuỗi handler:** Các handler được khởi tạo và liên kết với nhau thành một chuỗi.
- **Gửi yêu cầu:** Yêu cầu được gửi đến handler đầu tiên trong chuỗi.

- **Xử lý yêu cầu:**
  - Nếu handler hiện tại có thể xử lý yêu cầu, nó sẽ thực hiện các tác vụ cần thiết và dừng quá trình.
  - Nếu handler hiện tại không thể xử lý, nó sẽ chuyển yêu cầu đến handler tiếp theo trong chuỗi.
- **Lặp lại bước 3:** Quá trình này tiếp tục cho đến khi yêu cầu được xử lý hoặc đạt đến cuối chuỗi.

Khả năng ứng dụng

- **Xử lý đơn hàng:** *CoR Pattern* được sử dụng rộng rãi trong các hệ thống xử lý đơn hàng để quản lý các giai đoạn khác nhau như thanh toán, vận chuyển.
- **Xử lý sự kiện:** Có thể sử dụng CoR để xử lý các sự kiện trong giao diện người dùng, chẳng hạn như các sự kiện click chuột để chuyển sang công đoạn kế.
- **Logging:** Mỗi handler có thể ghi lại thông tin về quá trình xử lý yêu cầu.

Ưu nhược điểm

1. **Ưu điểm**
   - **Linh hoạt:** Dễ dàng thêm, xóa hoặc thay đổi các handler.
   - **Mở rộng:** Có thể dễ dàng mở rộng hệ thống bằng cách thêm các handler mới.
   - **Tái sử dụng:** Các handler có thể được sử dụng lại trong các hệ thống khác.
   - **Giảm sự kết hợp:** Các handler không cần biết về các handler khác trong chuỗi.
2. **Nhược điểm**
   - **Khó phát hiện lỗi và debug:** Nếu chuỗi (chain) quá dài và phức tạp, việc tìm ra lỗi có thể khó khăn.
   - **Hiệu suất:** Nếu có quá nhiều handler, việc truyền yêu cầu qua chuỗi có thể làm giảm hiệu suất.

Mã nguồn

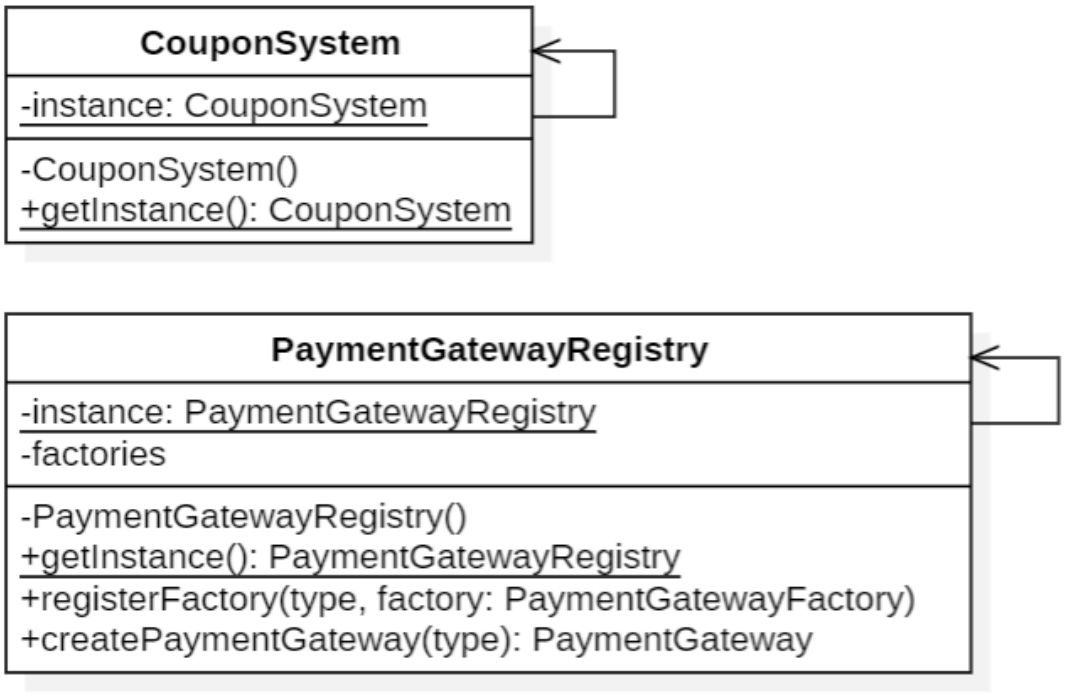
```
// Abstract Order Stage Handler
class OrderHandler {
 protected:
  shared_ptr<OrderHandler> nextStage;

 public:
  OrderHandler() = default;
  virtual ~OrderHandler() = default;

```

```
10    void setNext(const shared_ptr<OrderHandler> &next) { nextStage = next; }
11
12    virtual void handle(OrderStageSystem &, OrderContext &, GUI *, Application *) = 0;
13    virtual void render(OrderContext &, GUI *, Application *) = 0;
14  };
15
16  // Order Stage Context
17  class OrderContext {
18   public:
19    OrderStageState currentStage;
20
21    CartType cart;
22    Price totalCost;
23    Price discount;
24
25    string address;
26    string phone;
27
28    string deliveryProvider;
29
30    PaymentMethod paymentMethod;
31    pair<string, Price> paymentInfo;
32
33   public:
34    OrderContext();
35    ~OrderContext() = default;
36
37    Price getFinalCost() const {
38      return totalCost > discount ? totalCost - discount : Price(0, 0);
39    }
40  };
41
42  // Concrete Order Stages
43  class SelectItemStage : public OrderHandler {
44   public:
45    void handle(OrderStageSystem &, OrderContext &, GUI *, Application *) override;
46    void render(OrderContext &, GUI *, Application *) override;
47  };
48
49  class AddressInfoStage : public OrderHandler {
50   public:
51    void handle(OrderStageSystem &, OrderContext &, GUI *, Application *) override;
52    void render(OrderContext &, GUI *, Application *) override;
53  };
54
55  class ShippingStage : public OrderHandler {
56   public:
57    void handle(OrderStageSystem &, OrderContext &, GUI *, Application *) override;
58    void render(OrderContext &, GUI *, Application *) override;
```

```
59 };
60
61 class PaymentStage : public OrderHandler {
62  public:
63   void handle(OrderStageSystem &, OrderContext &, GUI *, Application *) override;
64   void render(OrderContext &, GUI *, Application *) override;
65 };
66
67 class CompletionStage : public OrderHandler {
68  public:
69   void handle(OrderStageSystem &, OrderContext &, GUI *, Application *) override;
70   void render(OrderContext &, GUI *, Application *) override;
71 };
```

5.4 Singleton

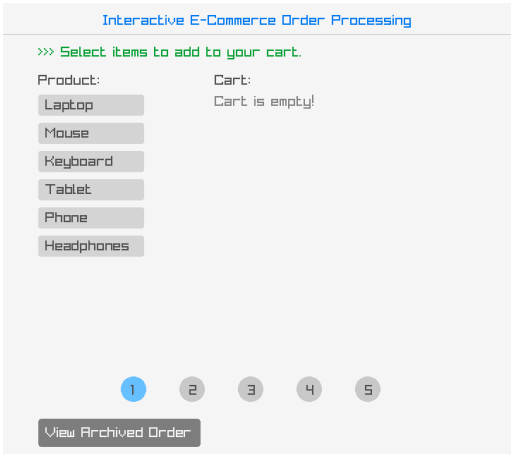
Ngoài 3 Design Patterns chính, nhóm còn sử dụng thêm một Design Pattern khác, đó là *Singleton*. **Singleton** là một Design Pattern cung cấp một cách để giới hạn việc tạo ra một đối tượng của một lớp đến một đối tượng duy nhất. Điều này đảm bảo rằng một lớp chỉ có một thể hiện và cung cấp một cách để truy cập nó.



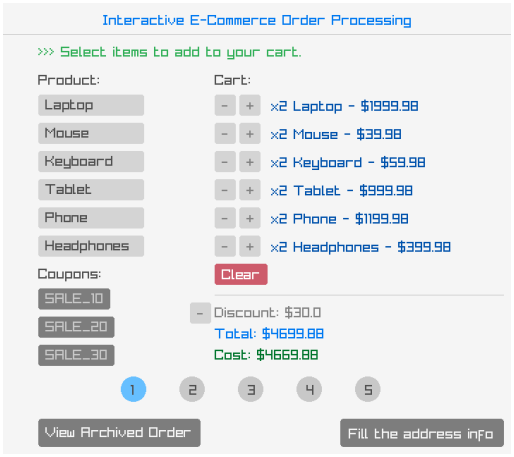
Hình 4: Factory Method UML Class Diagram

- **CouponSystem**:
  - **Vai trò:** Là điểm vào duy nhất để truy cập các chức năng liên quan đến mã giảm giá, như áp dụng mã giảm giá cho đơn hàng, chứa các logic kiểm tra tính hợp lệ của mã giảm giá, quản lý thời hạn sử dụng, v.v.
- **PaymentGatewayRegistry**:
  - **Vai trò:** Là một registry để quản lý các phương thức thanh toán khác nhau, cho phép hệ thống thêm các phương thức thanh toán mới một cách linh hoạt bằng cách đăng ký các factory tương ứng.

6 Ứng dụng Demo

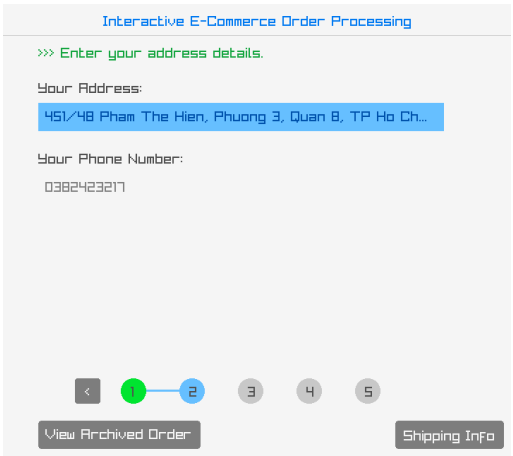


(a) Giao diện bắt đầu

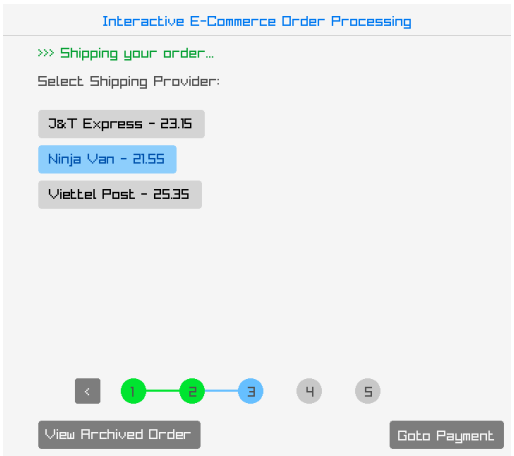


(b) Giao diện giỏ hàng với giảm giá

Hình 5: Xem, thêm, xóa sản phẩm trong giỏ hàng và áp mã giảm giá

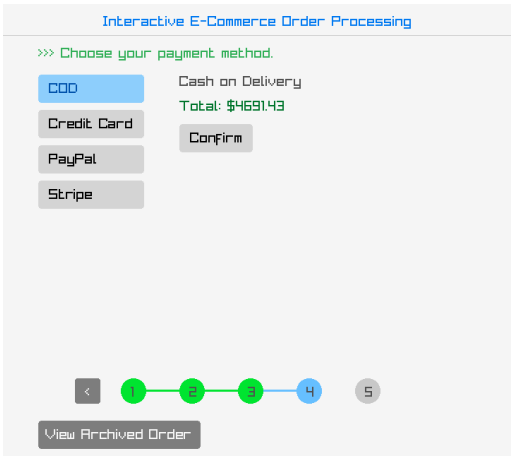


(a) Giao diện nhập thông tin địa chỉ



(b) Giao diện chọn đơn vị vận chuyển

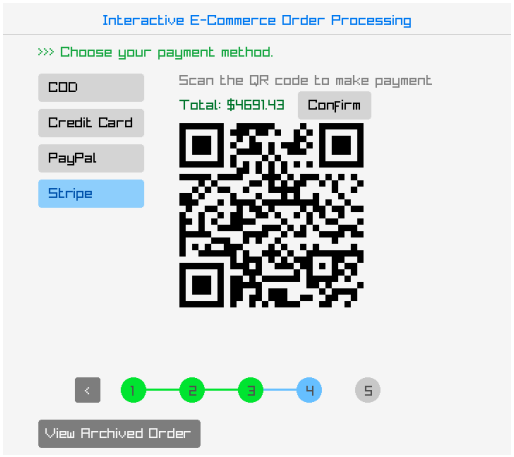
Hình 6: Nhập thông tin địa chỉ và chọn đơn vị vận chuyển



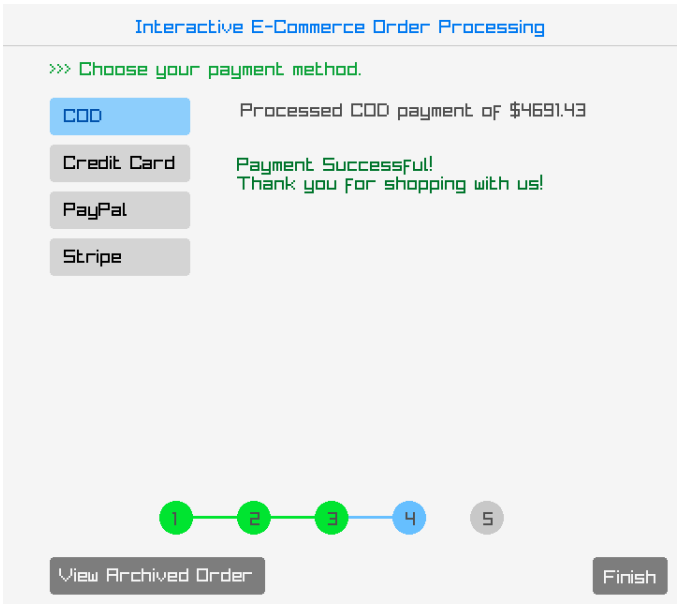
(a) Giao diện chọn phương thức thanh toán COD

(b) Giao diện chọn phương thức thanh toán thẻ tín dụng

(c) Giao diện chọn phương thức thanh toán PayPal

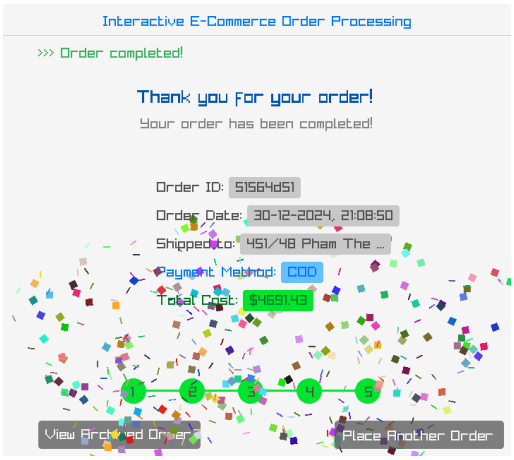


(d) Giao diện chọn phương thức thanh toán Stripe

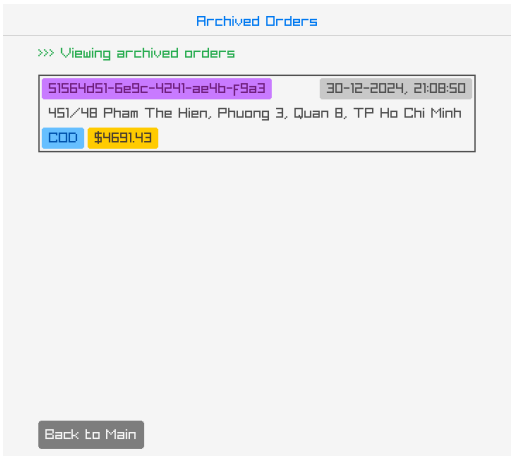


(e) Giao diện thanh toán bằng COD thành công

Hình 7: Chọn phương thức thanh toán và thanh toán thành công



(a) Giao diện hoàn tất đơn hàng



(b) Giao diện xem đơn hàng đã hoàn tất

Hình 8: Hoàn tất đơn hàng và xem đơn hàng đã hoàn tất

7 Nguồn tham khảo

1. Class Diagram in UML
2. Refactoring Guru (PDF)
3. Raylib Examples