

# Nguyên lý Thiết kế và Kiến trúc Phần mềm

## Chương 2: Nguyên lý SOLID trong thiết kế

Single  
responsibility



Open-Closed  
Principle

Liskov  
substitution



Interface  
segregation

Dependency  
inversion




# Giới thiệu về SOLID

- SOLID là một nguyên tắc thiết kế phần mềm được đưa ra bởi Robert C. Martin (còn được gọi là Uncle Bob) nhằm tạo ra các hệ thống phần mềm dễ bảo trì, dễ mở rộng, dễ hiểu và dễ tái sử dụng

# SOLID là viết tắt của 5 nguyên tắc

---

- S - Single Responsibility Principle (Nguyên tắc đơn trách nhiệm): Một lớp chỉ nên có một trách nhiệm duy nhất.
- O - Open/Closed Principle (Nguyên tắc đóng mở): Các thành phần phần mềm nên được thiết kế để dễ mở rộng và khó thay đổi.
- L - Liskov Substitution Principle (Nguyên tắc thay thế Liskov): Các đối tượng của lớp con nên có thể thay thế các đối tượng của lớp cha mà không làm thay đổi tính đúng đắn của chương trình.
- I - Interface Segregation Principle (Nguyên tắc phân chia giao diện): Nên thiết kế các giao diện nhỏ, cụ thể hơn thay vì một giao diện lớn tổng hợp nhiều chức năng.
- D - Dependency Inversion Principle (Nguyên tắc đảo ngược phụ thuộc): Các thành phần phần mềm nên phụ thuộc vào abstraction, chứ không phải concrete.



## Mục tiêu của các nguyên tắc SOLID

Những nguyên tắc này khi được áp dụng đúng cách sẽ giúp cho mã nguồn phần mềm dễ bảo trì, dễ hiểu, dễ mở rộng và dễ tái sử dụng.

## Single Responsibility Principle (Nguyên tắc đơn trách nhiệm)

---

Nguyên tắc đơn trách nhiệm (SRP) là một trong những nguyên tắc cơ bản trong thiết kế phần mềm. SRP khuyến khích ta thiết kế các lớp và phương thức chỉ đảm nhiệm một trách nhiệm duy nhất.





# Định nghĩa và lợi ích của SRP

- Định nghĩa: Mỗi lớp trong hệ thống chỉ nên có một trách nhiệm duy nhất và không nên có quá nhiều lớp cùng thực hiện một trách nhiệm. Lợi ích: Giúp tăng tính module hóa, giảm sự phức tạp của hệ thống, dễ dàng bảo trì và mở rộng.
-



# Cách triển khai SRP

- Cách triển khai: Chúng ta nên xác định các trách nhiệm của mỗi lớp và chỉ cung cấp các phương thức liên quan đến trách nhiệm đó. Nếu một lớp có quá nhiều trách nhiệm, chúng ta nên tách thành các lớp nhỏ hơn để mỗi lớp chỉ đảm nhiệm một trách nhiệm duy nhất.
-

# Open/Closed Principle (Nguyên tắc đóng mở)

Nguyên tắc đóng mở (OCP) là một trong những nguyên tắc cơ bản trong thiết kế phần mềm. OCP khuyến khích ta thiết kế các lớp và phương thức có thể mở rộng để thêm các tính năng mới mà không cần sửa đổi mã nguồn hiện có.





# Định nghĩa và lợi ích của OCP

- Định nghĩa: Mã nguồn phần mềm nên được thiết kế sao cho có thể mở rộng để thêm các tính năng mới mà không cần sửa đổi mã nguồn hiện có. Lợi ích: Giúp giảm tối thiểu việc sửa đổi mã nguồn cũ khi thêm các tính năng mới, tăng tính linh hoạt và dễ dàng bảo trì và mở rộng.



# Cách triển khai OCP

- Cách triển khai: Chúng ta nên thiết kế các lớp và phương thức để có thể mở rộng mà không cần sửa đổi mã nguồn hiện có. Các lớp và phương thức nên được thiết kế sao cho có tính trừu tượng và độc lập với các lớp và phương thức khác trong hệ thống.
-

## Trường hợp không sử dụng OCP

- Ở trường hợp không sử dụng OCP, việc tính phí vận chuyển được xử lý bên trong phương thức `calculateShipping()` của `Shipping` và bị phụ thuộc vào kiểu của `Shipper`. Khi có thêm một đơn vị vận chuyển mới cần hỗ trợ, ta phải sửa đổi phương thức `calculateShipping()` trong `Shipping`. Điều này làm tăng rủi ro cho mã nguồn và làm cho việc bảo trì và mở rộng hệ thống trở nên phức tạp hơn.

**C** Order

• `submitOrder()`

**C** Shipping

• `calculateShipping()`

**C** ShipperA

• `calculateShipping()`

**C** ShipperB

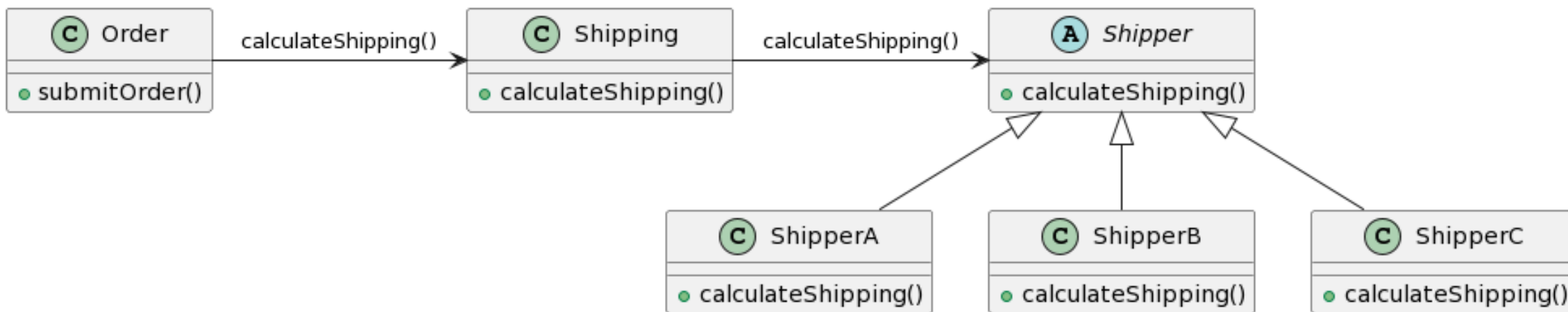
• `calculateShipping()`

**C** ShipperC

• `calculateShipping()`

# Trường hợp sử dụng OCP

- Trong trường hợp sử dụng OCP, việc tính phí vận chuyển được xử lý bên trong phương thức `calculateShipping()` của các lớp con của lớp trừu tượng `Shipper`. Khi có thêm một đơn vị vận chuyển mới cần hỗ trợ, ta chỉ cần tạo một lớp mới kế thừa từ `Shipper` và triển khai phương thức `calculateShipping()` mà không cần sửa đổi phương thức `calculateShipping()` của lớp `Shipping`.



# Nguyên tắc L - Liskov Substitution Principle (Nguyên tắc thay thế Liskov)

Nguyên tắc Liskov Substitution (LSP) là một nguyên tắc quan trọng trong lập trình hướng đối tượng.

Nguyên tắc này đảm bảo rằng các đối tượng con của một lớp có thể thay thế lớp cha mà không làm thay đổi tính đúng đắn của chương trình.

# Nguyên tắc L - Liskov Substitution Principle (Nguyên tắc thay thế Liskov)

Cụ thể, theo LSP, đối tượng của lớp con phải đáp ứng được tất cả các đặc điểm của đối tượng của lớp cha, bao gồm cả hành vi (method) và thuộc tính (attribute), và không nên có hành vi không mong đợi hoặc hành vi phụ thuộc vào kiểu của đối tượng.

# Nguyên tắc I - Interface Segregation Principle (Nguyên tắc phân chia giao diện)

Nguyên tắc này khuyến khích chia nhỏ các giao diện thành các phần nhỏ hơn, cụ thể là tách các giao diện lớn thành các giao diện nhỏ hơn, dễ quản lý và có tính độc lập cao.

## Nguyên tắc D - Dependency Inversion Principle (Nguyên tắc đảo ngược phụ thuộc)

Nguyên tắc này khuyến khích lập trình viên tập trung vào việc thiết kế hệ thống phần mềm sao cho các module phụ thuộc vào một interface chung thay vì phụ thuộc vào nhau.



# Tại sao SOLID lại quan trọng?

---

SOLID giúp tăng tính linh hoạt, dễ bảo trì và mở rộng trong quá trình phát triển phần mềm.

---

Tuy nhiên, thực hiện SOLID có thể gặp phải một số khó khăn như phức tạp hóa mã nguồn, tăng thời gian phát triển và giảm hiệu suất thực thi.

Khi nào áp  
dụng nguyên  
tắc SOLID?

Áp dụng SOLID khi code đã có sự ổn định và đang phát triển theo hướng mở rộng.

Nếu code đang trong giai đoạn thử nghiệm, nên tập trung vào chức năng thay vì SOLID.

# Áp dụng nguyên tắc nào trong SOLID?

Tùy thuộc vào từng tình huống cụ thể, ta sẽ chọn áp dụng nguyên tắc nào trong SOLID để tối ưu hóa mã nguồn và giảm độ phức tạp.

Ví dụ: Nếu chương trình chỉ có một class đơn giản, ta có thể bỏ qua SOLID. Tuy nhiên, khi chương trình lớn hơn và có nhiều class phức tạp, ta cần áp dụng các nguyên tắc SOLID để tăng tính bảo trì và mở rộng.

- Không nhất thiết phải áp dụng hết các nguyên tắc SOLID trong mọi tình huống.
- Nếu ta thấy rằng việc áp dụng SOLID sẽ làm phức tạp và làm giảm hiệu suất thực thi, ta có thể bỏ qua một số nguyên tắc SOLID.
- Tuy nhiên, ta nên áp dụng ít nhất một vài nguyên tắc SOLID để tăng khả năng bảo trì và mở rộng mã nguồn.

Có cần áp dụng hết  
các nguyên tắc  
SOLID?

## Kết luận

- SOLID là một tập hợp các nguyên tắc quan trọng để tăng tính linh hoạt và dễ bảo trì trong quá trình phát triển phần mềm.
- Tuy nhiên, ta cần cân nhắc và áp dụng SOLID đúng cách để tránh gặp phải các khó khăn và giảm hiệu suất thực thi.
- Không nhất thiết phải áp dụng hết các nguyên tắc SOLID trong mọi tình huống, nhưng nên áp dụng ít nhất một vài nguyên tắc