论述利斯科夫替换原则（里氏代换原则）、单一职责原则、开闭原则、德（迪）米特法则、依赖倒转原则、合成复用原则，结合自己的实践项目举例说明如何应用 （保存到每个小组选定的协作开发平台上，以组为单位）

* 利斯科夫替换原则（里氏代换原则）：

在软件中如果能够使用基类对象，那么一定能够使用其子类对象。把基类都替换成它的子类，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类的话，那么它不一定能够使用基类。

实践项目应用：在支付时，定义抽象接口PaymentProcessor，子类实现接口并确保行为一致。调用方依赖抽象。

public interface PaymentProcessor {

void processPayment(Order order);

}

public class CreditCardPaymentProcessor implements PaymentProcessor {

public void processPayment(Order order) {

// 信用卡支付实现

}

}

public class CheckoutService {

private PaymentProcessor processor; // 依赖抽象

public CheckoutService(PaymentProcessor processor) {

this.processor = processor;

}

public void processOrder(Order order) {

processor.processPayment(order); // 子类可替换基类

}

}

* 单一职责原则

在软件系统中，一个类只负责一个功能领域中的相应职责。也可以说就一个类而言，应该仅有一个引起它变化的原因。

一个类（或者大到模块，小到方法）承担的职责越多，它被复用的可能性越小。而且如果一个类承担的职责过多，就相当于将这些职责耦合在一起，当其中一个职责变化时，可能会影响其他职责的运作。

实践项目应用：将用户注册、登录、信息更新和通知发送这些功能进行职责隔离。

public class UserRegistrationService {

public void registerUser(User user) { /\* 仅处理注册 \*/ }

}

public class UserAuthenticationService {

public boolean login(String username, String password) { /\* 仅处理登录 \*/ }

}

public class UserProfileService {

public void updateProfile(User user) { /\* 仅处理更新 \*/ }

}

public class NotificationService {

public void sendEmailNotification(String email) { /\* 仅处理通知 \*/ }

}

* 开闭原则

一个软件实体应当对扩展开放，对修改关闭。也就是说在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展，即实现在不修改源代码的情况下改变这个模块的行为。

实践项目应用：在产品目录模块中，过滤产品功能最初只支持按价格过滤。当业务需添加新过滤条件（如按品牌、类别）时，避免修改核心过滤类。

public interface FilterCriteria {

boolean matches(Product product);

}

public class PriceCriteria implements FilterCriteria {

private double maxPrice;

public PriceCriteria(double maxPrice) { this.maxPrice = maxPrice; }

public boolean matches(Product product) {

return product.getPrice() <= maxPrice;

}

}

public class BrandCriteria implements FilterCriteria {

private String brand;

public BrandCriteria(String brand) { this.brand = brand; }

@Override

public boolean matches(Product product) {

return product.getBrand().equals(brand);

}

}

public class ProductFilter {

public List<Product> filter(List<Product> products, FilterCriteria criteria) {

return products.stream().filter(criteria::matches).collect(Collectors.toList());

}

}

* 德（迪）米特法则

(1) 不要和“陌生人”说话。

(2) 只与你的直接朋友通信。

(3) 每一个软件单位对其他的单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位。

一个软件实体对其他实体的引用越少越好，或者说如果两个类不必彼此直接通信，那么这两个类就不应当发生直接的相互作用，而是通过引入一个第三者发生间接交互。

实践项目应用：在订单处理模块，Order类需要访问客户地址或支付状态，但不直接耦合到Customer或Payment类的内部。

public class Customer {

private Address address;

public String getFullAddress() { // 提供封装方法

return address.getFullAddress();

}

}

public class Payment {

private PaymentProcessor processor;

public void executePayment() { // 封装支付执行

processor.execute();

}

}

public class Order {

private Customer customer;

private Payment payment;

public String getShippingAddress() {

return customer.getFullAddress(); // 只与Customer交互

}

public void processPayment() {

payment.executePayment(); // 只与Payment交互

}

}

* 依赖倒转原则

高层模块不应该依赖低层模块，它们都应该依赖抽象。抽象不应该依赖于细节，细节应该依赖于抽象。也可以说要针对接口编程，不要针对实现编程。

实践项目应用：在库存管理模块，高层InventoryService（库存服务）不直接依赖具体数据库操作，而是依赖抽象接口。

public interface InventoryRepository {

void save(InventoryItem item);

}

public class MySQLInventoryRepository implements InventoryRepository {

public void save(InventoryItem item) { /\* MySQL实现 \*/ }

}

public class MongoDBInventoryRepository implements InventoryRepository {

public void save(InventoryItem item) { /\* MongoDB实现 \*/ }

}

public class InventoryService {

private InventoryRepository repository; // 依赖抽象

public InventoryService(InventoryRepository repository) { // 依赖注入

this.repository = repository;

}

public void updateStock(InventoryItem item) {

repository.save(item);

}

}

* 合成复用原则

尽量使用对象组合，而不是继承来达到复用的目的。就是指在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用其已有功能的目的。

实践项目应用：在购物车模块，折扣功能不要通过继承基类购物车实现，而是组合折扣策略对象。

public interface DiscountStrategy {

double applyDiscount(double originalPrice);

}

public class PercentageDiscountStrategy implements DiscountStrategy {

private double rate;

public PercentageDiscountStrategy(double rate) { this.rate = rate; }

public double applyDiscount(double originalPrice) {

return originalPrice \* (1 - rate);

}

}

public class FixedDiscountStrategy implements DiscountStrategy {

private double amount;

public FixedDiscountStrategy(double amount) { this.amount = amount; }

@Override

public double applyDiscount(double originalPrice) {

return Math.max(0, originalPrice - amount);

}

}

public class ShoppingCart {

private List<Product> items;

private DiscountStrategy discountStrategy; // 组合策略

public ShoppingCart(DiscountStrategy strategy) {

this.discountStrategy = strategy;

}

public double calculateTotal() {

double subtotal = items.stream().mapToDouble(Product::getPrice).sum();

return discountStrategy.applyDiscount(subtotal);

}

}