Final Review (Fall 2024)

1. Given this scenario:

```
interface PaymentStrategy {
    void processPayment(Order order);
}

class CreditCardPayment implements PaymentStrategy {
    void processPayment(Order order) { /* implementation */ }
}

class PayPalPayment implements PaymentStrategy {
    void processPayment(Order order) { /* implementation */ }
}

class PaymentProcessor {
    private PaymentStrategy strategy;

    void setStrategy(PaymentStrategy strategy) {
        this.strategy = strategy;
    }

    void handlePayment(Order order) {
        strategy.processPayment(order);
    }
}
```

This design most effectively demonstrates:

a. Delegation
b. Aggregation
c. Pure fabrication design
d. Extension without modification

2. A development team is designing a system where:

```
interface WeatherDataSource {
    WeatherData getData();
}

class WeatherService {
    private WeatherDataSource source;

    WeatherService(WeatherDataSource source) {
        this.source = source;
    }

    WeatherData getWeather() {
        return source.getData();
    }
}
```

This approach best represents:

a. Pure fabrication
b. Protected variation
c. Open closed principle
d. None of the above

3. Consider this code structure:

```
interface DataExporter {
   void export(Data data);
}

class PDFExporter implements DataExporter { }
class ExcelExporter implements DataExporter { }
class JSONExporter implements DataExporter { }

class ReportGenerator {
   private DataExporter exporter;

   void setExporter(DataExporter exporter) {
      this.exporter = exporter;
   }
}
```

The primary benefit of this design is:

a. Performance optimization
b. Memory efficiency
c. Future format adaptability
d. Resource consolidation

4. In an e-commerce application, a new class called OrderCoordinator is created with this responsibility:

```
class OrderCoordinator {
   public void processOrder(Order order) {
      paymentService.process(order.getPayment());
      inventoryService.update(order.getItems());
      shippingService.schedule(order);
      notificationService.notify(order.getCustomer());
   }
}
```

This class best represents:

a. Domain modeling
b. Technical organization
c. Business entity
d. Real-world object

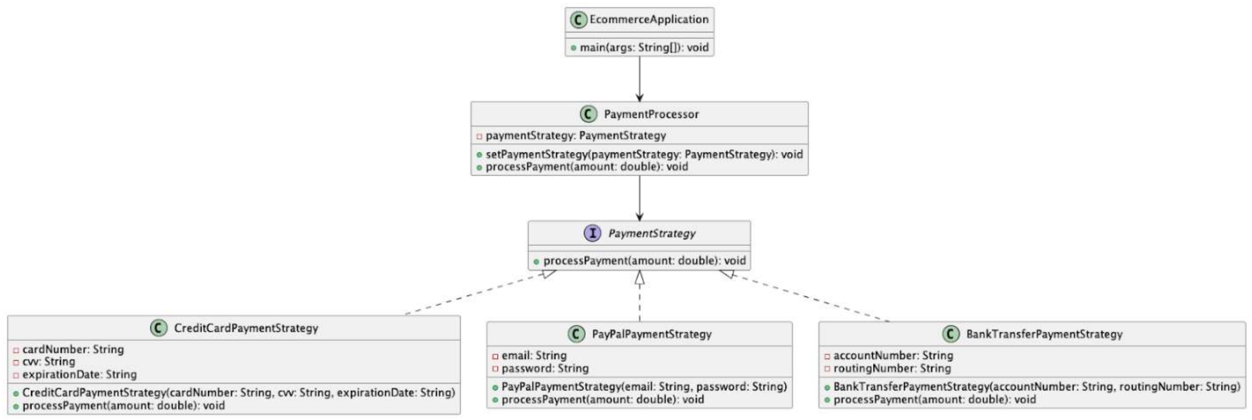
(Pure Fabrication where a class is created for technical purposes rather than representing a real-world entity)

1. You are designing a payment processing system for an e-commerce platform. The platform needs to support the following payment methods: credit card, PayPal, and bank transfer. Each payment method has its own unique implementation for processing payments. For example, PayPal requires an email and password, while Bank transfer requires an account and routing number. Additionally, the platform should be extensible so that new payment methods can be easily added in the future.

a. Mention which design pattern you implemented and provide a brief justification in one paragraph. (3 points).
b. Provide your solution as a UML class diagram depicting the implementation of the payment processing system using the design pattern that best fits the problem. Be as detailed as possible. (3 points)

ANSWER a: Strategy pattern. It is the best fit for implementing a payment processing system because it allows for a flexible and dynamic selection of payment methods while keeping the payment processing logic encapsulated within separate strategy classes. By defining a common interface for all payment strategies, the system achieves abstraction and separation of concerns, enabling easy addition of new payment methods without modifying existing code. This pattern promotes code reusability and maintainability by facilitating interchangeable behavior at runtime, making it ideal for scenarios where different algorithms or strategies may be applied depending on specific requirements or user preferences, such as in the case of processing payments with credit cards, PayPal, or bank transfers.

ANSWER b:


EcommerceApplication
main(args: String[]): void
PaymentProcessor
paymentStrategy: PaymentStrategy
setPaymentStrategy(paymentStrategy: PaymentStrategy): void
processPayment(amount: double): void
PaymentStrategy
processPayment(amount: double): void
CreditCardPaymentStrategy
cardNumber: String
cvv: String
expirationDate: String
CreditCardPaymentStrategy(cardNumber: String, cvv: String, expirationDate: String)
processPayment(amount: double): void
PayPalPaymentStrategy
email: String
password: String
PayPalPaymentStrategy(email: String, password: String)
processPayment(amount: double): void
BankTransferPaymentStrategy
accountNumber: String
routingNumber: String
BankTransferPaymentStrategy(accountNumber: String, routingNumber: String)
processPayment(amount: double): void