UNIVERSIDAD TECNOLÓGICA DE PANAMÁ
FACULTAD DE INGENIERIA DE SISTEMAS COMPUTACIONALES
VICEDECANATO DE INVESTIGACIÓN, POSGRADO Y EXTENSIÓN
MAESTRÍA EN INGENIERÍA DE SOFTWARE
VERANO 2026

Profesor: Danilo Domínguez Pérez, PhD
Email: danilo.dominguez1@utp.ac.pa

Asignatura #1 - Patrones, Antipatrones y Principios de Diseño
Fecha entrega: 23 de febrero de 2026

DESCRIPCIÓN GENERAL

Esta asignación busca afirmar los conocimientos de los diferentes conceptos vistos sobre principios de diseño de software, patrones de diseño, anti-patrones y conceptos generales en diseño de software.

100 puntos.

PREGUNTAS

1. Describe en sus propias palabras las diferentes complejidades de software que define Fred Brooks: Essential Complexity y Accidental Complexity. 10 puntos.

Complejidad escencial y complejidad accidental

2. ¿Cuáles son los atributos de una buena arquitectura? 4 puntos.

Escalabilidad, Mantenibilidad, Desempeño y Seguridad.

3. Nombre 2 restricciones de negocio y 2 restricciones técnicas en proyectos de desarrollo de software. 4 puntos.

Restricciones de negocio: El presupuesto del proyecto, El tiempo establecido para el proyecto. Restricciones técnicas: El lenguaje de programación del proyecto, Herramientas de

gestión del proyecto (Para comunicaciones, taskboards, etc).

4. Mencione 3 síntomas de complejidad según John Ousterhoot (en A Phylosophy of Software Design). Trate de definirlas con sus propias palabras. 6 puntos

- Amplificación de cambio, que indica que si modificamos una parte del código genera muchos errores en todos lados es una señal de que hay mucha dependencia entre los componentes.
- Carga Cognitiva, que es la cantidad de información que un desarrollador debe entender para una tarea, un buen diseño realmente minimiza la cantidad de conocimiento que se requiere para trabajar en el sistema.
- Desconocidos desconocidos (unknown unknows), es cuando simplemente no logra ser evidente lo que se debe mejorar o modificar para algún cambio del sistema ya que cuando diseñamos mal podemos hacer que los desarrolladores pierdan mucho tiempo buscando dónde están las dependencias o dónde hay afectaciones de código. Un buen diseño hace que el sistema sea predecible y hasta “obvio”.

5. Detectar Antipatrones y code smells (dados en clase). Proveer solución.

5.1 Detectar que antipatrón tiene el siguiente código y proveer código con solución para eliminar el antipatrón (10 puntos):

```
public class StudentManagementSystem {

   public void processRegularStudent(String name, int age, String course) {
       // Validate student data
       if (name == null || name.trim().isEmpty()) {
           System.out.println("Error: Name cannot be empty");
           return;
       }
       if (age < 16 || age > 99) {
           System.out.println("Error: Invalid age");
           return;
       }
       if (course == null || course.trim().isEmpty()) {
           System.out.println("Error: Course cannot be empty");
           return;
```

```
        }

        // Calculate student ID
        String studentId = name.substring(0, 3).toUpperCase() + age +
course.substring(0, 2).toUpperCase();

        // Format student data
        String studentInfo = String.format("Regular Student - ID: %s\nName: %s\nAge:
%d\nCourse: %s",
            studentId, name, age, course);

        // Save to database (simulated)
        System.out.println("Saving to database: " + studentInfo);
        System.out.println("Regular student processed successfully");
    }

    public void processExchangeStudent(String name, int age, String course) {
        // Validate student data
        if (name == null || name.trim().isEmpty()) {
            System.out.println("Error: Name cannot be empty");
            return;
        }
        if (age < 16 || age > 99) {
            System.out.println("Error: Invalid age");
            return;
        }
        if (course == null || course.trim().isEmpty()) {
            System.out.println("Error: Course cannot be empty");
            return;
        }

        // Calculate student ID (exactly the same logic)
        String studentId = name.substring(0, 3).toUpperCase() + age +
course.substring(0, 2).toUpperCase();

        // Format student data (almost identical)
        String studentInfo = String.format("Exchange Student - ID: %s\nName: %s\nAge:
%d\nCourse: %s",
            studentId, name, age, course);
```

```
        // Save to database (simulated)
        System.out.println("Saving to database: " + studentInfo);
        System.out.println("Exchange student processed successfully");
    }

    public static void main(String[] args) {
        StudentManagementSystem system = new StudentManagementSystem();

        // Test with regular student
        system.processRegularStudent("John Smith", 20, "Computer Science");

        // Test with exchange student
        system.processExchangeStudent("Maria Garcia", 22, "Physics");
    }
}
```

5.2 Detectar que antipatrón tiene el siguiente código y proveer código con solución para eliminar el antipatrón (10 puntos):

```
public class Order {
    private String streetAddress;
    private String city;
    private String state;
    private String zipCode;
    private String country;
    private int quantity;
    private double price;

    public Order(String streetAddress, String city, String state, String zipCode,
String country, int quantity, double price) {
        this.streetAddress = streetAddress;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
        this.country = country;
        this.quantity = quantity;
        this.price = price;
```

```
    }

    public void updateShippingAddress(String streetAddress, String city, String
state, String zipCode, String country) {
        this.streetAddress = streetAddress;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
        this.country = country;
    }
}

public class Customer {
    private String name;
    private String email;
    private String streetAddress;
    private String city;
    private String state;
    private String zipCode;
    private String country;

    public Customer(String name, String email, String streetAddress, String city,
String state, String zipCode, String country) {
        this.name = name;
        this.email = email;
        this.streetAddress = streetAddress;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
        this.country = country;
    }

    public void updateShippingAddress(String streetAddress, String city, String
state, String zipCode, String country) {
        this.streetAddress = streetAddress;
        this.city = city;
        this.state = state;
        this.zipCode = zipCode;
        this.country = country;
    }
```

```
}
```

5.3 Detectar que antipatrón tiene el siguiente código y proveer código con solución para eliminar el antipatrón (10 puntos):

```
class Email {
   private String email;

   public Email(String email) {
       this.email = email;
   }

   public String getEmail() {
       return email;
   }
}

class EmailValidator {
   public boolean isValidEmail(String email) {
       return email != null && email.contains("@") && email.contains(".");
   }
}
```

6. Principios SOLID. Detectar el o los principios SOLID que se pueden aplicar para mejorar el diseño de código.

6.1. Qué principio SOLID se puede aplicar para mejorar el diseño de este código. Aplique el principio y genere un código con mejor diseño (10 puntos).

```
class PaymentDetails {
   private final double amount;
   private final String currency;
   private final String cardNumber;
   private final String cvv;

   public PaymentDetails(double amount, String currency, String cardNumber, String
```

```
cvv) {
        this.amount = amount;
        this.currency = currency;
        this.cardNumber = cardNumber;
        this.cvv = cvv;
    }

    // Getters
    public double getAmount() { return amount; }
    public String getCurrency() { return currency; }
    public String getCardNumber() { return cardNumber; }
    public String getCvv() { return cvv; }
}

interface PaymentGateway {
     Public boolean processPayment(double amount, String…)
}

class StripePaymentGateway implements PaymentGateway {
    public boolean processStripePayment(double amount, String currency, String
cardNumber, String cvv) {
        System.out.println("Processing payment through Stripe");
        System.out.println("Amount: " + amount + " " + currency);
        return true;
    }
}

class PayPalPaymentGateway implements PaymentGateway {
    public boolean processPayPalPayment(double amount, String currency, String
cardNumber, String cvv) {
        System.out.println("Processing payment through PayPal");
        System.out.println("Amount: " + amount + " " + currency);
        return true;
    }
}

class PaymentProcessor {
    private final StripePaymentGateway stripeGateway;
    private final PayPalPaymentGateway paypalGateway;
```

```
    public PaymentProcessor() {
        this.stripeGateway = new StripePaymentGateway();
        this.paypalGateway = new PayPalPaymentGateway();
    }

    public boolean processOrder(String paymentMethod, double amount, String currency,
String cardNumber, String cvv) {
        if ("STRIPE".equals(paymentMethod)) {
            return stripeGateway.processStripePayment(amount, currency, cardNumber,
cvv);
        } else if ("PAYPAL".equals(paymentMethod)) {
            return paypalGateway.processPayPalPayment(amount, currency, cardNumber,
cvv);
        } else {
            throw new IllegalArgumentException("Unsupported payment method: " +
paymentMethod);
        }
    }
}

// Example usage
public class Solid1 {
    public static void main(String[] args) {
        PaymentProcessor processor = new PaymentProcessor();

        // Process payment with Stripe
        processor.processOrder("STRIPE", 99.99, "USD", "4111111111111111", "123");

        // Process payment with PayPal
        processor.processOrder("PAYPAL", 149.99, "EUR", "4111111111111111", "123");
    }
}
```

6.2. Qué principio SOLID se puede aplicar para mejorar el diseño de este código. Aplique el principio y genere un código con mejor diseño (10 puntos).

```
interface Document {
    void scan();
    void print();
    void fax();
```

```
    void photocopy();
    void staple();
    void encrypt();
    void sign();
}

Interface Scanner {}
Interface Printer {}
Interface ScannerPrinter implements Scanner, Printer {
    Void photocopy();
}

// Professional printer has to implement methods it doesn't support
class ProfessionalPrinter implements ScannerPrinter {
    @Override
    public void scan() {
        System.out.println("Scanning document at 300 DPI");
    }

    @Override
    public void print() {
        System.out.println("Printing document in high quality");
    }

    @Override
    public void fax() {
        // Don't support fax
        throw new UnsupportedOperationException("This printer doesn't support fax");
    }

    @Override
    public void photocopy() {
        System.out.println("Making a photocopy");
    }

    @Override
    public void staple() {
        // Don't support stapling
        throw new UnsupportedOperationException("This printer doesn't have a
stapler");
```

```
    }

    @Override
    public void encrypt() {
        // Don't support encryption
        throw new UnsupportedOperationException("This printer doesn't support
encryption");
    }

    @Override
    public void sign() {
        // Don't support digital signing
        throw new UnsupportedOperationException("This printer doesn't support digital
signatures");
    }
}
```

7. Nombrar 3 patrones de diseño estructurales (Structural design patterns) y 3 patrones de diseño creacionales (Creational design patterns). 6 puntos.

Patrones Estructurales: Decorator, State, Memento.
Patrones Creacionales: Singleton, Factory Method, Flyweight.

8. Aplicar patrones de diseño. 10 puntos.
   Antecedentes:
   Estás desarrollando un sistema de personalización de personajes para un juego de rol en el que los jugadores pueden equipar a sus personajes con diferentes objetos y encantamientos. Cada objeto o encantamiento afecta las estadísticas del personaje (ataque, defensa, velocidad) y la apariencia.
   Requisitos:
   1. Implementa las siguientes clases de personajes base:
      - Guerrero (estadísticas base: ataque=20, defensa=15, velocidad=10)
      - Mago (estadísticas base: ataque=15, defensa=10, velocidad=12)
      - Pícaro (estadísticas base: ataque=18, defensa=8, velocidad=20)
   2. Implementa los siguientes decoradores de equipo:

- Espada legendaria (+15 ataque, -2 velocidad)
- Armadura de escamas de dragón (+20 defensa, -5 velocidad)
- Botas veloces (+8 velocidad)
- Amuleto mágico (+10 ataque, +5 defensa)
- Capa invisible (+5 defensa, +10 velocidad)

3. Cada personaje debe proporcionar:
   - getStats() -> devuelve un objeto de estadísticas con los valores actuales
   - getDescription() -> devuelve la descripción del personaje con el equipo

Código Base

```
class Stats {
    private int attack;
    private int defense;
    private int speed;

    public Stats(int attack, int defense, int speed) {
        this.attack = attack;
        this.defense = defense;
        this.speed = speed;
    }

    // Getters and modifier methods
    public int getAttack() { return attack; }
    public int getDefense() { return defense; }
    public int getSpeed() { return speed; }

    public void addAttack(int value) { this.attack += value; }
    public void addDefense(int value) { this.defense += value; }
    public void addSpeed(int value) { this.speed += value; }

    @Override
    public String toString() {
        return String.format("ATK: %d, DEF: %d, SPD: %d", attack, defense,
speed);
    }
}

// Base Component Interface
interface GameCharacter {
```

```
    Stats getStats();
    String getDescription();
}

// Concrete Components (Base Characters)
class Warrior implements GameCharacter {
    @Override
    public Stats getStats() {
        return new Stats(20, 15, 10);
    }

    @Override
    public String getDescription() {
        return "Warrior";
    }
}

class Mage implements GameCharacter {
    @Override
    public Stats getStats() {
        return new Stats(15, 10, 12);
    }

    @Override
    public String getDescription() {
        return "Mage";
    }
}

class Rogue implements GameCharacter {
    @Override
    public Stats getStats() {
        return new Stats(18, 8, 20);
    }

    @Override
    public String getDescription() {
        return "Rogue";
    }
}
```

abstract class CharacterDecorator implements GameCharacter {

```
   Protected GameCharacter character;

   Public CharacterDecorator(GameCharacter character) {
      This.character = character;
   }
}

Public class LegendarySword extends CharacterDecorator {

   Public LegendarySword(GameCharacter character) : super(character)

   @Override
   Public Stats getStats() {
      Return new Stats(character.attack + 15, character.defense,
character.speed - 2);
   }

   @Override
   Public String getDescription() {
     Return character.description + " with legendary sword";
   }
}
```

9. Aplicar patrones de diseño. 10 puntos.
   Estás desarrollando un editor de texto que debe manejar documentos grandes (millones de caracteres) donde cada carácter puede tener diferentes propiedades de formato (fuente, tamaño, color, estilo).
   Almacenar la información de formato para cada carácter individualmente consumiría demasiada memoria.

   Requisitos:

   1. Cada carácter en el documento puede tener las siguientes propiedades:
      - Fuente (Arial, Times New Roman, Courier)
      - Tamaño (8, 10, 12, 14, 16)
      - Color (Negro, Rojo, Azul, Verde)
      - Estilo (Negrita, Cursiva, Normal)

   2. El sistema debe:
      - Minimizar el uso de memoria para documentos con formato repetido
      - Soportar actualizaciones eficientes del formato de caracteres
      - Permitir la iteración a través del documento manteniendo el formato
      - Soportar estadísticas básicas de texto (conteo de caracteres con formato específico)

   Utiliza el patrón Flyweight para implementar esta solución de manera eficiente.
   Implementar `CharacterFormattingFactory, FormattedCharacter, .`

   Código Base:

```
interface CharacterFormat {
   void applyFormat(char character, int position);
}

class CharacterFormatting implements CharacterFormat {
   private final String font;
   private final int size;
   private final String color;
   private final String style;

   public CharacterFormatting(String font, int size, String color, String
style) {
       this.font = font;
```

```
        this.size = size;
        this.color = color;
        this.style = style;
    }

    @Override
    public void applyFormat(char character, int position) {
        System.out.printf("Carácter '%c' en posición %d con formato: %s, %dpt,
%s, %s%n",
                          character, position, font, size, color, style);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        CharacterFormatting that = (CharacterFormatting) o;
        return size == that.size &&
               font.equals(that.font) &&
               color.equals(that.color) &&
               style.equals(that.style);
    }

    @Override
    public int hashCode() {
        return Objects.hash(font, size, color, style);
    }
}


class Document {
    private List<FormattedCharacter> characters = new ArrayList<>();
    private CharacterFormattingFactory factory;

    public Document(CharacterFormattingFactory factory) {
        this.factory = factory;
    }

    public void insertText(String text, String font, int size, String color,
String style) {
```

```
        CharacterFormat format = factory.getFormatting(font, size, color,
style);
        int position = characters.size();

        for (char c : text.toCharArray()) {
            characters.add(new FormattedCharacter(c, format, position++));
        }
    }

    public void display() {
        for (FormattedCharacter fc : characters) {
            fc.display();
        }
    }
}

// Clase de prueba
public class DesignPatternFlyweight {
    public static void main(String[] args) {
        CharacterFormattingFactory factory = new CharacterFormattingFactory();
        Document doc = new Document(factory);

        // Caso de prueba 1: Formato básico
        doc.insertText("Hello ", "Arial", 12, "Blue", "Bold");
        doc.insertText("World", "Times New Roman", 14, "Red", "Italic");

        System.out.println("Contenido del documento:");
        doc.display();

        System.out.println("\nObjetos de formato únicos creados: " +
factory.getCacheSize());
        // Salida esperada: 2 (no 11, demostrando el ahorro de memoria)
    }
}
```