

Fundamentos Avanzados de Diseño Orientado a Objetos

Introducción

El diseño orientado a objetos representa uno de los paradigmas más poderosos y extendidos en el desarrollo de software moderno. Esta unidad profundiza en los fundamentos avanzados que permiten crear sistemas robustos, mantenibles y escalables a través de principios sólidos de diseño. En este nivel avanzado, no nos limitaremos a la comprensión teórica de estos conceptos, sino que analizaremos su aplicación crítica en escenarios complejos de desarrollo de software.

A lo largo de esta unidad, exploraremos desde los pilares fundamentales de la abstracción y el encapsulamiento hasta los deseables principios SOLID y GRASP, pasando por técnicas avanzadas de modelado UML. El enfoque se centrará en desarrollar la capacidad para evaluar críticamente diseños existentes y proponer mejoras estructurales fundamentadas.

1.1 Abstracción y encapsulamiento

Abstracción Avanzada

La abstracción es la capacidad de identificar las características y comportamientos esenciales de un objeto, ignorando los detalles irrelevantes. En diseño avanzado, la abstracción va más allá de la simple identificación de clases y objetos.

Niveles de abstracción

1. **Abstracción de datos** : Es el nivel básico donde definimos estructuras de datos y sus operaciones asociadas.
2. **Abstracción procesal** : Enfoca en cómo se realizan las operaciones, ocultando los detalles de implementación.
3. **Abstracción conceptual** : Es el nivel más cómodo, donde se trabaja con modelos mentales completos de un dominio de problema.

Técnicas Avanzadas de Abstracción

- **Abstracción por especialización** : Refinamiento progresivo de conceptos generales a específicos.
- **Abstracción por generalización** : Identificación de características comunes para formar abstracciones de nivel superior.
- **Abstracción por análisis** : División de sistemas complejos en subsistemas más manejables.

Jerarquías de Abstracción

Las jerarquías de abstracción permiten manejar la complejidad a través de diferentes niveles de detalle. Un sistema bien diseñado presenta múltiples capas de abstracción, donde cada capa proporciona una vista coherente que oculta los detalles de las capas inferiores.

Vista de Usuario (UI) → Vista de Negocio → Vista de Dominio → Vista de Infraestructura

Encapsulamiento avanzado

El encapsulamiento es el mecanismo que combina datos y comportamientos en una única unidad y restringe el acceso directo a algunos de los componentes del objeto.

Niveles de encapsulamiento

1. **Encapsulamiento de implementación** : Oculta los detalles internos de una clase.
2. **Encapsulamiento de interfaz** : Defina contratos claros para la interacción entre componentes.
3. **Encapsulamiento de tipo** : Proporciona abstracciones que permiten el polimorfismo.

Patrones Avanzados de Encapsulamiento

- **Tell, Don't Ask** : Principio que sugiere dar órdenes a los objetos en lugar de solicitar información para tomar decisiones.
- **Principio de Conocimiento Mínimo (Ley de Demeter)** : Un objeto debe tener conocimiento limitado sobre otros objetos.
- **Encapsulamiento de Variaciones** : Aislar aspectos que son propensos a cambiar.

Ejemplo de Encapsulamiento Avanzado

Java

```
// Enfoque básico con violación de encapsulamiento
class UserBasic {
    public String name;
    public String email;
    public String password;
}

// Enfoque avanzado con encapsulamiento adecuado
class User {
    private String name;
    private Email email; // Tipo de valor encapsulado
    private Password password; // Tipo de valor encapsulado

    public User(String name, Email email, Password password) {
```

```
        this.name = name;
        this.email = email;
        this.password = password;
    }

    public void changeEmail(Email newEmail, AuthenticationService auth) {
        if (auth.isAuthenticated(this)) {
            this.email = newEmail;
            notifyEmailChange();
        } else {
            throw new SecurityException("Unauthorized email change attempt");
        }
    }

    private void notifyEmailChange() {
        // Lógica para notificar cambio de email
    }

    // No hay getters para password, implementando encapsulamiento estricto
    public String getName() {
        return name;
    }

    public String getEmailAddress() {
        return email.getAddress(); // Solo exponemos la dirección, no el objeto
        completo
    }
}

// Tipos de valor encapsulados
class Email {
    private String address;

    public Email(String address) {
        if (!isValid(address)) {
            throw new IllegalArgumentException("Invalid email format");
        }
        this.address = address;
    }
}
```

```

    }

    private boolean isValid(String address) {
        // Validación de formato de email
        return address.matches("^[\\w-\\.]+@[\\w-\\.]+\\.[\\w-]{2,4}$");
    }

    public String getAddress() {
        return address;
    }
}

class Password {
    private String hashedValue;

    public Password(String plainText) {
        this.hashedValue = hashPassword(plainText);
    }

    private String hashPassword(String plainText) {
        // Algoritmo de hash seguro
        return ""; // Implementación del hash
    }

    public boolean matches(String plainText) {
        return this.hashedValue.equals(hashPassword(plainText));
    }
}

```

Este ejemplo muestra un encapsulamiento avanzado donde:

1. Se utilizan tipos de valor para representar conceptos del dominio (correo electrónico, contraseña)
2. Se implementa validación de datos dentro de los constructores.
3. No se exponen getters para información sensible como contraseñas
4. Se controla el acceso operaciones a críticas mediante servicios de autenticación
5. Se aplica el principio Tell-Don't-Ask con métodos como changeEmail()

Beneficios estratégicos

La aplicación avanzada de abstracción y encapsulamiento proporciona:

1. **Mantenibilidad superior** : Los cambios en la implementación quedan contenidos.
2. **Reutilización estratégica** : Los componentes bien encapsulados son más fáciles de reutilizar.
3. **Seguridad mejorada** : Control preciso sobre el acceso a datos sensibles.
4. **Evolución flexible** : La capacidad de modificar internamente componentes sin afectar a sus clientes.

1.2 Acoplamiento y cohesión

Acoplamiento Avanzado

El acoplamiento mide el grado de interdependencia entre módulos. Un bajo acoplamiento permite que los componentes sean más independientes, facilitando su mantenimiento y reutilización.

Tipos de Acoplamiento (de menor a mayor)

1. **Acoplamiento de datos** : Módulos que se comunican pasando datos simples.
2. **Acoplamiento de marca** : Módulos que comparten estructuras de datos complejos.
3. **Acoplamiento de control** : Un módulo controla el flujo de otro enviando banderas o información de control.
4. **Acoplamiento externo** : Módulos que dependen de protocolos, dispositivos o formatos externos.
5. **Acoplamiento común** : Módulos que comparten datos globales.
6. **Acoplamiento de contenido** : Un módulo modifica directamente el estado interno de otro.

Estrategias Avanzadas para Reducir el Acoplamiento

- **Inversión de Dependencias** : Dependencia de abstracciones, no implementaciones concretas.
- **Inyección de Dependencias** : Proveer las dependencias a un objeto en lugar de que éste las cree.
- **Eventos y Observadores** : Comunicación indirecta entre componentes.
- **Adaptadores y Fachadas** : Capas de dirección para aislar sistemas.

Métricas de Acoplamiento

- **Acoplamiento Aferente (Ca)** : Número de clases externas que dependen de clases dentro del módulo.
- **Acoplamiento Eferente (Ce)** : Número de clases dentro del módulo que dependen de clases externas.
- **Inestabilidad (I)** : $Ce / (Ca + Ce)$, donde 0 significa estable y 1 inestable.

Cohesión Avanzada

La cohesión mide cuán relacionados están los elementos dentro de un módulo. Alta cohesión significa que los elementos de un módulo están fuertemente relacionados y enfocados en una única responsabilidad.

Tipos de Cohesión (de menor a mayor)

1. **Cohesión coincidental** : Elementos agrupados sin relación lógica.
2. **Cohesión lógica** : Elementos que realizan tareas de categoría similar.
3. **Cohesión temporal** : Elementos que se ejecutan en el mismo momento.
4. **Cohesión procesal** : Elementos que participan en una secuencia de pasos.
5. **Cohesión comunicacional** : Elementos que operan sobre los mismos datos.
6. **Cohesión secuencial** : La salida de un elemento es entrada para otro.
7. **Cohesión funcional** : Todos los elementos contribuyen a una única tarea bien definida.

Técnicas para Mejorar la Cohesión

- **Principio de Responsabilidad Única** : Una clase debe tener solo una razón para cambiar.
- **Extracción de Clases** : Mover funcionalidades relacionadas a nuevas clases especializadas.
- **Composición sobre Herencia** : Usar composición para combinar comportamientos.
- **Clases Cohesivas por Conceptos del Dominio** : Modelar clases según entidades del dominio del problema.

Interacción entre Acoplamiento y Cohesión

El equilibrio óptimo de diseño busca **minimizar el acoplamiento** mientras **maximiza la cohesión** . Estas propiedades están interrelacionadas:

- Al aumentar la cohesión de los módulos, a menudo se reduce su acoplamiento.
- Al reducir el acoplamiento, se facilita incrementar la cohesión de los módulos individuales.

Ejemplo de análisis crítico

Java

```
// Diseño con ALTO acoplamiento y BAJA cohesión

class OrderProcessor {
    private Database db;
    private EmailService emailService;
    private PaymentGateway paymentGateway;
    private InventorySystem inventory;
    private TaxCalculator taxCalculator;

    public void processOrder(Order order) {
```

```

    // Verificar inventario
    boolean inStock = inventory.checkStock(order.getItems());
    if (!inStock) {
        throw new OutOfStockException();
    }

    // Calcular impuestos
    double tax = taxCalculator.calculateTax(order);
    order.setTax(tax);

    // Procesar pago
    PaymentResult result = paymentGateway.processPayment(order.getCustomer(),
    order.getTotalWithTax());
    if (!result.isSuccessful()) {
        throw new PaymentFailedException();
    }

    // Actualizar inventario
    inventory.updateStock(order.getItems());

    // Guardar orden
    db.save(order);

    // Enviar confirmación
    emailService.sendOrderConfirmation(order);
}
}

```

Este diseño presenta problemas de:

- **Alto acoplamiento** : Depende directamente de múltiples sistemas externos.
- **Baja cohesión** : Realiza múltiples tareas no estrechamente relacionadas.
- **Dificultad para pruebas** : Difícil de probar sin integración completa con sistemas externos.

Java

```

// Diseño mejorado con BAJO acoplamiento y ALTA cohesión
interface StockChecker {
    boolean checkAvailability(List<OrderItem> items);
}

```

```
    void updateInventory(List<OrderItem> items);
}

interface PaymentProcessor {
    PaymentResult process(Customer customer, Money amount);
}

interface OrderRepository {
    void save(Order order);
}

interface OrderNotifier {
    void notifyOrderConfirmation(Order order);
}

class OrderProcessor {
    private final StockChecker stockChecker;
    private final TaxCalculator taxCalculator;
    private final PaymentProcessor paymentProcessor;
    private final OrderRepository orderRepository;
    private final OrderNotifier orderNotifier;

    // Inyección de dependencias
    public OrderProcessor(StockChecker stockChecker,
                          TaxCalculator taxCalculator,
                          PaymentProcessor paymentProcessor,
                          OrderRepository orderRepository,
                          OrderNotifier orderNotifier) {
        this.stockChecker = stockChecker;
        this.taxCalculator = taxCalculator;
        this.paymentProcessor = paymentProcessor;
        this.orderRepository = orderRepository;
        this.orderNotifier = orderNotifier;
    }

    public void processOrder(Order order) {
        ensureItemsAvailable(order);
        calculateAndApplyTax(order);
    }
}
```

```
    processPayment(order);
    updateInventory(order);
    saveOrder(order);
    notifyCustomer(order);
}

private void ensureItemsAvailable(Order order) {
    if (!stockChecker.checkAvailability(order.getItems())) {
        throw new OutOfStockException();
    }
}

private void calculateAndApplyTax(Order order) {
    Money tax = taxCalculator.calculateTax(order);
    order.setTax(tax);
}

private void processPayment(Order order) {
    PaymentResult result = paymentProcessor.process(
        order.getCustomer(), order.getTotalWithTax());
    if (!result.isSuccessful()) {
        throw new PaymentFailedException(result.getErrorMessage());
    }
}

private void updateInventory(Order order) {
    stockChecker.updateInventory(order.getItems());
}

private void saveOrder(Order order) {
    orderRepository.save(order);
}

private void notifyCustomer(Order order) {
    orderNotifier.notifyOrderConfirmation(order);
}
}
```

Este rediseño ofrece:

- **Bajo acoplamiento** : Depende de abstracciones en lugar de implementaciones concretas.
- **Alta cohesión** : Cada método tiene una única responsabilidad clara.
- **Facilidad de pruebas** : Posibilidad de crear simulacros o stubs para los componentes.
- **Flexibilidad** : Fácil adaptación a diferentes implementaciones de infraestructura.

1.3 Delegación y Responsabilidad

Delegación Avanzada

La delegación es un mecanismo mediante el cual un objeto transfiere la responsabilidad de una tarea específica a otro objeto. Es una alternativa a la herencia que promueve la composición sobre la jerarquía.

Tipos de Delegación

1. **Delegación Explícita** : El objeto delegante invoca explícitamente métodos en el objeto delegado.
2. **Delegación Implícita** : El comportamiento se delega automáticamente, como en los patrones de diseño Proxy o Decorator.
3. **Delegación Dinámica** : La delegación puede cambiar en tiempo de ejecución, como en el patrón State.

Patrones de la Delegación

- **Composición sobre Herencia** : Preferir la inclusión de objetos a clases extendidas.
- **Cadena de Responsabilidad** : Delegación en cadena hasta encontrar el objeto que pueda manejar la solicitud.
- **Mediador** : Centralizar la comunicación entre componentes mediante delegación.

Ventajas Estratégicas de la Delegación

1. **Evite la "explosión de clases"** que puede ocurrir con jerarquías de herencia complejas.
2. **Permite cambios de comportamiento en tiempo de ejecución** sin modificar la estructura de clases.
3. **Facilita la combinación de comportamientos** sin los problemas del "diamante de la muerte" en herencia múltiple.
4. **Mejora la testeabilidad** al poder sustituir componentes delegados con simulacros o stubs.

Responsabilidad en Diseño OO

La responsabilidad se refiere a las obligaciones que tiene un objeto en el sistema. Una clase bien diseñada debe tener responsabilidades coherentes y claramente definidas.

Asignación Estratégica de Responsabilidades

- **Basada en Conocimiento** : "¿Quién posee la información necesaria?"
- **Basada en Comportamiento** : "¿Quién debe realizar esta acción?"
- **Basada en Dominio** : Asignar responsabilidades según los conceptos del dominio del problema.

Indicadores de Mala Asignación de Responsabilidades

1. **Clases Diós** : Objetos que conocen o hacen demasiado.
2. **Alta Frecuencia de Cambio** : Clases que cambian por múltiples razones no relacionadas.
3. **Métodos Utilitarios Abundantes** : Señal de que la responsabilidad no está en la clase correcta.
4. **Acoplamiento Excesivo** : Necesidad de conocer muchos otros objetos para completar sus tareas.

Ejemplo Avanzado de Delegación y Responsabilidad

Java

```
// Sistema de generación de reportes con delegación avanzada

// Interfaz para estrategias de formato
interface ReportFormatter {
    String format(ReportData data);
}

// Implementaciones concretas
class HTMLReportFormatter implements ReportFormatter {
    @Override
    public String format(ReportData data) {
        StringBuilder html = new StringBuilder("<html><body>");
        html.append("<h1>").append(data.getTitle()).append("</h1>");
        // Implementación específica para HTML
        html.append("</body></html>");
        return html.toString();
    }
}

class PDFReportFormatter implements ReportFormatter {
    @Override
    public String format(ReportData data) {
```

```
// Implementación específica para PDF
    return "PDF content...";
}

}

// Interfaz para estrategias de distribución
interface ReportDistributor {
    void distribute(String formattedReport, List<String> recipients);
}

// Implementaciones concretas
class EmailDistributor implements ReportDistributor {
    private EmailService emailService;

    public EmailDistributor(EmailService emailService) {
        this.emailService = emailService;
    }

    @Override
    public void distribute(String formattedReport, List<String> recipients) {
        recipients.forEach(recipient ->
            emailService.sendEmail(recipient, "Report", formattedReport)
        );
    }
}

class FileSystemDistributor implements ReportDistributor {
    private String basePath;

    public FileSystemDistributor(String basePath) {
        this.basePath = basePath;
    }

    @Override
    public void distribute(String formattedReport, List<String> recipients) {
        // Guarda el reporte en archivos
    }
}
```

```
// Clase que coordina el proceso delegando responsabilidades
class ReportGenerator {
    private final ReportDataCollector dataCollector;
    private final ReportFormatter formatter;
    private final ReportDistributor distributor;

    public ReportGenerator(
        ReportDataCollector dataCollector,
        ReportFormatter formatter,
        ReportDistributor distributor) {
        this.dataCollector = dataCollector;
        this.formatter = formatter;
        this.distributor = distributor;
    }

    public void generateAndDistributeReport(
        ReportType type,
        Date startDate,
        Date endDate,
        List<String> recipients) {

        // Delega la recolección de datos
        ReportData data = dataCollector.collectData(type, startDate, endDate);

        // Delega el formateo
        String formattedReport = formatter.format(data);

        // Delega la distribución
        distributor.distribute(formattedReport, recipients);
    }
}

// Cliente que configura y usa el sistema
class ReportingSystem {
    public void createMonthlyReport() {
        // Configuración de componentes
        ReportDataCollector collector = new DatabaseReportCollector(dbConnection);
```

```

        ReportFormatter formatter = new HTMLReportFormatter();
        ReportDistributor distributor = new EmailDistributor(emailService);

        // Creación del generador con delegación
        ReportGenerator generator = new ReportGenerator(collector, formatter,
distributor);

        // Uso del generador
        generator.generateAndDistributeReport(
            ReportType.MONTHLY_SALES,
            getFirstDayOfMonth(),
            getLastDayOfMonth(),
            getRecipientList()
        );
    }
}

```

Este ejemplo muestra:

1. **Delegación limpia** : ReportGenerator delega tareas específicas a componentes especializados.
2. **Asignación clara de responsabilidades** : Cada clase tiene un propósito único y coherente.
3. **Flexibilidad mediante composición** : El comportamiento puede configurarse en el tiempo de ejecución.
4. **Bajo acoplamiento** : Los componentes interactúan a través de interfaces abstractas.
5. **Alta cohesión** : Cada clase se enfoca en una única responsabilidad.

1.4 Modelado en UML

El Lenguaje Unificado de Modelado (UML) es un lenguaje visual estandarizado para la especificación, construcción y documentación de sistemas de software. A nivel avanzado, UML no se utiliza solo para documentar, sino como herramienta activa en el proceso de diseño.

Diagramas Estructurales Avanzados

Diagramas de clases avanzadas

Los diagramas de clases avanzadas van más allá de la simple representación de atributos y métodos, incorporando:

- **Esterotipos personalizados** : Extensiones del lenguaje UML para representar conceptos específicos del dominio.

- **Restricciones OCL** (Object Constraint Language): Especificaciones formales que definen reglas de negocio.
- **Plantillas y parametrización de tipos** : Representación de clases genéricas.
- **Interfaces compuestas** : Interfaces que extienden múltiples interfaces.

Mostrar imagen

Diagramas de componentes detallados

Los diagramas de componentes a nivel avanzado muestran:

- **Puertos especificados** : Puntos de interacción bien definidos entre componentes.
- **Interfaces requeridas y proporcionadas** : Especificación clara de dependencias.
- **Subsistemas anidados** : Representación de estructuras jerárquicas complejas.
- **Artefactos de implementación** : Asociación entre componentes lógicos y físicos.

Mostrar imagen

Diagramas de Paquetes con Dependencias

Los diagramas de paquetes avanzados incluyen:

- **Análisis de métricas de acoplamiento** : Visualización de Ca, Ce e Inestabilidad.
- **Capas arquitectónicas** : Estructuración en capas con reglas de dependencia.
- **Espacios de nombres anidados** : Organización jerárquica del espacio de nombres.
- **Importaciones y exportaciones** : Control explícito de visibilidad entre paquetes.

Diagramas de Comportamiento Avanzados

Diagramas de secuencia con fragmentos

Los diagramas de secuencia avanzados utilizan:

- **Fragmentos combinados** : alt, opt, loop, par, region, neg, afirmar, etc.
- **Referencias a interacciones** : Modularización de escenarios complejos.
- **Ocurrencias de ejecución** : Representación detallada del tiempo de vida de métodos.
- **Mensajes perdidos y encontrados** : Modelado de comunicación asíncrona.

Mostrar imagen

Diagramas de Estado con Regiones

Los diagramas de estado avanzados incluyen:

- **Estados compuestos** : Estados que contienen subestados.
- **Regiones ortogonales** : Paralelismo dentro de un objeto.
- **Pseudestados complejos** : Puntos de unión, puntos de elección, puntos de historia.
- **Máquinas de estado jerárquicas** : Modelado de comportamiento a múltiples niveles.

Diagramas de actividad con particiones

Los diagramas de actividad avanzada presentan:

- **Particiones múltiples** : Organización por actor, subsistema o capa.
- **Flujos de excepción** : Modelado comprensivo de escenarios de error.
- **Nodos de expansión** : Representación de procesamiento iterativo o paralelo.
- **Señales enviadas y recibidas** : Modelado de comunicación asíncrona entre procesos.

Modelado Avanzado Orientado al Dominio

El modelado UML a nivel avanzado integra conceptos de Diseño Dirigido por el Dominio (DDD):

Modelado de Contextos Delimitados

- Representación limpia de límites entre diferentes contextos de dominio.
- Mapeo de relaciones entre contextos: conformista, anticorrupción, núcleo compartido.

Modelado de Agregados y Entidades

- Identificación clara de agregados y sus raíces.
- Especificación de invariantes y reglas de integridad a nivel de agregado.
- Distinción visual entre entidades y objetos de valor.

Ejemplo de Modelado Avanzado: Sistema de Gestión de Pedidos

```
@startuml

!define ENTITY class
!define VALUE_OBJECT class
!define SERVICE class
!define AGGREGATE_ROOT class

skinparam class {
    BackgroundColor<<AGGREGATE_ROOT>> LightSalmon
    BackgroundColor<<ENTITY>> Wheat
    BackgroundColor<<VALUE_OBJECT>> LightBlue
    BackgroundColor<<SERVICE>> LightGreen
}

package "OrderManagement" {
    AGGREGATE_ROOT "Order" as Order {
```

```
- id: OrderId
- customerId: CustomerId
- status: OrderStatus
- items: List<OrderLineItem>
- paymentInfo: PaymentInfo
- shippingAddress: Address
+ createOrder(customer, items)
+ addItem(product, quantity)
+ removeItem(lineItemId)
+ confirmOrder()
+ markAsShipped()
+ cancel()
}
```

```
ENTITY "OrderLineItem" as OrderLineItem {
- id: OrderLineItemId
- productId: ProductId
- quantity: int
- price: Money
+ calculateTotal(): Money
}
```

```
VALUE_OBJECT "Money" as Money {
- amount: BigDecimal
- currency: CurrencyCode
+ add(Money): Money
+ subtract(Money): Money
+ multiply(factor): Money
}
```

```
VALUE_OBJECT "OrderStatus" as OrderStatus {
<<enumeration>>
NEW
CONFIRMED
PAID
SHIPPED
DELIVERED
CANCELLED
```

```
}

VALUE_OBJECT "Address" as Address {
    - street: String
    - city: String
    - zipCode: String
    - country: String
    + validate()
}

SERVICE "OrderProcessingService" as OrderProcessingService {
    + processOrder(orderId)
    + cancelOrder(orderId, reason)
    + applyDiscount(orderId, discount)
}
}

package "Inventory" {
    AGGREGATE_ROOT "Product" as Product {
        - id: ProductId
        - name: String
        - description: String
        - price: Money
        - inStock: int
        + decreaseStock(quantity)
        + increaseStock(quantity)
        + changePrice(newPrice)
    }
}

package "Customer" {
    AGGREGATE_ROOT "Customer" as Customer {
        - id: CustomerId
        - name: String
        - email: EmailAddress
        - defaultShippingAddress: Address
        - paymentMethods: List<PaymentMethod>
        + updateContactInfo(name, email)
    }
}
```

```

+ addPaymentMethod(paymentMethod)
+ removePaymentMethod(paymentMethodId)

}

VALUE_OBJECT "EmailAddress" as EmailAddress {
    - address: String
    + validate()
}
}

Order "1" *-- "0..*" OrderLineItem : contains
Order --> "1" OrderStatus : has
Order --> "1" Address : ships to
OrderLineItem --> "1" Money : has price
OrderLineItem --> "1" Product : references
Order --> "1" Customer : belongs to

@enduml

```

Este diagrama UML de clases contiene elementos avanzados como:

- Estereotipos personalizados para identificar patrones DDD
- Relaciones necesarias entre agregados
- Separación clara de entidades y objetos de valor
- Agrupación por contextos delimitados

Mejores Prácticas Avanzadas de Modelado UML

1. **Vistas 4+1** : Estructurar modelos en vistas lógicas, de procesos, de desarrollo, físicas y de escenarios.
2. **Niveles de abstracción coherentes** : Mantener consistente en el nivel de detalle dentro de cada diagrama.
3. **Trazabilidad de requisitos** : Vincular elementos de modelo con requisitos del sistema.
4. **Perfil UML personalizado** : Desarrollar extensiones UML específicas para el dominio.
5. **Versionado de modelos** : Gestionar la evolución de los modelos UML como parte del ciclo de vida del software.

1.5 Principios SOLID

Los principios SOLID constituyen un conjunto de directrices fundamentales para el diseño orientado a objetos que, cuando se aplican adecuadamente, conducen a sistemas más mantenibles, extensibles y robustos.

S - Principio de Responsabilidad Única (SRP)

"Una clase debe tener una, y solo una, razón para cambiar."

Implementación Avanzada del SRP

El SRP va más allá de simplemente tener clases pequeñas. Implica alinear las responsabilidades con los actores o stakeholders del sistema.

Java

```
// Violación del SRP
class Employee {
    private String name;
    private double salary;

    public void calculatePay() { /* Lógica de cálculo de nómina */ }
    public void saveToDatabase() { /* Lógica de persistencia */ }
    public void generateReport() { /* Lógica de reportes */ }
}
```

```
// Aplicación correcta del SRP
class Employee {
    private String name;
    private double salary;

    public String getName() { return name; }
    public double getSalary() { return salary; }
    // Otros getters/setters pert
```

Reintentar

Claude aún no tiene la capacidad de ejecutar el código que genera.

Yo

Continuar

Editar

A continuación, desarollo la parte restante del documento sobre Fundamentos Avanzados de Diseño Orientado a Objetos.

1.5 Principios SOLID (continuación)

Implementación Avanzada del SRP (continuación)

Java

```
// Aplicación correcta del SRP

class Employee {
    private String name;
    private double salary;
    private EmployeeType type;

    public String getName() { return name; }
    public double getSalary() { return salary; }
    public EmployeeType getType() { return type; }
    // Otros getters/setters pertinentes
}

class PayrollCalculator {
    public Money calculatePay(Employee employee) {
        // Lógica específica de cálculo de nómina
        return new Money(/* cálculo basado en employee */);
    }
}

class EmployeeRepository {
    public void save(Employee employee) {
        // Lógica específica de persistencia
    }

    public Employee findById(EmployeeId id) {
        // Lógica de búsqueda
        return null;
    }
}

class EmployeeReportGenerator {
```

```

public Report generateReport(Employee employee) {
    // Lógica específica de generación de reportes
    return new Report(/* basado en employee */);
}

```

Detección de violaciones del SRP

- Análisis de cambios** : Si una clase cambia por múltiples razones no relacionadas.
- Análisis de cohesión** : Métodos que no utilizan los mismos atributos de clase.
- Análisis de dependencias** : Una clase con dependencias diversas no relacionadas.
- Patrones lingüísticos** : Nombres de métodos como "calcular", "guardar", "validar", "enviar" en la misma clase.

Beneficios Estratégicos del SRP

- Rastreo claro de cambios** : Las modificaciones se concentran en áreas específicas del código.
- Menor riesgo de efectos secundarios** : Los cambios por una razón no afectan código no relacionado.
- Facilita la comprensión** : Las clases más pequeñas y enfocadas son más fáciles de entender.
- Mejora la testabilidad** : Las responsabilidades aisladas se prueban de forma independiente.

O - Principio de Abierto/Cerrado (OCP)

"Las entidades de software deben estar abiertas para la extensión, pero cerradas para la modificación".

Implementación Avanzada del OCP

El OCP va más allá de usar herencia o interfaces; implica identificar y aislar los puntos de variación en el sistema.

Java

```

// Violación del OCP
class OrderProcessor {
    public void process(Order order) {
        if (order.getType() == OrderType.RETAIL) {
            // Lógica para pedidos minoristas
        } else if (order.getType() == OrderType.WHOLESALE) {
            // Lógica para pedidos mayoristas
        } else if (order.getType() == OrderType.INTERNATIONAL) {

```

```
// Lógica para pedidos internacionales
}

// Si agregamos un nuevo tipo de pedido, tenemos que modificar esta clase
}

}

// Aplicación correcta del OCP
interface OrderHandler {
    void process(Order order);
}

class RetailOrderHandler implements OrderHandler {
    @Override
    public void process(Order order) {
        // Lógica específica para pedidos minoristas
    }
}

class WholesaleOrderHandler implements OrderHandler {
    @Override
    public void process(Order order) {
        // Lógica específica para pedidos mayoristas
    }
}

class InternationalOrderHandler implements OrderHandler {
    @Override
    public void process(Order order) {
        // Lógica específica para pedidos internacionales
    }
}

class OrderProcessorV2 {
    private Map<OrderType, OrderHandler> handlers;

    public OrderProcessorV2() {
        handlers = new HashMap<>();
        // Registrar handlers para diferentes tipos de pedidos
    }
}
```

```

        handlers.put(OrderType.RETAIL, new RetailOrderHandler());
        handlers.put(OrderType.WHOLESALE, new WholesaleOrderHandler());
        handlers.put(OrderType.INTERNATIONAL, new InternationalOrderHandler());
    }

    public void process(Order order) {
        OrderHandler handler = handlers.get(order.getType());
        if (handler == null) {
            throw new UnsupportedOrderTypeException(order.getType());
        }
        handler.process(order);
    }

    // Para agregar soporte para un nuevo tipo de pedido,
    // simplemente registramos un nuevo handler sin modificar el código existente
    public void registerHandler(OrderType type, OrderHandler handler) {
        handlers.put(type, handler);
    }
}

```

Técnicas Avanzadas para el OCP

1. **Mecanismos de Plugin** : Permitir que las extensiones se carguen dinámicamente.
2. **Inyección de Estrategias** : Configurar comportamientos en tiempo de ejecución.
3. **Puntos de Extensión definidos** : Identificar y documentar claramente dónde y cómo extender.
4. **Composición sobre Herencia** : Preferir composición para mayor flexibilidad.

Equilibrio en la Aplicación del OCP

El OCP no se aplica debe prematuramente a todas las áreas del código:

1. **Análisis de estabilidad** : Aplicar primero a las áreas más estables y fundamentales.
2. **Principio YAGNI** : No implementar puntos de extensión que no se necesitan aún.
3. **Costo vs Beneficio** : Equilibrar la complejidad adicional con el beneficio de extensibilidad.

L - Principio de Sustitución de Liskov (LSP)

"Los objetos de una clase derivada deben poder reemplazar a los objetos de la clase base sin afectar la corrección del programa."

Implementación Avanzada del LSP

El LSP va más allá de la simple compatibilidad de tipos; abarca el comportamiento completo de los objetos.

Java

```
// Violación del LSP

class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}

class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width; // Violación del LSP - cambia comportamiento
        esperado
    }

    @Override
    public void setHeight(int height) {
        this.height = height;
        this.width = height; // Violación del LSP - cambia comportamiento
        esperado
    }
}

// Código que depende del comportamiento de Rectangle
```

```
void calculateArea(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    assert r.getArea() == 20; // Falla si r es un Square
}

// Aplicación correcta del LSP
interface Shape {
    int getArea();
}

class Rectangle implements Shape {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    @Override
    public int getArea() {
        return width * height;
    }
}

class Square implements Shape {
    private int side;

    public Square(int side) {
```

```

        this.side = side;
    }

    public void setSide(int side) {
        this.side = side;
    }

    @Override
    public int getArea() {
        return side * side;
    }
}

```

Condiciones Avanzadas del LSP

1. **Precondiciones** : No fortalecer las precondiciones en las subclases.
2. **Postcondiciones** : No debilitar postcondiciones en las subclases.
3. **Invariantes** : Mantener los invariantes de la clase base.
4. **Principio de Historia** : Subclases no deben lanzar excepciones para métodos que no las lanzan en la clase base.
5. **Sustitución de Comportamiento** : La implementación debe corresponder a la intención semántica del contrato.

Indicadores de Violaciones del LSP

1. **Comprobaciones de tipo en tiempo de ejecución** : `instanceof` o equivalentes.
2. **Métodos que arrojan excepciones tipo "UnsupportedOperationException"** .
3. **Sobrescrituras que vacían funcionalidad** : Métodos que no hacen nada o lanzan excepciones.
4. **Comentarios de advertencia** : "No usar con clase X" o "Precaución al usar con subclase Y".

I - Principio de Segregación de Interfaces (ISP)

"Los clientes no deben verse forzados a depender de interfaces que no utilizan."

Implementación Avanzada del ISP

El ISP promueve interfaces cohesivas, específicas para cada cliente en lugar de interfaces monolíticas.

Java

```
// Violación del ISP
interface Worker {
```

```

    void work();
    void eat();
    void sleep();
}

class Human implements Worker {
    public void work() { /* implementación */ }
    public void eat() { /* implementación */ }
    public void sleep() { /* implementación */ }
}

class Robot implements Worker {
    public void work() { /* implementación */ }
    public void eat() { /* No aplicable - implementación vacía o error */ }
    public void sleep() { /* No aplicable - implementación vacía o error */ }
}

// Aplicación correcta del ISP
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

interface Sleepable {
    void sleep();
}

class Human implements Workable, Eatable, Sleepable {
    public void work() { /* implementación */ }
    public void eat() { /* implementación */ }
    public void sleep() { /* implementación */ }
}

class Robot implements Workable {
    public void work() { /* implementación */ }
}

```

```
// No implementa interfaces que no necesita
}
```

Técnicas Avanzadas para el ISP

1. **Análisis de dependencias** : Identificar grupos de métodos utilizados juntos por clientes específicos.
2. **Refactorización de interfaces existentes** : Dividir interfaces grandes en componentes cohesivos.
3. **Interfaces adaptadoras** : Crear vistas específicas de cliente sobre interfaces más grandes cuando sea necesario.
4. **Composición de interfaces** : Construir interfaces complejas mediante la extensión de interfaces más simples.

Beneficios Estratégicos del ISP

1. **Desacoplamiento entre clientes** : Los cambios en una interfaz afectan solo a los clientes relevantes.
2. **Mejor expresividad del código** : Las interfaces comunican claramente su propósito.
3. **Mayor cohesión** : Interfaces enfocadas en aspectos específicos del sistema.
4. **Facilitar la implementación** : Reducir la probabilidad de métodos vacíos o innecesarios.

D - Principio de Inversión de Dependencias (DIP)

"Los módulos de alto nivel no deben depender de módulos de bajo nivel. Ambos deben depender de abstracciones. Las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones."

Implementación Avanzada del DIP

El DIP establece una arquitectura donde el flujo de control y el flujo de dependencias van en direcciones opuestas.

```
Java
// Violación del DIP
class ReportGenerator {
    private MySQLDatabase database; // Dependencia directa de un detalle

    public ReportGenerator() {
        this.database = new MySQLDatabase(); // Creación directa de dependencia
    }

    public Report generateReport(ReportType type) {
```

```
        List<ReportData> data = database.query("SELECT * FROM report_data WHERE
type = " + type);
        // Procesamiento y generación del reporte
        return new Report(data);
    }
}

// Aplicación correcta del DIP
interface DataSource {
    List<ReportData> getReportData(ReportType type);
}

class MySQLDataSource implements DataSource {
    public List<ReportData> getReportData(ReportType type) {
        // Implementación específica para MySQL
        return executeQuery("SELECT * FROM report_data WHERE type = " + type);
    }

    private List<ReportData> executeQuery(String sql) {
        // Lógica específica de MySQL
        return new ArrayList<>();
    }
}

class MongoDBDataSource implements DataSource {
    public List<ReportData> getReportData(ReportType type) {
        // Implementación específica para MongoDB
        return findDocuments("report_data", Map.of("type", type));
    }

    private List<ReportData> findDocuments(String collection, Map<String, Object>
criteria) {
        // Lógica específica de MongoDB
        return new ArrayList<>();
    }
}

class ReportGenerator {
```

```

private final DataSource dataSource; // Dependencia de una abstracción

// Inyección de dependencia - La dependencia viene del exterior
public ReportGenerator(DataSource dataSource) {
    this.dataSource = dataSource;
}

public Report generateReport(ReportType type) {
    List<ReportData> data = dataSource.getReportData(type);
    // Procesamiento y generación del reporte
    return new Report(data);
}
}

```

Patrones Avanzados de Inyección de Dependencias

1. **Inyección por Constructor** : Las dependencias se pasan al constructor.
2. **Inyección por Setter** : Las dependencias se configuran mediante métodos setter.
3. **Inyección por interfaz** : La clase implementa una interfaz que define métodos para establecer dependencias.
4. **Inyección por Contexto de Servicio** : Las dependencias se obtienen de un registro central.

Contenedores de IoC (Inversión de Control)

Los contenedores de IoC automatizan la gestión e inyección de dependencias:

1. **Registro de Componentes** : Los componentes y sus dependencias se registran.
2. **Resolución de Dependencias** : El contenedor resuelve automáticamente el gráfico de dependencias.
3. **Gestión de Ciclo de Vida** : Control sobre cuándo y cómo se crean, se comparten y se destruyen objetos.
4. **Configuración por Composición** : Las dependencias se configuran externamente sin cambiar el código.

Estrategias para abstracciones efectivas

1. **Diseño orientado al dominio** : Las abstracciones deben reflejar conceptos del dominio, sin detalles técnicos.
2. **Abstracciones basadas en comportamiento** : Enfocarse en lo que hacen los objetos, no en lo que son.
3. **Principio de Hollywood** : "No nos llames, nosotros te llamaremos" - definir puntos de devolución de llamada.
4. **Abstracciones delgadas** : Interfaces con pocos métodos, enfocadas en un aspecto específico.

1.6 Principios GRASP

Los principios GRASP (General Responsibility Assignment Software Patterns) proporcionan directrices para asignar responsabilidades a clases e interacciones entre objetos, considerando el comportamiento futuro y la mantenibilidad del sistema.

1. Experto en información

"Asignar una responsabilidad a la clase que tiene la información necesaria para cumplirla."

Implementación Avanzada del Experto en Información

Java

```
// Simplista (sin aplicar Experto)
class OrderProcessor {
    public void processOrder(Order order, Customer customer, Inventory inventory) {
        // OrderProcessor calcula el total pero la información está en Order
        double total = 0;
        for (OrderLine line : order.getLines()) {
            total += line.getQuantity() * line.getProduct().getPrice();
        }
        order.setTotal(total);

        // Verifica disponibilidad pero la información está en Inventory
        for (OrderLine line : order.getLines()) {
            if (!inventory.hasStock(line.getProduct(), line.getQuantity())) {
                throw new InsufficientStockException();
            }
        }

        // Actualiza el saldo pero la información está en Customer
        if (customer.getBalance() < total) {
            throw new InsufficientFundsException();
        }
        customer.decreaseBalance(total);
    }
}

// Aplicando Experto
```

```

class Order {
    private List<OrderLine> lines;
    private Customer customer;
    private Money total;

    public Money calculateTotal() {
        // Order conoce sus líneas y puede calcular su propio total
        return lines.stream()
            .map(OrderLine::calculateSubtotal)
            .reduce(Money.ZERO, Money::add);
    }

    public boolean canBeFullfilledBy(Inventory inventory) {
        // Delega en cada línea para verificar disponibilidad
        return lines.stream()
            .allMatch(line -> line.isAvailableIn(inventory));
    }

    public void confirm() {
        // Order conoce su cliente y puede solicitar el pago
        total = calculateTotal();
        customer.charge(total);
    }
}

class OrderLine {
    private Product product;
    private int quantity;

    public Money calculateSubtotal() {
        // OrderLine conoce su producto y cantidad
        return product.getPrice().multiply(quantity);
    }

    public boolean isAvailableIn(Inventory inventory) {
        // OrderLine conoce qué necesita verificar
        return inventory.hasStock(product, quantity);
    }
}

```

```

}

class Customer {
    private Money balance;

    public void charge(Money amount) {
        // Customer conoce su balance y puede verificar fondos
        if (balance.isLessThan(amount)) {
            throw new InsufficientFundsException();
        }
        balance = balance.subtract(amount);
    }
}

// Proceso simplificado usando Experto
class OrderProcessor {
    public void processOrder(Order order, Inventory inventory) {
        if (!order.canBeFullfilledBy(inventory)) {
            throw new InsufficientStockException();
        }
        order.confirm();
        inventory.fillOrder(order);
    }
}

```

Ventajas Estratégicas del Experto

- Encapsulamiento conservado** : La información y el comportamiento relacionado permanecen juntos.
- Representación natural del dominio** : El diseño refleja cómo los expertos del dominio piensan sobre el problema.
- Sistemas más mantenibles** : Los cambios suelen estar localizados en clases específicas.
- Sistemas más reutilizables** : El comportamiento está donde tiene sentido reutilizarlo.

2. Creador

"Asignar a la clase B la responsabilidad de crear una instancia de clase A si se cumple al menos una de estas condiciones: B agrega objetos A, B contiene objetos A, B registra instancias de objetos A, B usa específicamente objetos A, B tiene los datos necesarios para inicializar A."

Implementación Avanzada del Creador

Java

```
// Sin aplicar Creador
class OrderSystem {
    public void createNewOrder(Customer customer) {
        // Creación directa de objetos relacionados
        Order order = new Order();
        order.setCustomer(customer);
        order.setCreationDate(new Date());
        order.setStatus(OrderStatus.NEW);

        // Sistema externo creando componentes internos del pedido
        OrderHeader header = new OrderHeader();
        header.setOrderNumber(generateOrderNumber());
        order.setHeader(header);
    }
}

// Aplicando Creador
class Order {
    private Customer customer;
    private Date creationDate;
    private OrderStatus status;
    private OrderHeader header;
    private List<OrderLine> lines;

    // Constructor que sigue el patrón Creador
    public Order(Customer customer) {
        this.customer = customer;
        this.creationDate = new Date();
        this.status = OrderStatus.NEW;
        this.header = createHeader(); // La Orden crea su propio encabezado
        this.lines = new ArrayList<>();
    }

    private OrderHeader createHeader() {
        return new OrderHeader(generateOrderNumber());
    }
}
```

```

    }

    private String generateOrderNumber() {
        // Lógica para generar número de orden único
        return "ORD-" + System.currentTimeMillis();
    }

    // Order es creador natural de OrderLine (las contiene)
    public OrderLine addProduct(Product product, int quantity) {
        OrderLine line = new OrderLine(this, product, quantity);
        lines.add(line);
        return line;
    }
}

// Uso simplificado con Creador
class OrderSystem {
    public Order createNewOrder(Customer customer) {
        return new Order(customer); // Order se encarga de crear sus componentes
    }
}

```

Consideraciones Avanzadas del Creador

- Fábricas dedicadas** : Cuando la creación es compleja, considere el patrón Factory.
- Constructores para configuración flexible** : cuando un objeto tiene múltiples configuraciones opcionales.
- Equilibrio con Responsabilidad Única** : La responsabilidad de creación no debe comprometer la cohesión.
- Contexto de la creación** : Considere si la creación debe ocurrir en el contexto de una transacción o proceso.

3. Controlador

"Asignar la responsabilidad de gestionar un mensaje de evento del sistema a una clase que represente uno de estos: (1) El subsistema global (fachada); (2) Un escenario de caso de uso; (3) Un dispositivo o componente de la capa de presentación."

Implementación Avanzada del Controlador

Java

```
// Controlador demasiado acoplado
```

```
class OrderController {  
    private Database database;  
    private EmailService emailService;  
    private PaymentGateway paymentGateway;  
  
    public void createOrder(OrderDTO orderDTO) {  
        // Controlador realizando operaciones de dominio  
        Customer customer = database.findCustomer(orderDTO.getCustomerId());  
  
        Order order = new Order();  
        for (OrderLineDTO lineDTO : orderDTO.getLines()) {  
            Product product = database.findProduct(lineDTO.getProductId());  
            order.addLine(product, lineDTO.getQuantity());  
        }  
  
        // Controlador realizando Lógica de negocio  
        double total = order.calculateTotal();  
        if (customer.getBalance() < total) {  
            throw new InsufficientFundsException();  
        }  
  
        // Procesamiento de pago directo  
        PaymentResult result = paymentGateway.processPayment(  
            customer.getPaymentInfo(), total);  
        if (!result.isSuccessful()) {  
            throw new PaymentFailedException();  
        }  
  
        // Persistencia directa  
        database.saveOrder(order);  
  
        // Notificación directa  
        emailService.sendOrderConfirmation(customer.getEmail(), order);  
    }  
}  
  
// Controlador GRASP bien implementado  
class CreateOrderController {
```

```
private final OrderService orderService;

public CreateOrderController(OrderService orderService) {
    this.orderService = orderService;
}

public OrderResponseDTO createOrder(OrderRequestDTO request) {
    try {
        // Controlador delegando en servicios de aplicación
        Order order = orderService.createOrder(
            request.getCustomerId(),
            request.getItems(),
            request.getShippingAddress()
        );

        // Solo transformación de la respuesta
        return OrderDTOMapper.toResponseDTO(order);
    } catch (BusinessException e) {
        // Manejo de excepciones de negocio
        throw new OrderCreationException("Failed to create order", e);
    }
}

// Servicio de aplicación que coordina el caso de uso
class OrderService {
    private final CustomerRepository customerRepository;
    private final ProductRepository productRepository;
    private final OrderRepository orderRepository;
    private final PaymentService paymentService;
    private final NotificationService notificationService;

    public Order createOrder(CustomerId customerId,
                           List<OrderItemRequest> items,
                           Address shippingAddress) {
        // Recuperación de entidades y validación inicial
        Customer customer = customerRepository.findById(customerId)
            .orElseThrow(() -> new CustomerNotFoundException(customerId));
    }
}
```

```

// Creación de la orden usando el patrón Creador
Order order = customer.createOrder(shippingAddress);

// Agregar productos verificando disponibilidad
for (OrderItemRequest item : items) {
    Product product = productRepository.findById(item.getProductId())
        .orElseThrow(() -> new
ProductNotFoundException(item.getProductId()));

    order.addProduct(product, item.getQuantity());
}

// Procesamiento de pago delegado al dominio
paymentService.processPayment(order);

// Persistencia
orderRepository.save(order);

// Notificaciones como eventos del dominio
notificationService.notifyOrderCreated(order);

return order;
}
}

```

Tipos de controladores

1. **Controladores de Fachada** : Representan el sistema completo.
2. **Controladores de Caso de Uso** : Manejan un escenario específico de interacción.
3. **Controladores de Dispositivo** : Gestionan eventos desde dispositivos específicos.
4. **Controladores de Comando** : Implementan un patrón Command para operaciones específicas.

Principios para Controladores Efectivos

1. **Delegación adecuada** : El controlador coordina pero no procesa directamente.
2. **Cohesión de caso de uso** : Un controlador por caso de uso o grupo relacionado.
3. **Sin lógica de dominio** : Los controladores no implementan reglas de negocio.
4. **Sin estado de dominio** : Los controladores no mantienen el estado del modelo.

4. Bajo Acoplamiento

"Asignar una responsabilidad para mantener bajo acoplamiento. El acoplamiento es una medida de qué tan interconectado es un elemento con otros."

Técnicas Avanzadas para Bajo Acoplamiento

1. **Dependencia de abstracciones** : Depender de interfaces o clases abstractas.
2. **Eventos de dominio** : Comunicación por eventos en lugar de referencias directas.
3. **Mediadores** : Objetos intermediarios que coordinan interacciones.
4. **Adaptadores** : Traducción entre interfaces incompatibles.

Java

```
// Alto acoplamiento
class ShoppingCart {
    private List<Product> products;
    private Customer customer;
    private PaymentProcessor paymentProcessor;
    private InventorySystem inventorySystem;
    private EmailService emailService;
    private Logger logger;

    public void checkout() {
        double total = calculateTotal();

        // Acoplamiento directo a muchos sistemas externos
        if (!paymentProcessor.processPayment(customer, total)) {
            logger.error("Payment failed for customer: " + customer.getId());
            return;
        }

        // Actualización directa de inventario
        for (Product product : products) {
            inventorySystem.decreaseStock(product.getId(), 1);
        }

        // Envío directo de correo
        emailService.sendOrderConfirmation(customer.getEmail(), products, total);

        logger.info("Order completed for customer: " + customer.getId());
    }
}
```

```

// Bajo acoplamiento
class ShoppingCart {
    private List<CartItem> items;
    private CustomerId customerId;

    public Order checkout(CheckoutService checkoutService) {
        // El carrito solo conoce sus propios items
        return checkoutService.createOrder(customerId, items);
    }

    public Money calculateTotal() {
        return items.stream()
            .map(CartItem::getPrice)
            .reduce(Money.ZERO, Money::add);
    }
}

// Servicio que coordina el checkout
class CheckoutService {
    private final CustomerRepository customerRepository;
    private final OrderFactory orderFactory;
    private final PaymentService paymentService;
    private final OrderRepository orderRepository;
    private final EventPublisher eventPublisher;

    public Order createOrder(CustomerId customerId, List<CartItem> items) {
        Customer customer = customerRepository.findById(customerId)
            .orElseThrow(() -> new CustomerNotFoundException(customerId));

        // Creación de orden mediante factory
        Order order = orderFactory.createFrom(customer, items);

        // Procesamiento de pago mediante servicio
        paymentService.processPayment(order);

        // Persistencia
        orderRepository.save(order);
    }
}

```

```

    // Publicación de evento de dominio
    eventPublisher.publish(new OrderCreatedEvent(order));

    return order;
}

}

// Manejadores de eventos independientes
class InventoryEventHandler {
    private final InventoryService inventoryService;

    @EventHandler
    public void on(OrderCreatedEvent event) {
        // Actualización de inventario en respuesta al evento
        inventoryService.updateStock(event.getOrderItems());
    }
}

class NotificationEventHandler {
    private final EmailService emailService;

    @EventHandler
    public void on(OrderCreatedEvent event) {
        // Envío de correo en respuesta al evento
        emailService.sendOrderConfirmation(event.getOrder());
    }
}

```

5. Alta cohesión

"Asignar una responsabilidad para mantener alta cohesión. La cohesión mide cuán relacionados y enfocados están los elementos de un módulo."

Técnicas Avanzadas para Alta Cohesión

1. **Cohesión funcional** : Todos los elementos contribuyen a una única tarea.
2. **Descomposición basada en responsabilidades** : Dividir clases grandes por áreas de responsabilidad.
3. **Modelado por subdominios** : Agrupar clases por área del dominio del problema.

4. **Servicios de dominio dedicados** : Para operaciones que no encajan naturalmente en entidades.

Java

```
// Baja cohesión
class CustomerManager {
    public void register(Customer customer) { /* registro */ }
    public void updateProfile(Customer customer) { /* actualización */ }
    public void processPayment(Customer customer, double amount) { /* pago */ }
    public void generateReport(ReportType type) { /* reportes */ }
    public void sendPromotionalEmail(Customer customer) { /* marketing */ }
    public void calculateLoyaltyPoints(Customer customer) { /* programa lealtad */ }
}
public void handleComplaint(Customer customer, String complaint) { /* quejas */ }
```

```
// Alta cohesión con servicios especializados
class CustomerRegistrationService {
    private final CustomerRepository repository;
    private final EventPublisher eventPublisher;

    public Customer register(CustomerRegistrationRequest request) {
        Customer customer = Customer.create(request);
        repository.save(customer);
        eventPublisher.publish(new CustomerRegisteredEvent(customer));
        return customer;
    }
}

class CustomerProfileService {
    private final CustomerRepository repository;

    public void updateProfile(CustomerId id, CustomerProfileUpdate update) {
        Customer customer =

```

Reintentar

Claude aún no tiene la capacidad de ejecutar el código que genera.

Yo

Continuar

Editar

A continuación, continúa con el desarrollo del documento sobre Fundamentos Avanzados de Diseño Orientado a Objetos:

Java

```
class CustomerProfileService {  
    private final CustomerRepository repository;  
    private final EventPublisher eventPublisher;  
  
    public void updateProfile(CustomerId id, CustomerProfileUpdate update) {  
        Customer customer = repository.findById(id)  
            .orElseThrow(() -> new CustomerNotFoundException(id));  
  
        customer.updateProfile(update);  
        repository.save(customer);  
        eventPublisher.publish(new CustomerProfileUpdatedEvent(customer));  
    }  
}  
  
class PaymentService {  
    private final CustomerRepository customerRepository;  
    private final PaymentGateway paymentGateway;  
    private final TransactionRepository transactionRepository;  
  
    public Transaction processPayment(CustomerId customerId, Money amount,  
        PaymentMethod method) {  
        Customer customer = customerRepository.findById(customerId)  
            .orElseThrow(() -> new CustomerNotFoundException(customerId));  
  
        Transaction transaction = customer.createTransaction(amount, method);  
        PaymentResult result = paymentGateway.process(transaction);  
  
        if (result.isSuccessful()) {  
            transaction.markAsSuccessful(result.getTransactionId());  
        } else {  
            transaction.markAsFailed(result.getErrorCode(),  
                result.getErrorMessage());  
        }  
    }  
}
```

```

    }

    transactionRepository.save(transaction);
    return transaction;
}

}

class CustomerMarketingService {
    // Métodos específicos para marketing y promociones
}

class CustomerSupportService {
    // Métodos específicos para atención al cliente
}

class LoyaltyProgramService {
    // Métodos específicos para el programa de lealtad
}

```

Este ejemplo muestra cómo desglosar una clase poco cohesiva (`CustomerManager`) en servicios de alta cohesión donde cada uno se enfoca en un conjunto específico de responsabilidades relacionadas.

6. Polimorfismo

"Asignar responsabilidades para comportamientos que varían por tipo usando operaciones polimórficas. Cuando alternativas o comportamientos relacionados varían por tipo, asigne la responsabilidad para el comportamiento a los tipos para los que el comportamiento varía."

Implementación Avanzada del Polimorfismo

Java

```

// Sin polimorfismo (usando condicionales)
class TaxCalculator {

    public double calculateTax(Product product, String region) {
        double tax = 0;

        if (region.equals("EU")) {
            if (product.getType() == ProductType.DIGITAL) {
                tax = product.getPrice() * 0.21; // IVA digital en UE
            } else if (product.getType() == ProductType.PHYSICAL) {

```

```

        tax = product.getPrice() * 0.18; // IVA físico en UE
    } else if (product.getType() == ProductType.SERVICE) {
        tax = product.getPrice() * 0.25; // IVA servicios en UE
    }
} else if (region.equals("US")) {
    if (product.getState().equals("CA")) {
        tax = product.getPrice() * 0.0725; // Impuesto California
    } else if (product.getState().equals("NY")) {
        tax = product.getPrice() * 0.045; // Impuesto Nueva York
    }
    // Más estados...
}

return tax;
}
}

// Con polimorfismo
interface TaxCalculationStrategy {
    Money calculateTax(Product product);
}

// Jerarquía para regiones
abstract class RegionalTaxCalculator implements TaxCalculationStrategy {
    protected abstract Money getTaxRate(ProductType productType);

    @Override
    public Money calculateTax(Product product) {
        Money taxRate = getTaxRate(product.getType());
        return product.getPrice().multiply(taxRate);
    }
}

class EUTaxCalculator extends RegionalTaxCalculator {
    private static final Map<ProductType, Money> TAX_RATES = Map.of(
        ProductType.DIGITAL, new Money("0.21"),
        ProductType.PHYSICAL, new Money("0.18"),
        ProductType.SERVICE, new Money("0.25")
    )
}
```

```
);

@Override
protected Money getTaxRate(ProductType productType) {
    return TAX_RATES.getOrDefault(productType, Money.ZERO);
}

}

// Jerarquía para estados de EE.UU.

abstract class USTaxCalculator implements TaxCalculationStrategy {
    @Override
    public Money calculateTax(Product product) {
        // Posible acceso a APIs de impuestos
        return getStateRate().multiply(product.getPrice());
    }

    protected abstract Money getStateRate();
}

class CaliforniaTaxCalculator extends USTaxCalculator {
    @Override
    protected Money getStateRate() {
        return new Money("0.0725");
    }
}

class NewYorkTaxCalculator extends USTaxCalculator {
    @Override
    protected Money getStateRate() {
        return new Money("0.045");
    }
}

// Fábrica que proporciona el calculador adecuado
class TaxCalculatorFactory {
    private final Map<String, TaxCalculationStrategy> calculators = new
    HashMap<>();
}
```

```

public TaxCalculatorFactory() {
    // Registro de calculadores
    calculators.put("EU", new EUTaxCalculator());
    calculators.put("US-CA", new CaliforniaTaxCalculator());
    calculators.put("US-NY", new NewYorkTaxCalculator());
    // Más regiones...
}

public TaxCalculationStrategy getCalculator(String regionCode) {
    TaxCalculationStrategy calculator = calculators.get(regionCode);
    if (calculator == null) {
        throw new UnsupportedRegionException(regionCode);
    }
    return calculator;
}

// Uso del polimorfismo
class OrderService {
    private final TaxCalculatorFactory taxCalculatorFactory;

    public void calculateTaxes(Order order) {
        TaxCalculationStrategy calculator =
taxCalculatorFactory.getCalculator(order.getCustomer().getRegionCode());

        // Aplicación polimórfica del cálculo de impuestos
        for (OrderLine line : order.getLines()) {
            Money tax = calculator.calculateTax(line.getProduct());
            line.setTax(tax);
        }
    }
}

```

Ventajas Estratégicas del Polimorfismo

- Extensibilidad mejorada** : Nuevas variantes requieren nuevas clases, no modificar el código existente.
- Código más limpio** : Eliminación de condicionales y lógica de tipo.

3. **Mejor distribución de responsabilidades** : Cada clase maneja su comportamiento específico.
4. **Facilita pruebas unitarias** : Cada variante puede probarse de forma aislada.

7. Fabricación Pura

"Asignar un conjunto altamente cohesivo de responsabilidades a una clase artificial o de conveniencia que no representa un concepto del dominio del problema—algo inventado para soportar una alta cohesión, bajo acoplamiento, y reutilización."

Implementación Avanzada de Fabricación Pura

Java

```
// Sin fabricación pura

class Customer {
    private String email;

    public void sendWelcomeEmail() {
        // Lógica de envío de email directamente en Customer
        String subject = "Welcome to our platform!";
        String body = "Dear customer, thank you for registering...";

        // Configuración SMTP
        Properties props = new Properties();
        props.put("mail.smtp.host", "smtp.example.com");
        props.put("mail.smtp.port", "587");
        // Más configuración...

        // Código de envío de email
        // ...
    }

    public void sendPasswordResetEmail(String token) {
        // Duplicación de lógica de email similar
    }

    public void sendOrderConfirmation(Order order) {
        // Más duplicación...
    }
}
```

```
// Con fabricación pura
class Customer {
    private String email;

    public String getEmail() {
        return email;
    }
}

// Fabricación pura para envío de emails
class EmailService {
    private final EmailTemplateEngine templateEngine;
    private final EmailSender emailSender;

    public void sendWelcomeEmail(String recipientEmail) {
        EmailTemplate template = templateEngine.getTemplate("welcome");
        EmailMessage message = template.createMessage(Map.of("email",
recipientEmail));
        emailSender.send(recipientEmail, message);
    }

    public void sendPasswordResetEmail(String recipientEmail, String token) {
        EmailTemplate template = templateEngine.getTemplate("password-reset");
        EmailMessage message = template.createMessage(Map.of(
            "email", recipientEmail,
            "token", token,
            "resetUrl", "https://example.com/reset?token=" + token
        ));
        emailSender.send(recipientEmail, message);
    }

    public void sendOrderConfirmation(String recipientEmail, Order order) {
        EmailTemplate template = templateEngine.getTemplate("order-confirmation");
        Map<String, Object> params = new HashMap<>();
        params.put("order", order);
        params.put("items", order.getItems());
        params.put("total", order.getTotal());
    }
}
```

```

        EmailMessage message = template.createMessage(params);
        emailSender.send(recipientEmail, message);
    }
}

// Uso de la fabricación pura
class CustomerService {
    private final CustomerRepository repository;
    private final EmailService emailService;

    public void registerCustomer(CustomerRegistrationRequest request) {
        Customer customer = new Customer(request.getEmail(), request.getName());
        repository.save(customer);

        // Customer ya no necesita saber cómo enviar emails
        emailService.sendWelcomeEmail(customer.getEmail());
    }

    public void requestPasswordReset(String email) {
        Customer customer = repository.findByEmail(email)
            .orElseThrow(() -> new CustomerNotFoundException(email));

        String token = generateResetToken(customer);
        emailService.sendPasswordResetEmail(customer.getEmail(), token);
    }
}

```

Indicadores para Usar Fabricación Pura

- Responsabilidades que no encajan naturalmente** : Funcionalidades que no pertenecen conceptualmente a ninguna clase existente.
- Servicios técnicos reutilizables** : Funcionalidades de infraestructura o transversales.
- Prevención de asociación** : Cuando asignar responsabilidades a clases del dominio crearía dependencias indeseables.
- Encapsulación de API externas** : Aislar el sistema de dependencias externas.

8. Indirección

"Asignar la responsabilidad a un objeto intermedio que media entre componentes o servicios para que estos no se acoplen directamente."

Implementación Avanzada de Indirección

```
Java

// Sin indirección
class ReportGenerator {
    private MySQLDatabase database; // Acoplamiento directo a MySQL

    public Report generateSalesReport(Date from, Date to) {
        // Consulta directa a MySQL
        String sql = "SELECT * FROM sales WHERE date BETWEEN ? AND ?";
        List<Sale> sales = database.executeQuery(sql, from, to);

        // Procesamiento del reporte
        return new Report(sales);
    }
}

// Con indirección
// Abstracción de acceso a datos
interface SaleRepository {
    List<Sale> findSalesBetween(Date from, Date to);
}

// Implementación concreta que conoce MySQL
class MySQLSaleRepository implements SaleRepository {
    private final DataSource dataSource;

    @Override
    public List<Sale> findSalesBetween(Date from, Date to) {
        // Implementación específica para MySQL
        return executeQuery("SELECT * FROM sales WHERE date BETWEEN ? AND ?",
                           from, to);
    }
}

// Alternativa para MongoDB
class MongoDBSaleRepository implements SaleRepository {
    private final MongoClient mongoClient;
```

```

@Override
public List<Sale> findSalesBetween(Date from, Date to) {
    // Implementación específica para MongoDB
    return executeQuery(
        Document.parse("{date: {$gte: '" + from + "', $lte: '" + to + "'}}"));
}
}

// ReportGenerator usa el repositorio a través de indirección
class ReportGenerator {
    private final SaleRepository saleRepository; // Acoplado a La abstracción

    public ReportGenerator(SaleRepository saleRepository) {
        this.saleRepository = saleRepository;
    }

    public Report generateSalesReport(Date from, Date to) {
        // Uso de La abstracción sin conocer Los detalles
        List<Sale> sales = saleRepository.findSalesBetween(from, to);

        // Procesamiento del reporte
        return new Report(sales);
    }
}

```

Patrones Comunes de Indirección

1. **Adaptador** : Convierte la interfaz de una clase en otra interfaz que esperan los clientes.
2. **Fachada** : Proporciona una interfaz unificada a un conjunto de interfaces.
3. **Proxy** : Proporciona un sustituto o marcador de posición para controlar el acceso a un objeto.
4. **Mediador** : Define un objeto que encapsula cómo interactúan un conjunto de objetos.
5. **Capa de servicio** : Centraliza la lógica de aplicación y coordina transacciones.

9. Variaciones Protegidas

"Proteger elementos de las variaciones en otros elementos encapsulando la variación detrás de una interfaz y usando polimorfismo."

Implementación Avanzada de Variaciones Protegidas

Java

```
// Sin protección contra variaciones
class PaymentProcessor {
    public void processPayment(Order order) {
        if (order.getPaymentMethod() == PaymentMethod.CREDIT_CARD) {
            // Procesamiento específico para tarjetas de crédito
            validateCreditCard(order.getCreditCardNumber());
            chargeCreditCard(order.getCreditCardNumber(), order.getTotal());
        }
        else if (order.getPaymentMethod() == PaymentMethod.PAYPAL) {
            // Procesamiento específico para PayPal
            redirectToPayPal(order.getCustomerEmail(), order.getTotal());
        }
        else if (order.getPaymentMethod() == PaymentMethod.BANK_TRANSFER) {
            // Procesamiento específico para transferencias bancarias
            generateBankTransferInstructions(order);
        }
        // Si se agrega un nuevo método de pago, esta clase debe modificarse
    }
}

// Con variaciones protegidas
interface PaymentStrategy {
    PaymentResult process(Order order);
}

class CreditCardPaymentStrategy implements PaymentStrategy {
    private final CreditCardProcessor processor;

    @Override
    public PaymentResult process(Order order) {
        CreditCard card = order.getCreditCard();
        if (!card.isValid()) {
            return PaymentResult.failed("Invalid credit card");
        }
    }
}
```

```
        try {
            TransactionId id = processor.charge(card, order.getTotal());
            return PaymentResult.successful(id);
        } catch (PaymentException e) {
            return PaymentResult.failed(e.getMessage());
        }
    }

}

class PayPalPaymentStrategy implements PaymentStrategy {
    private final PayPalClient payPalClient;

    @Override
    public PaymentResult process(Order order) {
        try {
            PayPalSession session = payPalClient.createSession(
                order.getCustomerEmail(),
                order.getTotal(),
                "Order #" + order.getId()
            );

            return PaymentResult.pending(session.getRedirectUrl(),
                session.getToken());
        } catch (PayPalException e) {
            return PaymentResult.failed(e.getMessage());
        }
    }
}

class BankTransferPaymentStrategy implements PaymentStrategy {
    private final BankAccountRepository bankAccountRepository;

    @Override
    public PaymentResult process(Order order) {
        BankAccount account = bankAccountRepository.getCompanyAccount();

        BankTransferInstructions instructions = new BankTransferInstructions(
            account.getNumber(),

```

```

        account.getBank(),
        account.getSwiftCode(),
        order.getTotal(),
        "Order #" + order.getId()
    );

    return PaymentResult.pending(instructions);
}
}

// Factory para crear la estrategia adecuada

class PaymentStrategyFactory {
    private final Map<PaymentMethod, PaymentStrategy> strategies = new
HashMap<>();

    public PaymentStrategyFactory(
        CreditCardProcessor creditCardProcessor,
        PayPalClient payPalClient,
        BankAccountRepository bankAccountRepository) {
        strategies.put(PaymentMethod.CREDIT_CARD,
            new CreditCardPaymentStrategy(creditCardProcessor));
        strategies.put(PaymentMethod.PAYPAL,
            new PayPalPaymentStrategy(payPalClient));
        strategies.put(PaymentMethod.BANK_TRANSFER,
            new BankTransferPaymentStrategy(bankAccountRepository));
    }

    public PaymentStrategy getStrategy(PaymentMethod method) {
        PaymentStrategy strategy = strategies.get(method);
        if (strategy == null) {
            throw new UnsupportedPaymentMethodException(method);
        }
        return strategy;
    }

// Método para registrar nuevas estrategias

public void registerStrategy(PaymentMethod method, PaymentStrategy strategy) {
    strategies.put(method, strategy);
}

```

```

    }
}

// Procesador de pagos que utiliza estrategias
class PaymentProcessor {
    private final PaymentStrategyFactory strategyFactory;

    public PaymentProcessor(PaymentStrategyFactory strategyFactory) {
        this.strategyFactory = strategyFactory;
    }

    public PaymentResult processPayment(Order order) {
        PaymentStrategy strategy =
strategyFactory.getStrategy(order.getPaymentMethod());
        return strategy.process(order);
    }
}

```

Beneficios de Variaciones Protegidas

- Extensibilidad sin modificación** : Nuevas variantes requieren implementar interfaces existentes.
- Aislamiento de cambios** : Los cambios en una variante no afectan a otras variantes.
- Integración tecnológica más sencilla** : Facilita la incorporación de nuevas tecnologías o servicios.
- Composición dinámica** : Permite configurar o cambiar comportamientos en el tiempo de ejecución.

Actividad de evaluación

Ejercicio: Análisis Crítico y Rediseño de un Sistema de Gestión de Biblioteca

Plantación

Usted ha sido contratado como consultor para evaluar y mejorar el diseño de un sistema de gestión para una biblioteca universitaria. El sistema actual fue desarrollado hace algunos años y presenta problemas de mantenibilidad, extensibilidad y rendimiento.

Se le proporciona un diagrama UML simplificado del sistema actual y fragmentos de código que ilustran algunas de sus funcionalidades principales. Su tarea consiste en realizar un análisis crítico identificando las violaciones a los principios de diseño estudiados en la unidad, y proponer un rediseño que mejore la estructura del sistema siguiendo los principios SOLID, GRASP y otros fundamentos de diseño orientado a objetos.

Sistema Actual

Diagrama de Clases Simplificadas:

```
Library
└── Book
└── User
└── Loan
└── LibrarySystem
    ├── processLoan()
    ├── returnBook()
    ├── searchBooks()
    ├── notifyUser()
    ├── generateReports()
    ├── manageUsers()
    └── calculateFines()
```

Fragментos de código:

```
Java
// Clase principal del sistema
class LibrarySystem {
    private Database db;
    private List<Book> books;
    private List<User> users;
    private List<Loan> loans;
    private EmailService emailService;

    // Constructor inicializa todo directamente
    public LibrarySystem() {
            db = new MySQLDatabase();
            loadData();
            emailService = new EmailService();
    }

    private void loadData() {
            books = db.executeQuery("SELECT * FROM books");
            users = db.executeQuery("SELECT * FROM users");
            loans = db.executeQuery("SELECT * FROM loans");
    }
```

```
// Proceso de préstamo

public void processLoan(int userId, int bookId) {
    User user = null;
    Book book = null;

    // Búsqueda manual en las listas
    for (User u : users) {
        if (u.getId() == userId) {
            user = u;
            break;
        }
    }

    for (Book b : books) {
        if (b.getId() == bookId) {
            book = b;
            break;
        }
    }

    if (user == null || book == null) {
        throw new IllegalArgumentException("Usuario o libro no encontrado");
    }

    // Verificaciones directas
    if (user.getLoans().size() >= 5) {
        throw new LoanLimitExceeded();
    }

    if (book.getStatus() != BookStatus.AVAILABLE) {
        throw new BookNotAvailableException();
    }

    // Creación y persistencia de préstamo
    Loan loan = new Loan();
    loan.setBook(book);
    loan.setUser(user);
```

```

loan.setLoanDate(new Date());
loan.setDueDate(calculateDueDate());

// Actualización directa del estado del libro
book.setStatus(BookStatus.LOANED);

// Persistencia
db.executeUpdate("INSERT INTO loans VALUES (...)", loan);
db.executeUpdate("UPDATE books SET status = 'LOANED' WHERE id = ?",
bookId);

// Notificación por email
String message = "Dear " + user.getName() + ", you have borrowed " +
                book.getTitle() + ". Due date: " + loan.getDueDate();
emailService.sendEmail(user.getEmail(), "Book Loan Confirmation",
message);

// Búsqueda de Libros
public List<Book> searchBooks(String criteria, SearchType type) {
    List<Book> results = new ArrayList<>();

    for (Book book : books) {
        if (type == SearchType.TITLE && book.getTitle().contains(criteria)) {
            results.add(book);
        } else if (type == SearchType.AUTHOR &&
book.getAuthor().contains(criteria)) {
            results.add(book);
        } else if (type == SearchType.ISBN && book.getIsbn().equals(criteria))
{
            results.add(book);
        } else if (type == SearchType.CATEGORY &&
book.getCategory().equals(criteria)) {
            results.add(book);
        }
    }

    return results;
}

```

```

// Cálculo de multas
public double calculateFines(Loan loan) {
    if (loan.getReturnDate() == null) {
        // Préstamo aún activo
        return 0;
    }

    long daysLate = (loan.getReturnDate().getTime() -
loan.getDueDate().getTime()) /
                    (24 * 60 * 60 * 1000);

    if (daysLate <= 0) {
        return 0;
    }

// Lógica de cálculo de multas
    if (loan.getBook().getCategory().equals("RESERVED")) {
        return daysLate * 2.0; // $2 por día para libros reservados
    } else {
        return daysLate * 1.0; // $1 por día para libros normales
    }
}

// Otras funcionalidades...
}

// Clase Book con responsabilidades limitadas
class Book {
    private int id;
    private String title;
    private String author;
    private String isbn;
    private String category;
    private BookStatus status;

// Getters y setters
}

```

```

// Clase User con responsabilidades limitadas
class User {
    private int id;
    private String name;
    private String email;
    private UserType type;
    private List<Loan> loans;

    // Getters y setters
}

// Clase Loan como simple contenedor de datos
class Loan {
    private int id;
    private User user;
    private Book book;
    private Date loanDate;
    private Date dueDate;
    private Date returnDate;

    // Getters y setters
}

```

Requisitos de la tarea

- 1. Análisis crítico :**
 - Identificar al menos 5 violaciones específicas a los principios SOLID.
 - Identificar problemas relacionados con asociación, cohesión y asignación de responsabilidades.
 - Analizar limitaciones de extensibilidad y mantenibilidad del diseño actual.
- 2. Propuesta de Rediseño :**
 - Crea un nuevo diagrama de clases con la estructura mejorada.
 - Implementar al menos dos patrones de diseño relevantes para el problema.
 - Aplique los principios SOLID y GRASP en el rediseño.
 - Justificar técnicamente las decisiones de diseño tomadas.
- 3. Demostración de mejoras :**
 - Reimplementar la funcionalidad de préstamo de libros con el nuevo diseño.
 - Demostrar cómo el nuevo diseño facilita la extensión para incorporar:
 - Diferentes tipos de recursos (libros físicos, e-books, revistas, etc.)
 - Diversos métodos de notificación (correo electrónico, SMS, aplicación móvil)

- Políticas variables de préstamo por tipo de usuario o recurso

Entregables

- Documento de análisis :**
 - Identificación y explicación detallada de problemas en el diseño original.
 - Principios violados y consecuencias potenciales.
- Documentación del Rediseño :**
 - Diagrama de clases UML del nuevo diseño.
 - Justificación técnica de las decisiones de diseño.
 - Explicación de patrones implementados.
- Código de demostración :**
 - Implementación del procesamiento de préstamos con el nuevo diseño.
 - Ejemplos de extensibilidad para nuevos tipos de recursos, notificaciones y políticas.

Rúbrica de evaluación

Criterio	Nivel Insuficiente (0-3)	Nivel básico (4-6)	Nivel avanzado (7-8)	Nivel experto (9-10)
Identificación de problemas de diseño	Identifica menos de 3 problemas sin fundamento técnico	Identifica al menos 3 problemas con fundamento parcial	Identifica 5 o más problemas con buen fundamento técnico	Identifica exhaustivamente los problemas con fundamento sólido en principios OO
Aplicación de principios SOLID	Aplica incorrectamente los principios o menos de 3	Aplica correctamente al menos 3 principios SOLID	Aplica correctamente todos los principios SOLID	Aplica magistralmente todos los principios con justificación excepcional.
Rediseño en UML	Diagrama incompleto o con errores graves	Diagrama completo con estructura mejorada	Diagrama bien estructurado con patrones adecuados	Diagrama profesional con excelente aplicación de patrones y principios.
Implementación de código	Código con errores o sin mejoras sustanciales	Código funcional con algunas mejoras de diseño.	Código bien estructurado que demuestra las mejoras	Código ejemplar que demuestra excelencia en diseño OO
Extensibilidad de la solución	La solución no demuestra mejoras en extensibilidad.	La solución permite algunas extensiones con modificaciones.	La solución facilita extensiones con mínimas modificaciones.	La solución permite extensiones sin modificar código existente.

Criterio	Nivel Insuficiente (0-3)	Nivel básico (4-6)	Nivel avanzado (7-8)	Nivel experto (9-10)
Justificación técnica	Justificación ausente o incorrecta	Justificación basica de decisiones principales	Justificación sólida con referencias a principios	Justificación excepcional con análisis de alternativas.

Conclusión

En esta unidad hemos explorado los fundamentos avanzados del diseño orientado a objetos, comenzando con los pilares básicos de abstracción y encapsulamiento, avanzando hacia conceptos más atractivos como los principios SOLID y GRASP. Hemos visto cómo estos principios no son simplemente reglas abstractas, sino herramientas prácticas que nos permiten crear sistemas más mantenibles, extensibles y robustos.

El diseño orientado a objetos efectivo no se trata solo de aplicar patrones o principios mecánicamente, sino de desarrollar un entendimiento profundo de cómo asignar responsabilidades, gestionar dependencias y modelar el dominio del problema de manera coherente. A medida que los sistemas de software crecen en complejidad, la aplicación juiciosa de estos principios se vuelve cada vez más crítica para su éxito a largo plazo.

La evaluación crítica de diseños existentes y la capacidad para proponer mejoras fundamentadas son habilidades esenciales para cualquier ingeniero de software avanzado. Esperamos que esta unidad haya proporcionado no solo el conocimiento técnico necesario, sino también la perspectiva para aplicar estos conceptos de manera efectiva en su práctica profesional.