UNIVERSIDAD
Popular del cesar

Ingeniería de Sistemas

ESPECIALIZACION EN INGENIERIA DE SOFTWARE
MODULO PATRONES DE DISEÑO DE SOFTWARE

EL DOCENTE

JAIRO FRANCISCO SEOANES LEON

jairoseoanes@unicesar.edu.co
(300) 600 06 70

Educación formal

- ✓ **Ingeniero de sistemas**, Universidad Popular del Cesar sede Valledupar, Feb 2002 – Jun 2009.
- ✓ **MsC en Ingeniería de Sistemas y Computación**, Universidad Nacional de Colombia, Bogotá, Feb 2011 – Mar 2015
- ✓ **PhD Ciencia, Tecnologia e innovación, Urbe, Venezuela, Mayo 2024**

Formación complementaria

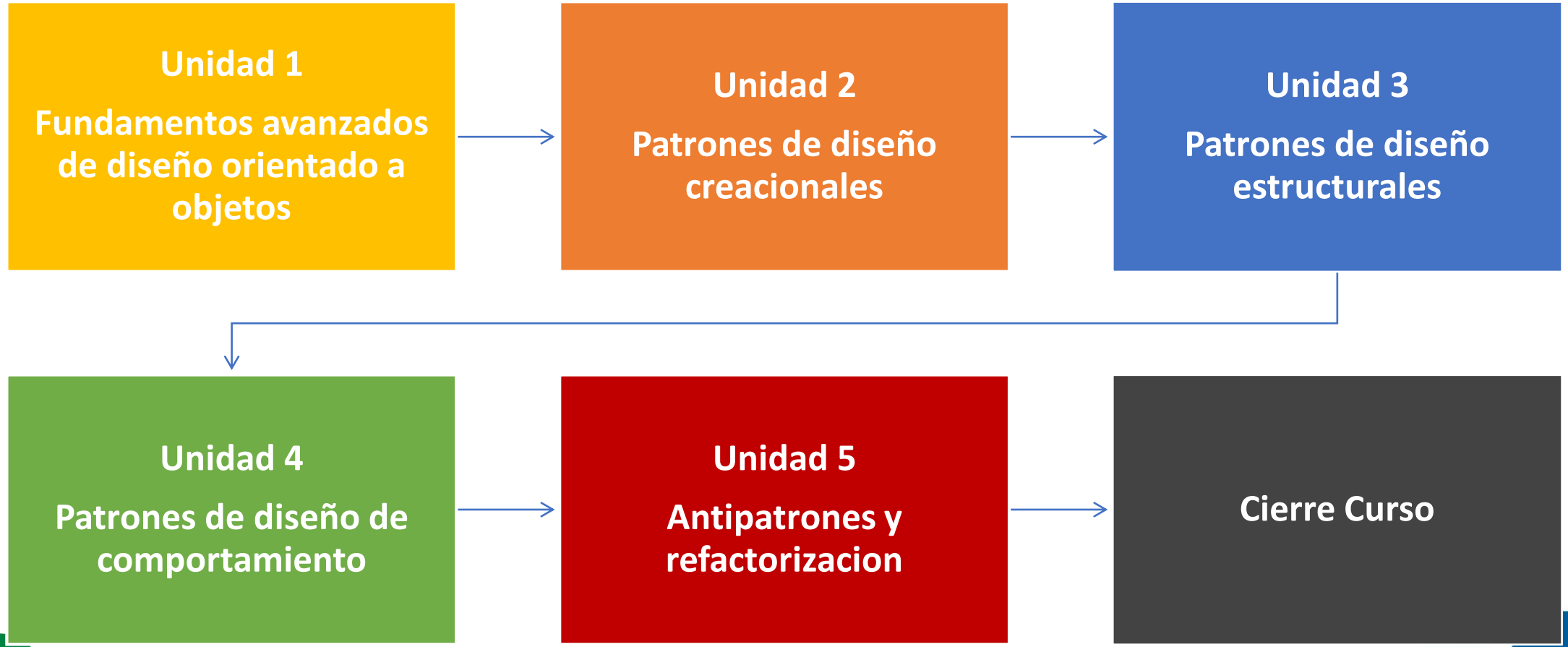
- ✓ **AWS Academy Graduate -** AWS Academy Cloud Foundations, 2022
  https://www.credly.com/go/p3Uwht36
- ✓ **Associate Cloud Engineer Path -** Google Cloud Academy, 2022
  https://www.cloudskillsboost.google/public_profiles/c7e7936c-3e37-4bad-b822-74d40c49d0db
- ✓ **Fundamentos De Programación Con Énfasis En Cloud Computing –** AWS Academy y Misión Tic 2022
- ✓ **Google Cloud Computing Foundations –** Google Academy, 2022
- ✓ **Aplicación de cloud: retos y oportunidades de mejora para las empresas de software gestionando la computación en la nube –** Fedesoft, 2023
- ✓ **Desarrollo De Aplicaciones Web En Angular, Para El Nivel Frontend** – Universidad EAFIT, 2023
- ✓ **Microsoft Scrum Foundations –** Intelligent Training - MinTic , 2023

Experiencia profesional

- ✓ **Docente Universitario**, Universidad Popular del Cesar sede Valledupar, marzo del 2013.
- ✓ **Técnico de Sistemas Grado 11**, Rama judicial Seccional Cesar, SRPA Valledupar, Junio del 2009

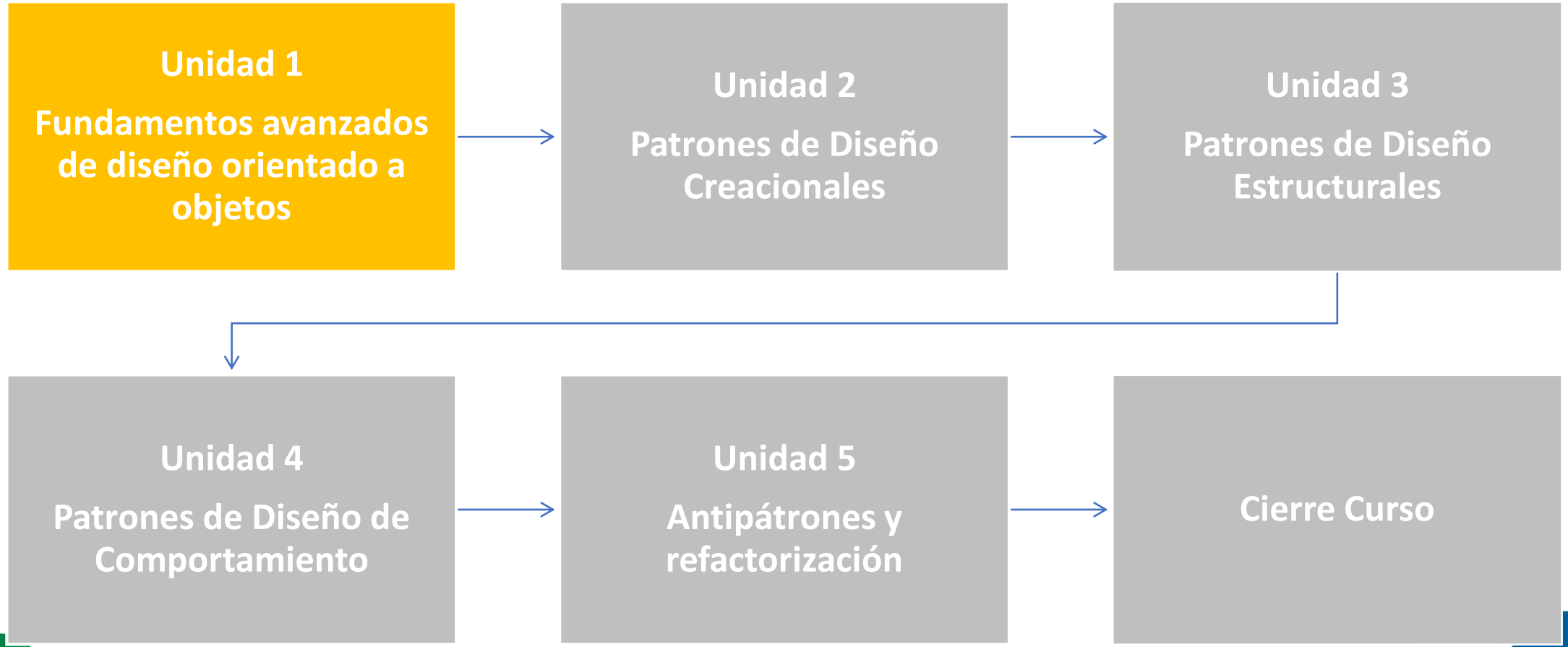
UNIVERSIDAD Popular del cesar

MODULO DE PATRONES DE DISEÑO DE SOFTWARE



UNIVERSIDAD Popular del cesar

MODULO DE PATRONES DE DISEÑO DE SOFTWARE



UNIVERSIDAD Popular del cesar

Unidad 1. Fundamentos avanzados de diseño O.O

UNIVERSIDAD Popular del cesar

Introducción - Motivación

“ Un código elegante no es aquel que tiene menos líneas, sino el que saca mayor provecho de ellas ”

Oscar Blancharte (2016)

UNIVERSIDAD Popular del cesar

Introducción – Que es un buen diseño

¿ Características de un buen diseño ?

Cosas que buscar

- Reutilización de código
- Extensibilidad

Cosas que evitar

- Antipatrones

Introducción – Que es un buen diseño

Reutilización de código

Reutilizar el código en nuevos proyectos

Costo y tiempo

Requiere un esfuerzo exigente

Reutilización de nivel bajo (Clases)

Nivel intermedio (patrones)

Reutilización de alto nivel (Frameworks)

Extensibilidad

El cambio es lo único constante en la vida de un programador

Comprendemos mejor el problema una vez que comenzamos a resolverlo

Algo fuera de tu control ha cambiado.

Los postes de la portería se mueven

Principios de diseño

¿ Que es un buen diseño de software?

¿ Como evaluamos la calidad de un buen diseño ?

¿ Que practicas debo llevar a cabo para un buen diseño?

¿ Como hacer una arquitectura flexible, estable, fácil de comprender?

Principios Universales del Diseño de Software

- Encapsula lo que varía
- Programa a una interfaz, no a una implementación
- Favorece la composición sobre la herencia

SOLID - GRASP

Principios universales de diseño

Encapsula lo que varía

Identifica los aspectos de tu aplicación que varían y sepáralos de los que se mantienen inalterables.

Objetivo:
Minimizar el efecto provocado por los cambios

Proteger el resto del código frente a efectos adversos

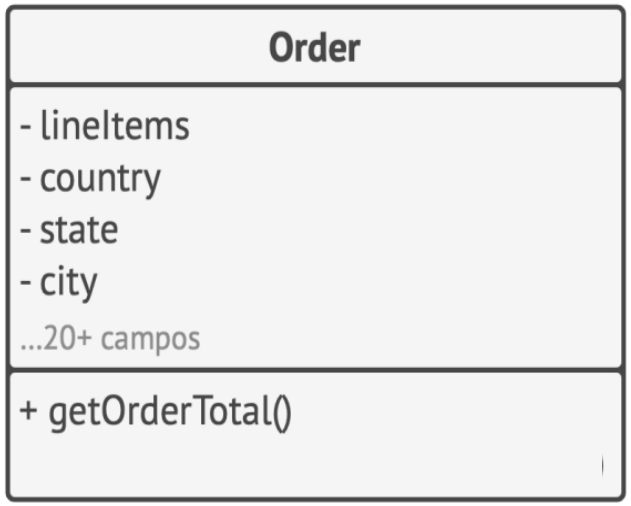
Menos tiempo para lograr que el programa vuelva a funcionar, al implementar y probar cambios.

Encapsulación a nivel de métodos

Encapsulación a nivel de clases

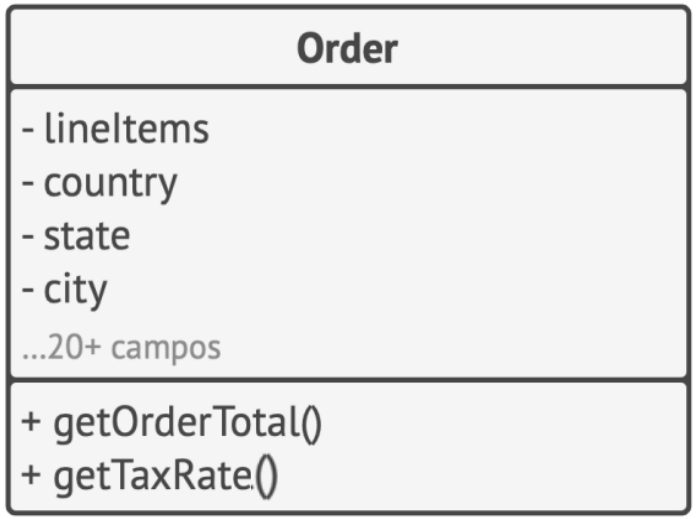
Principios universales de diseño

Encapsulación a nivel de métodos



El método **getOrderTotal** que calcula un total del pedido, impuestos incluido.

el código relacionado con el calculo de los impuestos tendrá que cambiar en el futuro



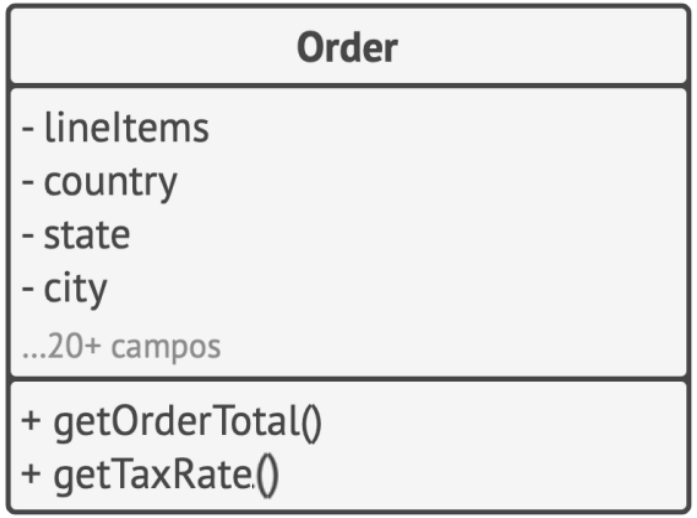
Encapsula lo que varía

Identifica los aspectos de tu aplicación que varían y sepáralos de los que se mantienen inalterables.

Puedes extraer la lógica de cálculo del impuesto a un método separado, escondiéndolo del método original

Principios universales de diseño

Encapsulación a nivel de clases

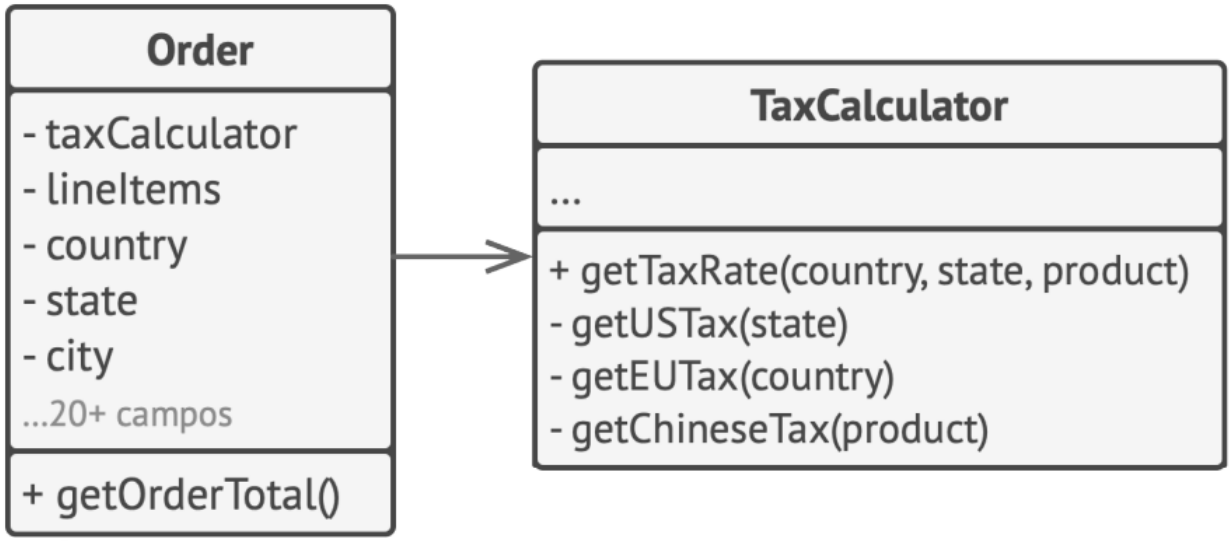


Con el tiempo puedes añadir más y más responsabilidades a un método que solía hacer algo sencillo

Encapsula lo que varía

Identifica los aspectos de tu aplicación que varían y sepáralos de los que se mantienen inalterables.

Si se extrae todo a una nueva clase se puede conseguir mayor claridad y sencillez.



Los objetos de la clase Pedido delegan todo el trabajo relacionado con el impuesto a un objeto especial dedicado justo a eso.

Principios universales de diseño

Programa a una interfaz, no a una implementación

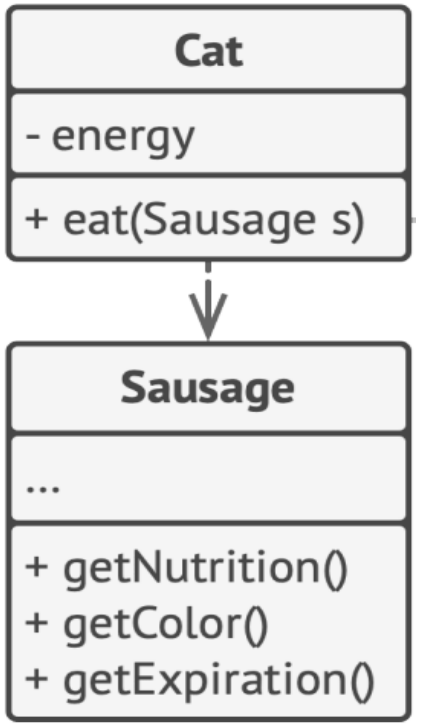
Depende de abstracciones y no de clases concretas

Objetivo:
Lograr diseño flexible, fácil de extender, sin requerir descomponer el código existente

Eliminar dependencia entre clases, cuando estas colaboren

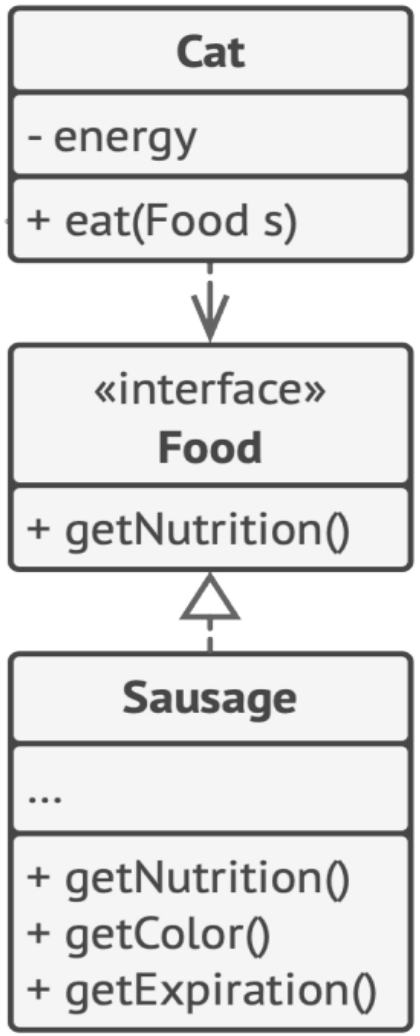
1. Determina lo que necesita exactamente un objeto del otro
2. Define los métodos en una nueva interfaz o clase abstracta
3. Haz que la clase que es una dependencia implemente esta interfaz
4. Ahora, haz la segunda clase dependiente de esta interfaz en lugar de la clase concreta.

Principios universales de diseño



La clase **Cat** se limita únicamente a la uso de la clase **Sausage**

Cualquier cambio que se realice en la interfaz afectara a la clase dependiente



Programa a una interfaz, no a una implementación

Depende de abstracciones y no de clases concretas

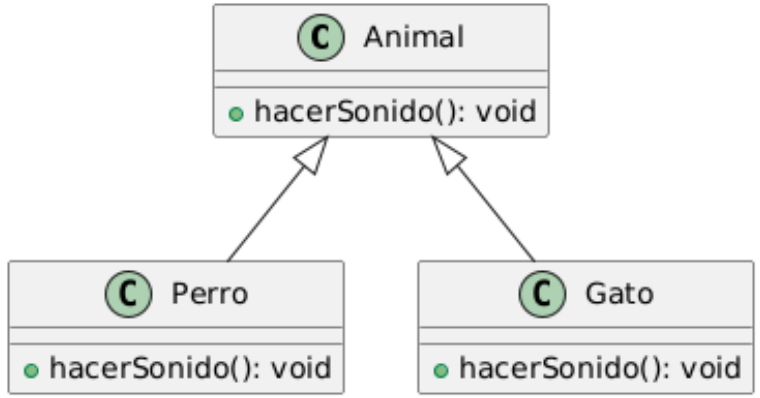
Con la dependencia a la interfaz, la conexión es mucho más flexible.

Principios universales de diseño

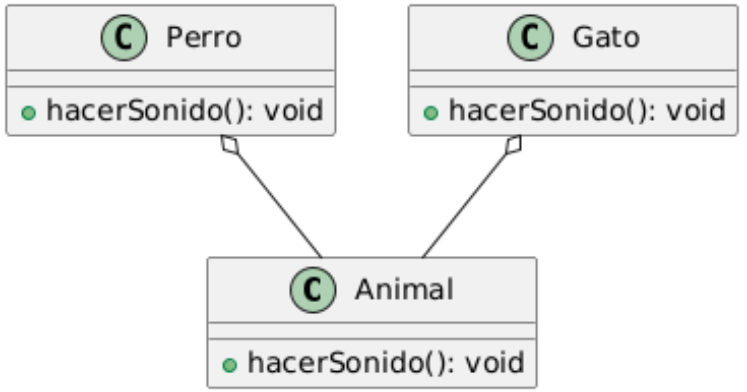
Favorece la composición sobre la herencia

La herencia a pesar de sus beneficios, también tiene sus contras, que a menudo no resultan aparentes

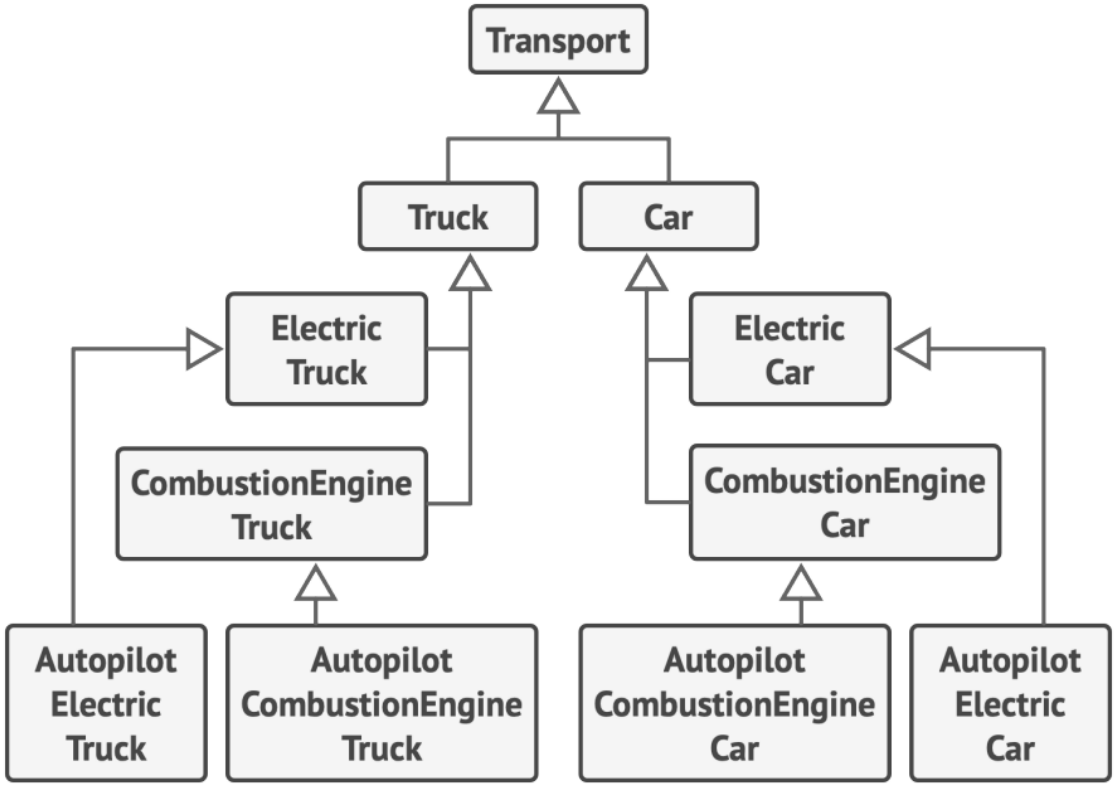
HERENCIA



COMPOSICION



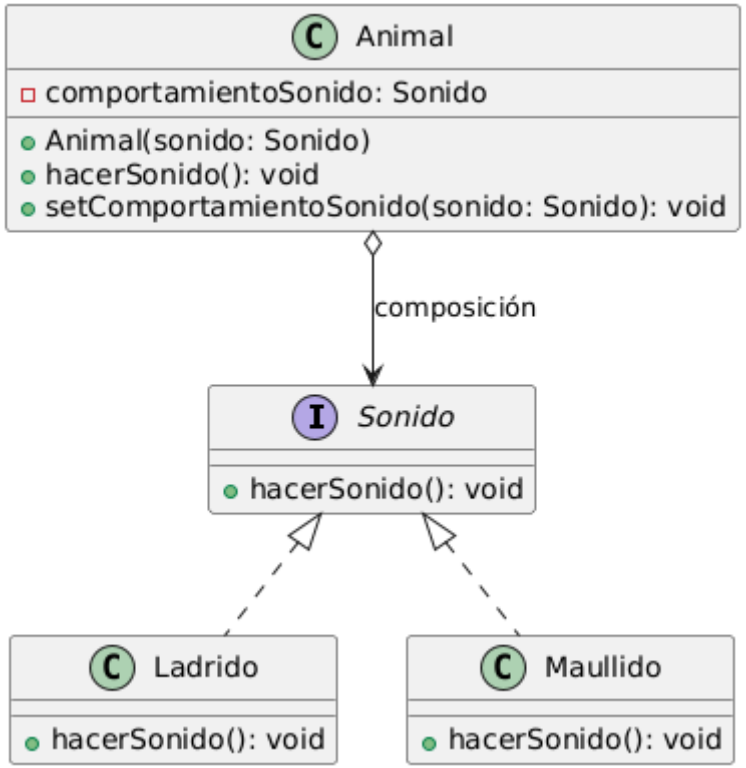
Principios universales de diseño



Problemas asociado a la herencia:

1. Una subclase no puede reducir la interfaz de la superclase
2. La herencia rompe la encapsulación de la superclase
3. Las subclases están fuertemente acopladas a superclases
4. Jerarquías de herencia paralelas

Principios universales de diseño



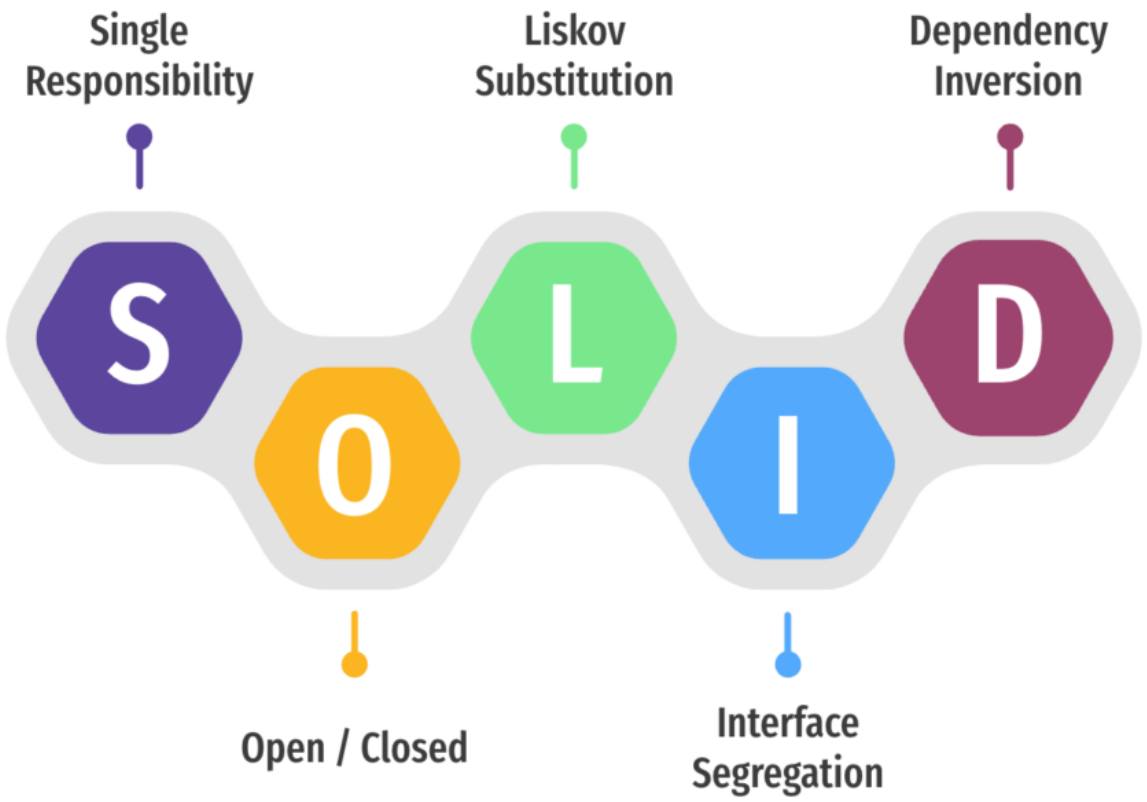
Beneficios de la composición:

1. Permite construir objetos a partir de componentes reutilizables.
2. Ofrece mayor flexibilidad y menor acoplamiento
3. Facilita el cambio de comportamiento en tiempo de ejecución

Principios universales de diseño

Aspecto	Herencia	Composición
Reutilización	A través de la superclase	A través de objetos internos (atributos)
Acoplamiento	Fuerte (subclase depende de superclase)	Bajo (cada clase puede cambiar independientemente)
Flexibilidad	Limitada (herencia es estática)	Alta (puedes cambiar el comportamiento en tiempo de ejecución)
Cambios en la base	Afectan a todas las subclases	No afectan a otras clases directamente

Principios SOLID



Cinco **principios de diseño** ideados para hacer que los diseños de software sean más **comprensibles**, **flexibles** y **fáciles de mantener**.

"Aspirar a estos principios es bueno, pero intenta siempre ser pragmático y no tomes todo lo escrito aquí como un dogma"

A. Shvets

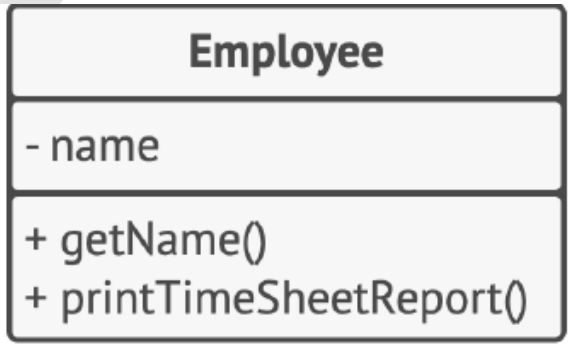
Robert Martin los presentó en el libro Desarrollo ágil de software: principios, patrones y prácticas.

Principios SOLID

Simple Responsability Principle

Principio de responsabilidad única

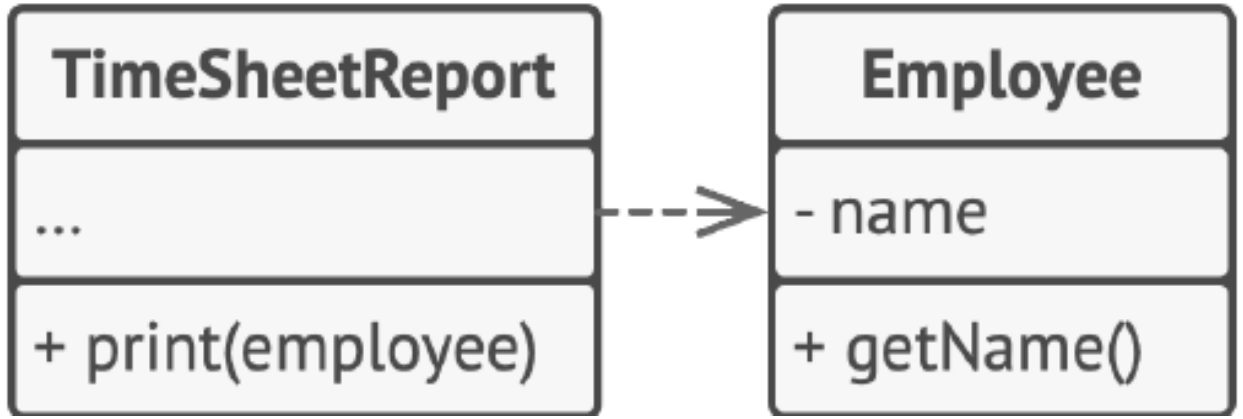
“ Una clase sólo debe tener una razón para cambiar. “



La clase contiene varios comportamientos diferentes

“varias razones para cambiar”

- ✓ El principal objetivo de este principio es reducir la complejidad
- ✓ Si una clase hace demasiadas cosas, tienes que cambiarla cada vez que una de esas cosas cambia



El comportamiento adicional está en su propia clase.

Principios SOLID

Simple Responsability Principle

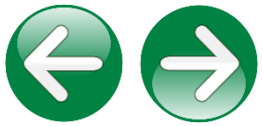
Principio de responsabilidad única

```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 incorrecta SRP

Aplicación correcta SRP

```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 */);
    }
}
```



Principios SOLID

Simple Responsability Principle

Principio de responsabilidad única

Aplicación correcta SRP

```
// 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 */);
    }
}
```

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.

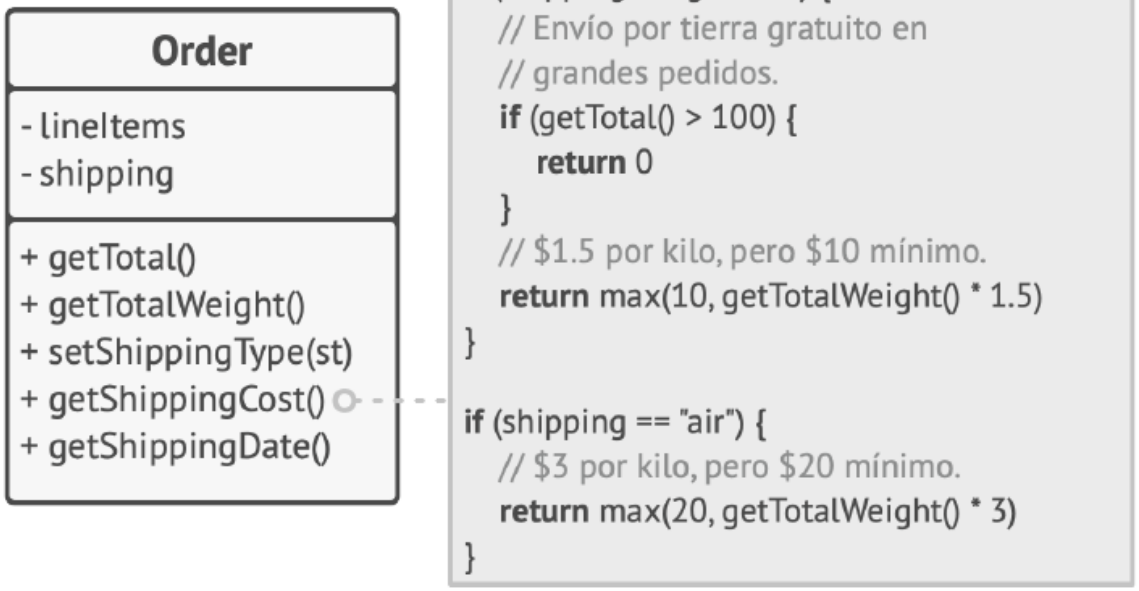
Mejora la testeabilidad: Las responsabilidades aisladas se prueban de forma independiente.

Principios SOLID

Open / Close Principle

Principio de abierto / cerrado

> "Las entidades software (clases, módulos, funciones…) deben estar abiertas para su extensión, pero cerradas para su modificación"



- ✓ Evitar que el código existente se descomponga cuando implementas nuevas funciones.

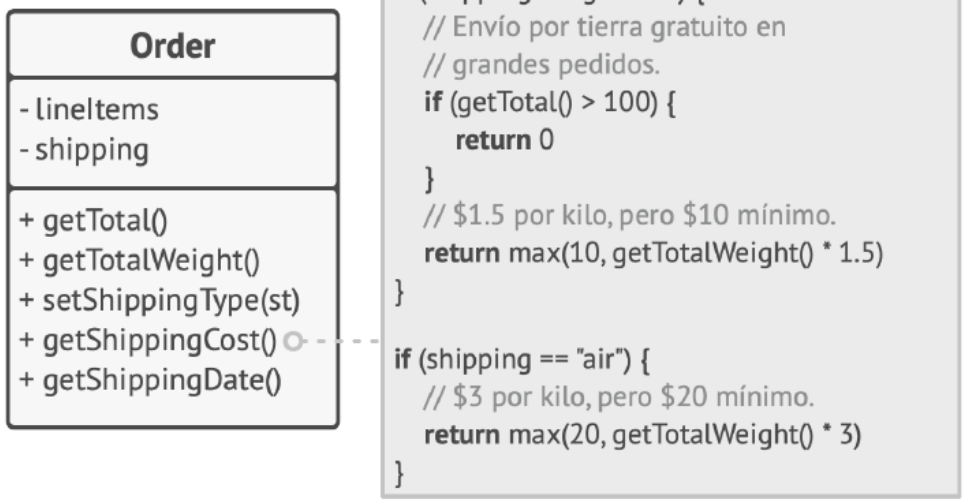
Tienes que cambiar la clase **Order** siempre que añades un nuevo método de envío a la aplicación

Principios SOLID

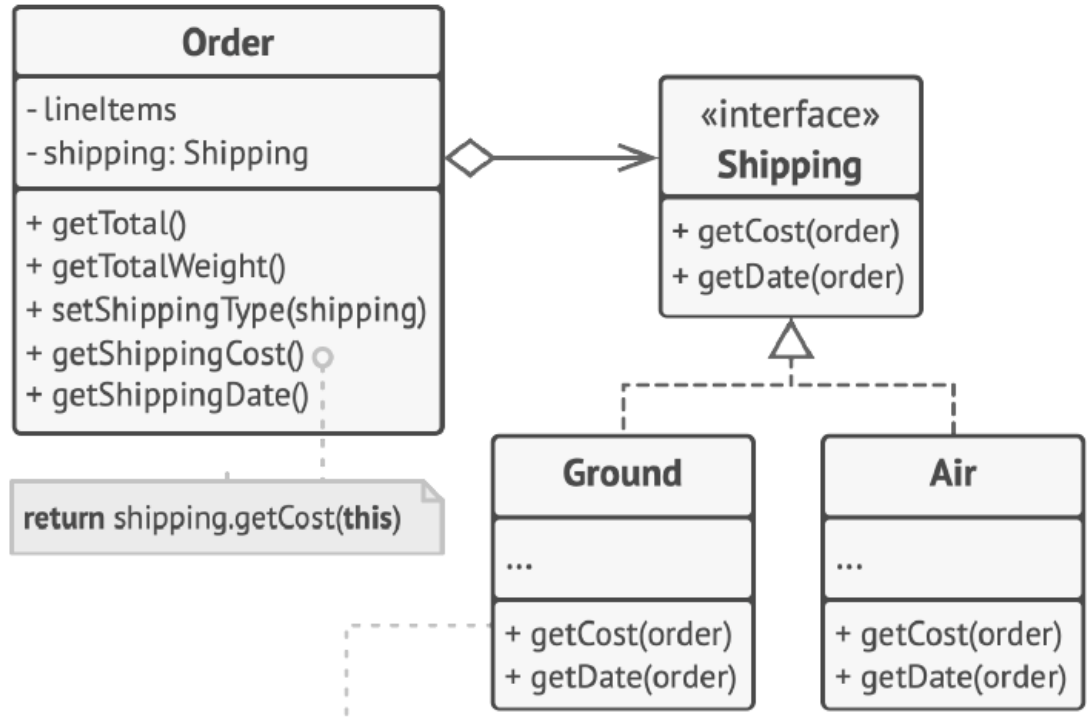
Open / Close Principle

Principio de abierto / cerrado

Tienes que cambiar la clase **Order** siempre que añades un nuevo método de envío a la aplicación




```
if (shipping == "ground") {
    // Envío por tierra gratuito en
    // grandes pedidos.
    if (getTotal() > 100) {
        return 0
    }
    // $1.5 por kilo, pero $10 mínimo.
    return max(10, getTotalWeight() * 1.5)
}

if (shipping == "air") {
    // $3 por kilo, pero $20 mínimo.
    return max(20, getTotalWeight() * 3)
}
```




```
return shipping.getCost(this)
```

```
// Envío por tierra gratuito en
// grandes pedidos.
if (order.getTotal() > 100) {
    return 0
}
// $1.5 por kilo, pero $10 mínimo.
return max(10, order.getTotalWeight() * 1.5)
```

Añadir un nuevo método de envío no requiere cambiar clases existentes

UNIVERSIDAD Popular del cesar

Principios SOLID

Open / Close Principle

Principio de abierto / cerrado

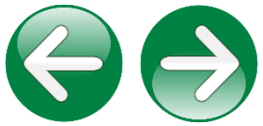
```java
// Violación del OCP
class OrderProcessor {
    public void process(Order order) {
        if (order.getType() == OrderType.RETAIL) {
            // Lógica para pedidos minoristas
        } else if (order.getType() == OrderType.WHOLESALE) {
            // Lógica para pedidos mayoristas
        } else if (order.getType() == OrderType.INTERNATIONAL) {
            // Lógica para pedidos internacionales
        }
        // Si agregamos un nuevo tipo de pedido, tenemos que modificar esta clase
    }
}
```

Violación OCP

```java
// 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
    }
}
```

```java
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);
    }
}
```

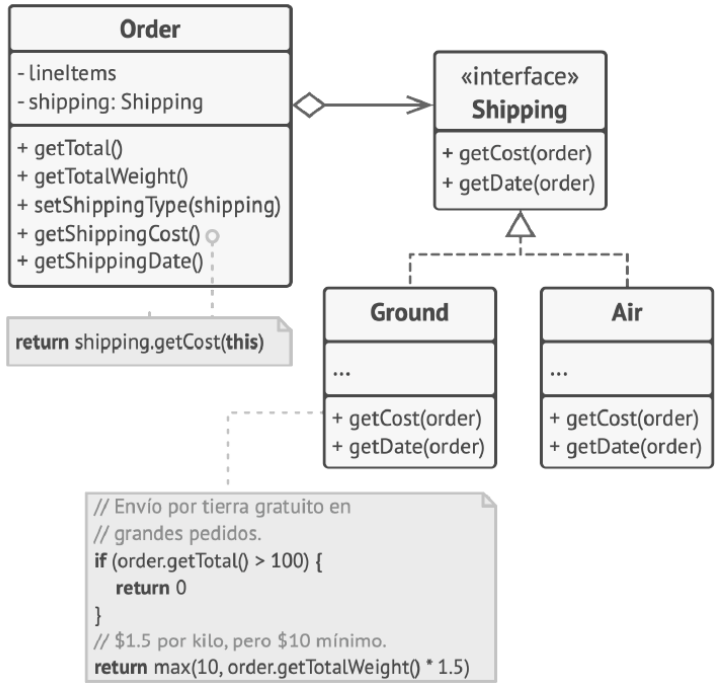
Aplicación Correcta OCP



Principios SOLID

O Open / Close Principle

Principio de abierto / cerrado

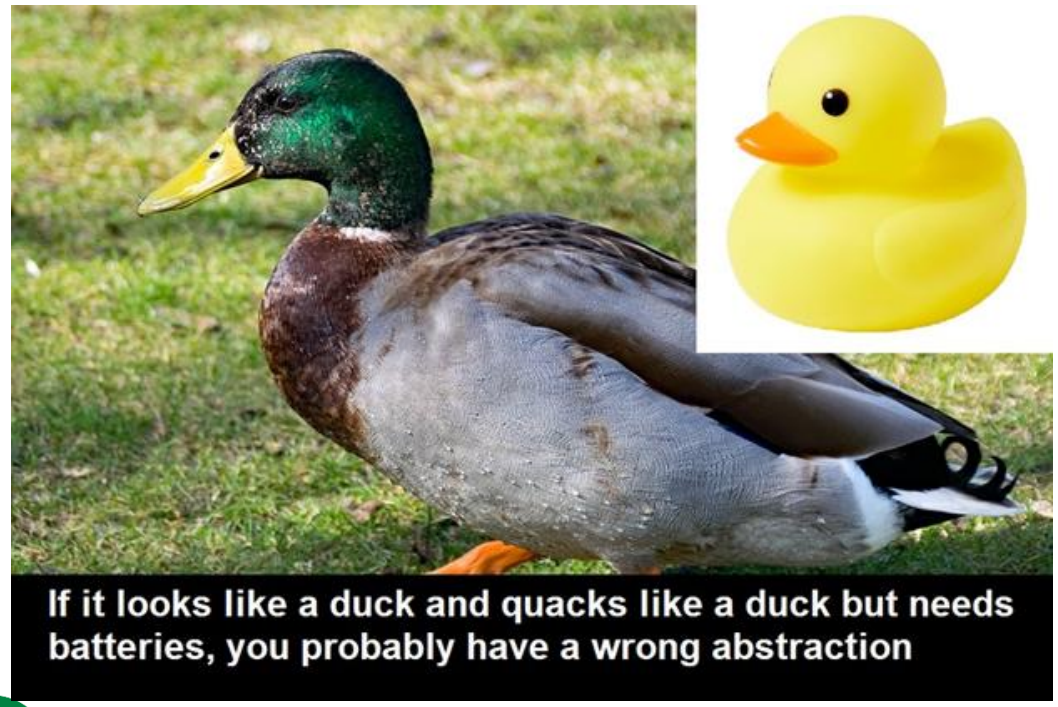


Beneficio Estratégico	Descripción
1. Mayor mantenibilidad	El código es más fácil de mantener porque los cambios no afectan lo ya probado.
2. Reducción del riesgo de errores	Al no tocar el código existente, se minimiza el riesgo de introducir fallos.
3. Escalabilidad y adaptabilidad	Se pueden añadir nuevas funcionalidades sin romper otras partes del sistema.
4. Mejora la reutilización	Los módulos bien diseñados pueden reutilizarse en otros contextos.
5. Aumenta la estabilidad del sistema	Las partes estables del sistema permanecen intactas mientras se extiende.
6. Facilita pruebas unitarias y TDD	Las clases nuevas se pueden probar de forma aislada; el sistema es más testable.
7. Favorece el diseño basado en interfaces	Esto fomenta bajo acoplamiento y alto grado de abstracción.
8. Habilita el uso de patrones de diseño	Patrones como Strategy, Decorator, Factory y Observer dependen del OCP.
9. Reduce el costo a largo plazo	Un sistema extensible requiere menos esfuerzo y dinero para evolucionar.

Principios SOLID

Liskov Substitution Principle

Principio de sustitución de Liskov



" Si parece un pato, grazna como un pato, pero necesita pilas: estás en la abstracción equivocada "

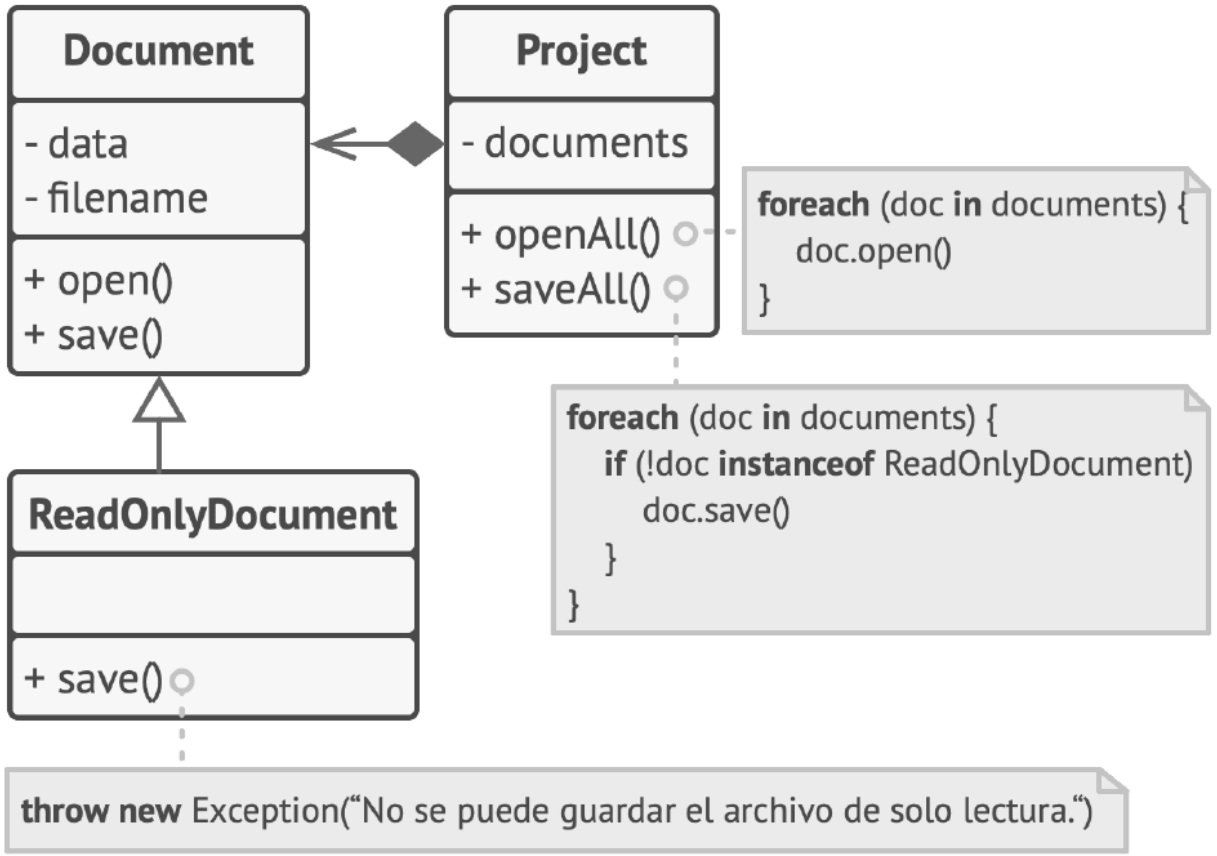
"Al extender una clase, recuerda que debes tener la capacidad de pasar objetos de las subclases en lugar de objetos de la clase padre, sin descomponer el código cliente"

- ✓ Esto significa que la subclase debe permanecer compatible con el comportamiento de la superclase.
- ✓ Otras personas utilizarán tus clases y no podrás acceder directamente ni cambiar su código
- ✓ Extiende el comportamiento, en vez de sustituirlo por completo

Principios SOLID

Liskov Substitution Principle

Principio de sustitución de Liskov



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

Las subclases deben respetar el contrato de su superclase

Principios SOLID

Liskov Substitution Principle

Principio de sustitución de Liskov

Requisitos formales:

1. Los parámetros de métodos de una subclase deben coincidir o ser más abstractos que los de la superclase.

2. El tipo de retorno de métodos de la subclase debe coincidir o ser un subtipo de los de la superclase.

3. Un método de subclase no debe arrojar excepciones que no se espere que arroje el método base

4. Una subclase no debe fortalecer las condiciones previas.

5. Una subclase no debe debilitar las condiciones posteriores

UNIVERSIDAD Popular del cesar

Principios SOLID

L Liskov Substitution Principle

Principio de sustitución de Liskov

1. Los parámetros de métodos de una subclase deben coincidir o ser más abstractos que los de la superclase.

```
class Alimentador {
    void alimentar(Animal animal) {
        System.out.println("Alimentando un animal");
    }
}

class AlimentadorPerros extends Alimentador {
    @Override
    void alimentar(Perro perro) { // ❌ más específico
        System.out.println("Alimentando un perro");
    }
}
```

```
class Alimentador {
    void alimentar(Animal animal) {
        System.out.println("Alimentando un animal");
    }
}

class AlimentadorEspecial extends Alimentador {
    @Override
    void alimentar(Animal animal) {
        System.out.println("Alimentando especial a un animal");
    }
}
```

Principios SOLID

Liskov Substitution Principle

Principio de sustitución de Liskov

2. El tipo de retorno de métodos de la subclase debe coincidir o ser un subtipo de los de la superclase.

```
class Figura {
    Figura clonar() {
        return new Figura();
    }
}

class Circulo extends Figura {
    @Override
    Object clonar() { // tipo más general ✗ rompe LSP
        return new Circulo();
    }
}
```

```
class Figura {
    Figura clonar() {
        return new Figura();
    }
}

class Circulo extends Figura {
    @Override
    Circulo clonar() { // subtipo del tipo de retorno ✔
        return new Circulo();
    }
}
```

Principios SOLID

Liskov Substitution Principle

Principio de sustitución de Liskov

3. Un método de subclase no debe arrojar excepciones que no se espere que arroje el método base

```
class Procesador {
    void procesar() throws IOException {
        System.out.println("Procesando desde la superclase");
    }
}

class ProcesadorArchivo extends Procesador {
    @Override
    void procesar() throws FileNotFoundException { // ✅ más específica
        System.out.println("Procesando archivo");
    }
}
```

La subclase no debe lanzar nuevas excepciones que no estén especificadas por la superclase.

```
class Super {
    void procesar() throws IOException {}
}

class Sub extends Super {
    @Override
    void procesar() throws Exception {} // ❌ más general
}
```

Principios SOLID

Liskov Substitution Principle

Principio de sustitución de Liskov

4. Una subclase no debe fortalecer las condiciones previas.

La subclase no puede imponer más condiciones que la superclase

```
class Super {
    void operar(int x) {
        // acepta cualquier x
    }
}

class Sub extends Super {
    @Override
    void operar(int x) {
        if (x < 0) throw new IllegalArgumentException(); // ❌ violación
    }
}
```

Principios SOLID

Liskov Substitution Principle

Principio de sustitución de Liskov

5. Una subclase no debe debilitar las condiciones posteriores

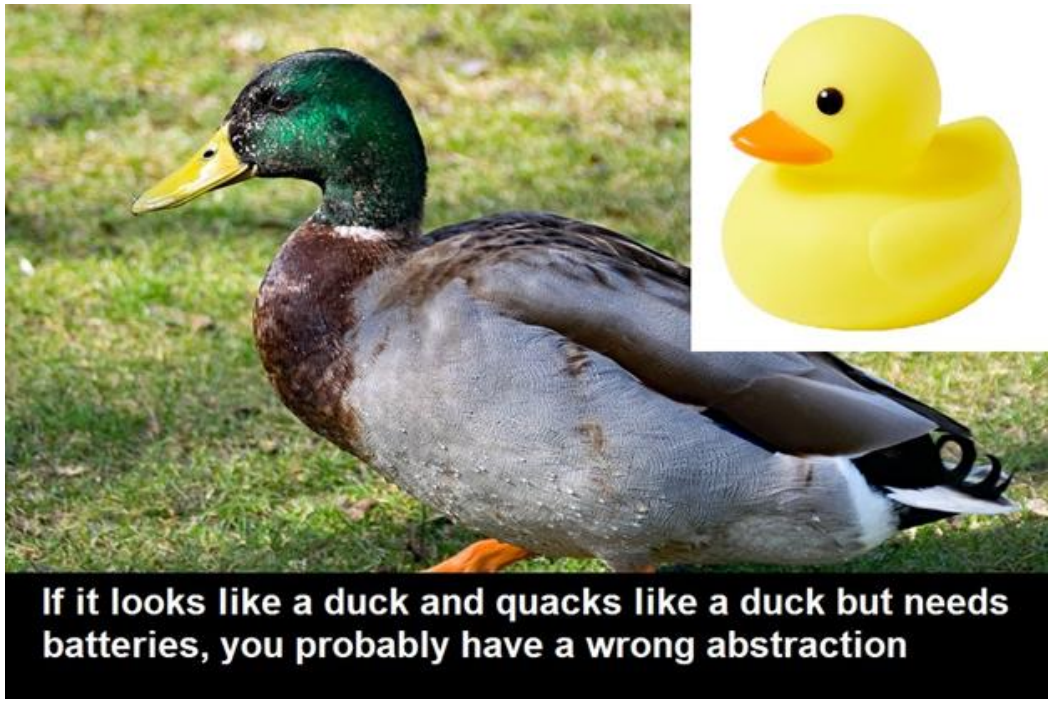
La subclase debe cumplir al menos los mismos resultados garantizados por la superclase.

```
class Super {
    int doble(int x) {
        return x * 2;
    }
}

class Sub extends Super {
    @Override
    int doble(int x) {
        return x * 2 + 1; // ✗ no cumple el contrato original
    }
}
```

Principios SOLID

Liskov Substitution Principle

Principio de sustitución de Liskov



Indicadores de Violaciones del LSP

Comprobaciones de tipo en tiempo de ejecución: instanceof o equivalentes.

Métodos que arrojan excepciones tipo "UnsupportedOperation" .

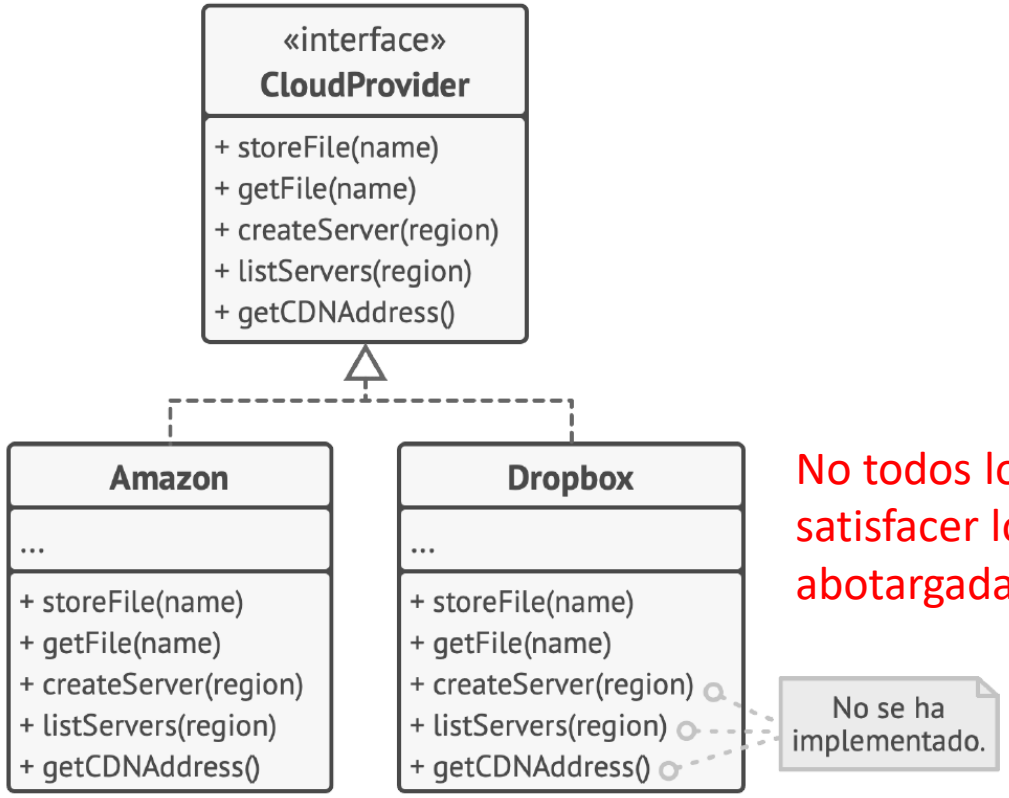
Sobreescrituras que vacían funcionalidad: Métodos que no hacen nada o lanzan excepciones.

Comentarios de advertencia: "No usar con clase X" o "Precaución al usar con subclase Y".

Principios SOLID

Interface Segregation Principle

Principio de segregación de interfaces



No todos los clientes pueden satisfacer los requisitos de la abotargada interfaz.

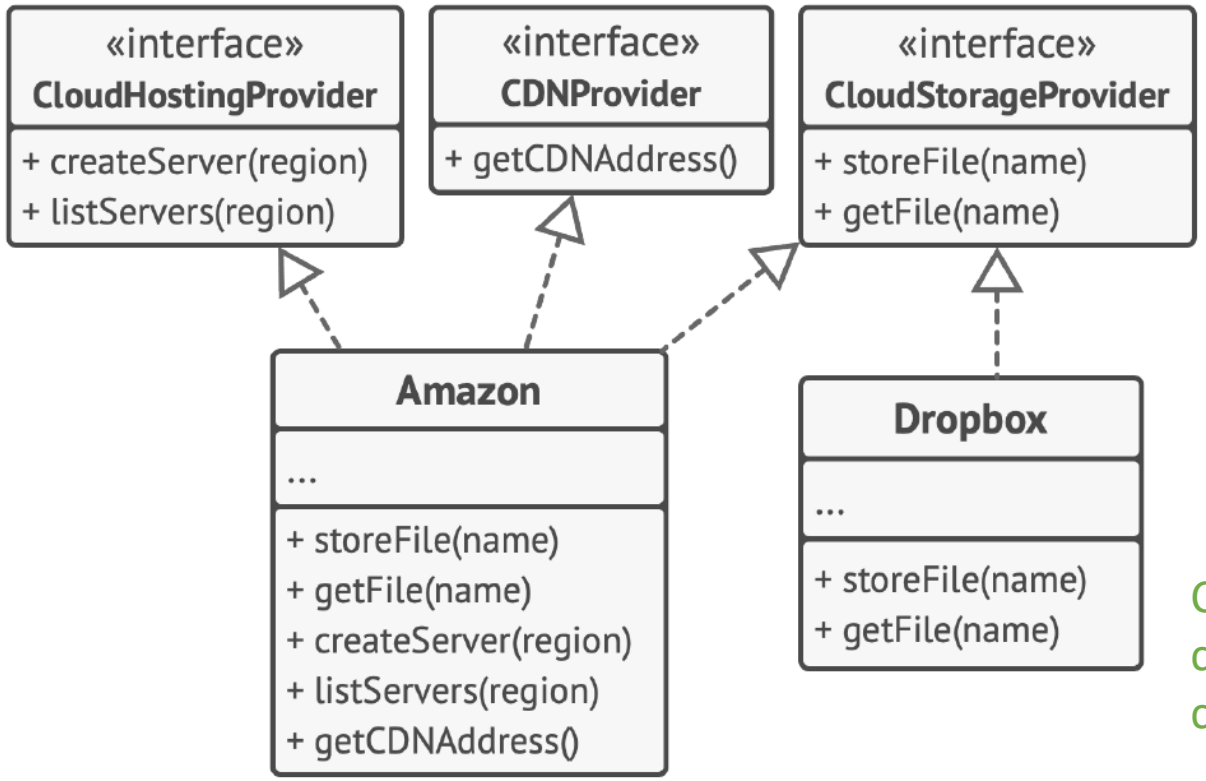
> "No se debe forzar a los clientes a depender de métodos que no utilizan."

- ✓ Evitar que las clases del cliente tengan que implementar comportamientos que no necesitan
- ✓ Divide las interfaces grandes, en varias interfaces pequeñas

Principios SOLID

Interface Segregation Principle

Principio de segregación de interfaces



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

Con interfaces mas pequeñas, los clientes pueden decidir cuales son capaces de implementar

Principios SOLID

I Interface Segregation Principle

Principio de segregación de interfaces

```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 */ }
}
```

Violación ISP

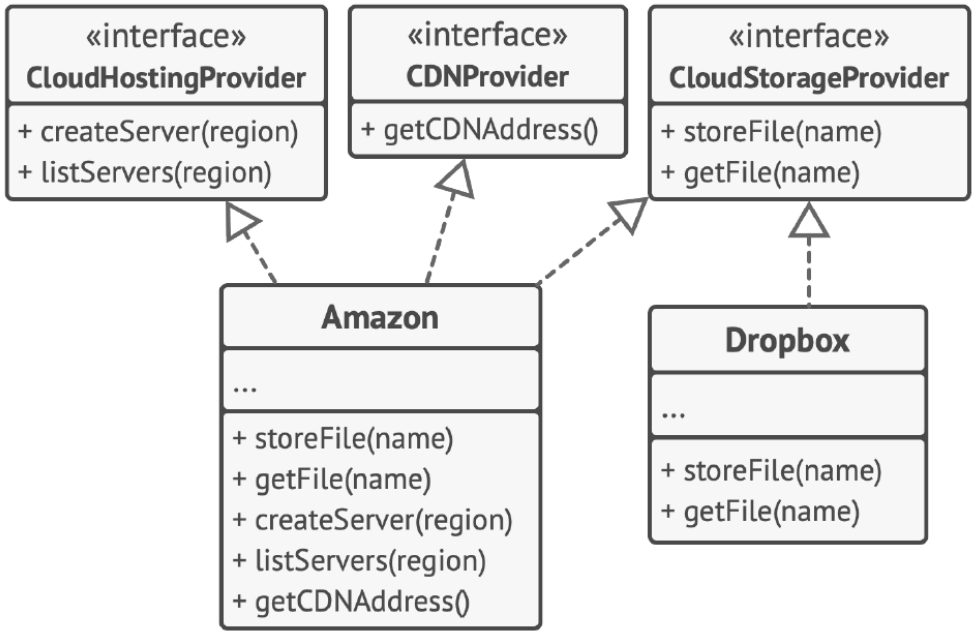
Aplicación correcta de ISP

```java
// 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
}
```

Principios SOLID

I Interface Segregation Principle

Principio de segregación de interfaces



Beneficios Estratégicos del ISP

Desacoplamiento entre clientes : Los cambios en una interfaz afectan solo a los clientes relevantes.

Mejor expresividad del código : Las interfaces comunican claramente su propósito.

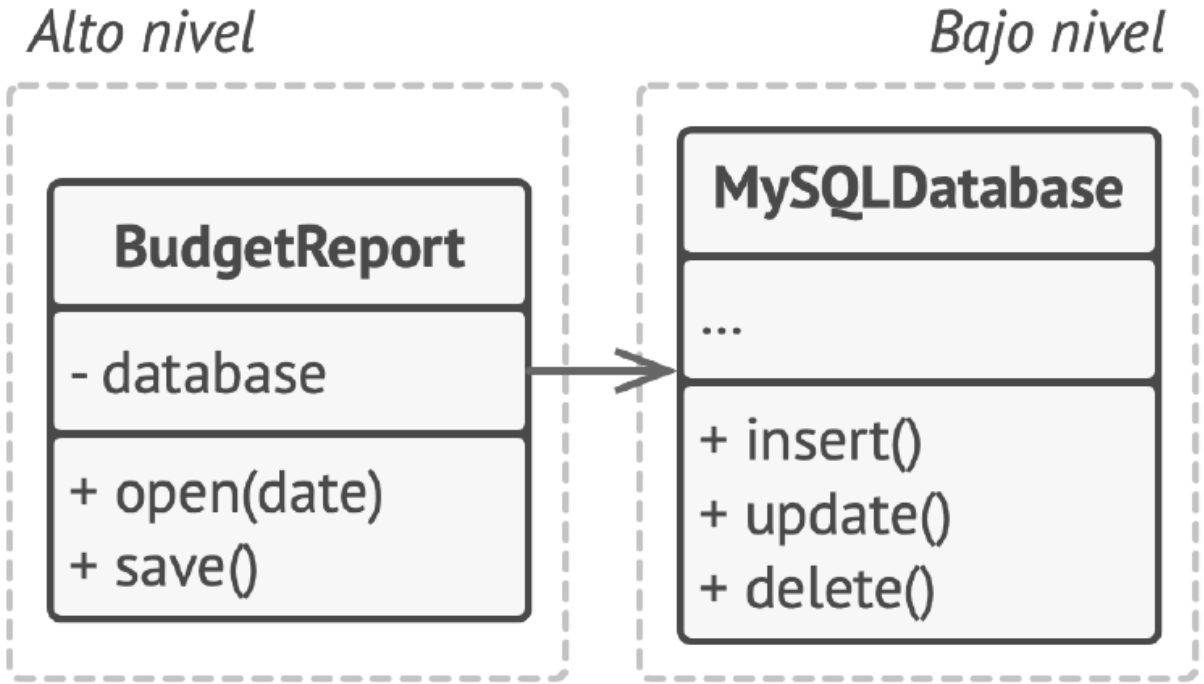
Mayor cohesión : Interfaces enfocadas en aspectos específicos del sistema.

Facilitar la implementación : Reducir la probabilidad de métodos vacíos o innecesarios.

Principios SOLID

Dependency Inversion Principle

Principio de inversión de dependencias



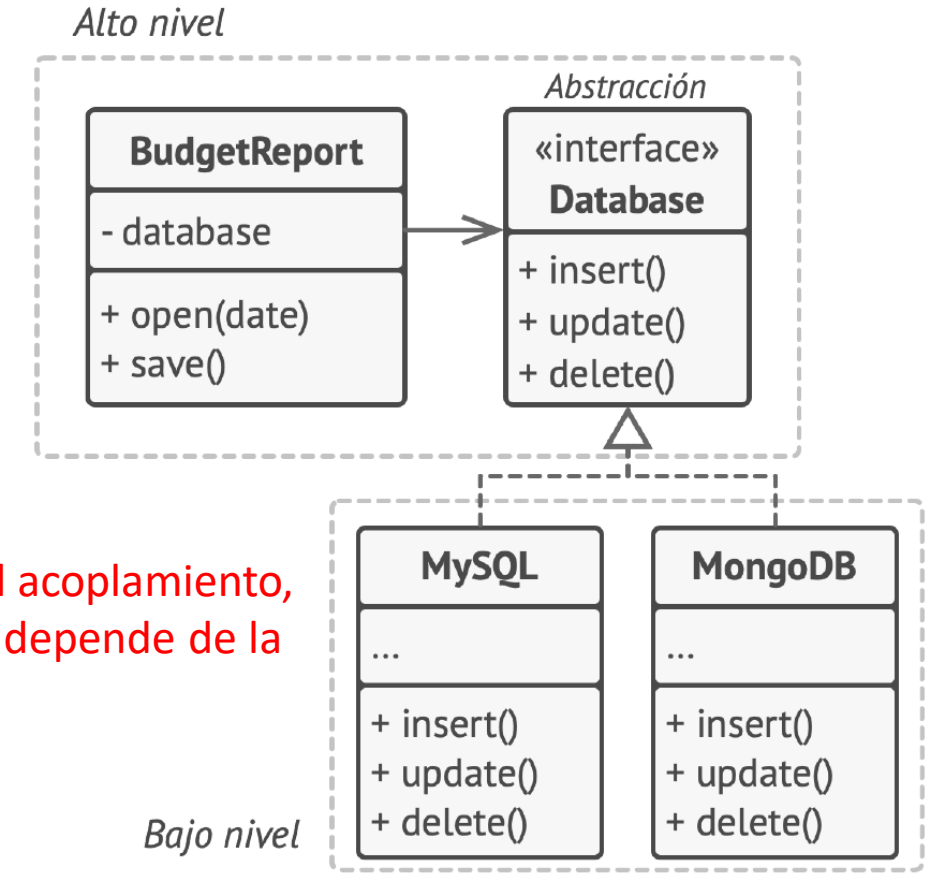
> "Las clases de alto nivel no deben depender de clases de bajo nivel. Ambas deben depender de abstracciones. Las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones."

- ✓ Al diseñar software generalmente encontramos clases de **bajo nivel** y clases de **alto nivel**
- ✓ Disminuir el acoplamiento, evitar que las clases de la lógica (Alto nivel), dependan de las de bajo nivel.

Principios SOLID

Dependency Inversion Principle

Principio de inversión de dependencias



Se minimiza el acoplamiento, la clase ahora depende de la interfaz

> "Las clases de alto nivel no deben depender de clases de bajo nivel. Ambas deben depender de abstracciones. Las abstracciones no deben depender de detalles. Los detalles deben depender de abstracciones."

Concepto	Significado
Módulo de alto nivel	Lógica de negocio, reglas importantes del sistema.
Módulo de bajo nivel	Implementaciones concretas, acceso a datos, APIs, utilidades.
Abstracción	Interfaces o clases abstractas que definen qué se debe hacer, pero no cómo.
Detalles	Clases concretas que implementan la abstracción.

Principios SOLID

Dependency Inversion Principle

Principio de inversión de dependencias

```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 = " + typ
        // Procesamiento y generación del reporte
        return new Report(data);
    }
}
```

Violación DIP

Aplicación Correcta DIP

```java
// 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 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);
    }
}
```

Principios GRASP - General Responsibility Assignment Software Patterns

Nueve patrones de diseño orientado a objetos que ayudan a asignar responsabilidades a clases y objetos de manera coherente, flexible y mantenible.

Principio	Descripción breve
1. Experto en informacion(Information Expert)	Asignar una responsabilidad a la clase que tiene la mayor cantidad de la información necesaria para cumplirla.
2. Creador (Creator)	Asignar la responsabilidad de crear un objeto a una clase que lo contiene, usa o tiene los datos necesarios.
3. Controlador (Controller)	Asignar la responsabilidad de manejar eventos del sistema a una clase que representa una interfaz entre el sistema y el mundo exterior.
4. Bajo Acoplamiento (Low Coupling)	Diseñar clases con baja dependencia entre sí para mejorar la flexibilidad y la reutilización.
5. Alta Cohesión (High Cohesion)	Mantener la unidad funcional y enfoque de una clase, agrupando comportamientos relacionados.
6. Polimorfismo (Polymorphism)	Usar interfaces o clases abstractas para permitir que el comportamiento varíe según el tipo del objeto.
7. Indirección (Indirection)	Usar un objeto intermedio para desacoplar dos elementos que necesitan interactuar.
8. Variación Protegida (Protected Variations)	Diseñar para proteger los elementos contra los efectos de cambios, utilizando interfaces o encapsulamiento.
9. Fabricación Pura (Pure Fabrication)	Crear una clase artificial que no pertenece al dominio pero ayuda a lograr bajo acoplamiento o alta cohesión.

Principios de diseño – SOLID vs GRASP

Relación	Descripción
Complementarios	GRASP se enfoca en "quién hace qué", mientras que SOLID se centra en "cómo lo hace".
Responsabilidades	Ambos promueven una distribución clara y adecuada de responsabilidades.
Bajo acoplamiento y alta cohesión	El principio GRASP de Bajo Acoplamiento y Alta Cohesión está directamente alineado con los principios SOLID (especialmente SRP y DIP).
Diseño orientado a abstracciones	Tanto GRASP (Polimorfismo, Variación Protegida) como SOLID (DIP, OCP) promueven el uso de interfaces y clases abstractas.
Mantenimiento y escalabilidad	Ambos ayudan a crear sistemas más flexibles, fáciles de mantener y con menor deuda técnica.

Patrón de diseño – Recursos

Libros:

- ***"Patrones de diseño. Elementos de software orientado a objetos reutilizables"***
  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

- ***"Sumérgete en los patrones de diseño"***
  Alexander Shvets

- ***"Introducción a los patrones de diseño. Un enfoque practico"***
  Oscar Blancharte

- ***"Patrones de diseño en java. Los 23 modelos de diseño: descripción y soluciones ilustradas "***
  Lauren Debrauwer

UNIVERSIDAD
Popular del cesar