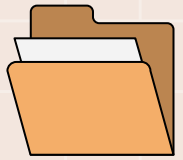
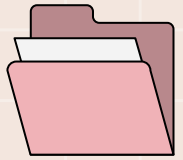
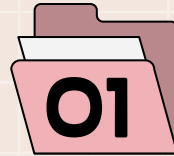


PATRONES DE DISEÑO GRASP Y SOLID

Prof. Ing. Pablo Hernandez



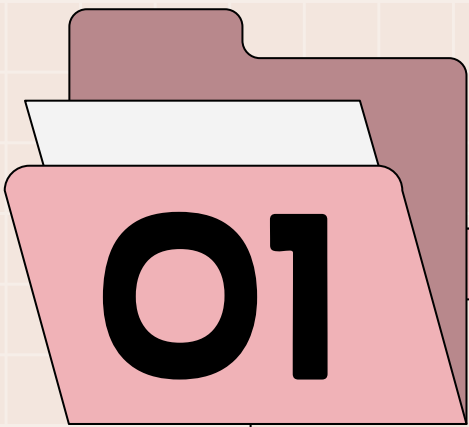
TEMAS



GRASP



SOLID



GRASP



QUE ES GRASP

- En diseño orientado a objetos, GRASP son patrones generales de software para asignación de responsabilidades.
- Es el acrónimo de "GRASP (object-oriented design General Responsibility Assignment Software Patterns)".
- Aunque se considera que más que patrones propiamente dichos, son una serie de "buenas prácticas" de aplicación recomendable en el diseño de software.

Tipos

- Experto en información
- Creador
- Controlador
- Alta cohesión y bajo acoplamiento

PATRON EXPERTO

- El GRASP de experto en información es el principio básico de asignación de responsabilidades.
- La responsabilidad de la creación de un objeto o la implementación de un método debe recaer sobre la clase que conoce toda la información necesaria para crearlo.
- Obtendremos un diseño con mayor cohesión y así la información se mantiene encapsulada (disminución del acoplamiento).

Problema

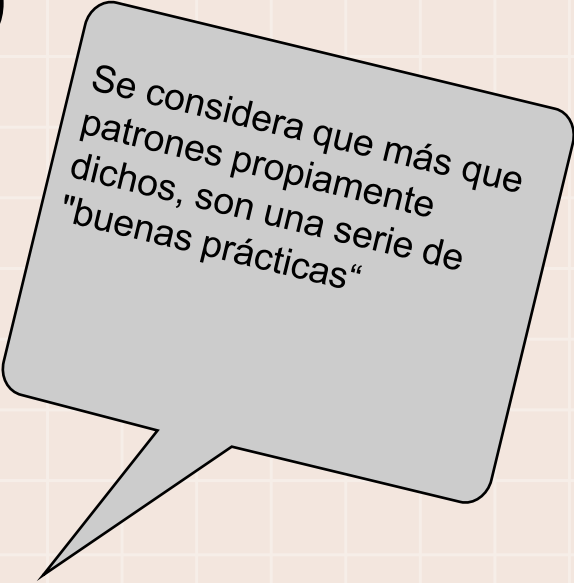
¿Cuál es el principio general para asignar responsabilidades a los objetos?

Solución

Asignar una responsabilidad al experto en información.

Beneficios

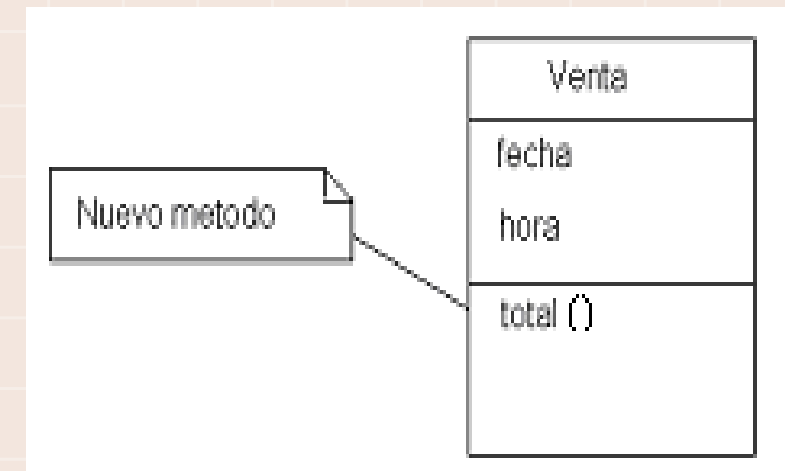
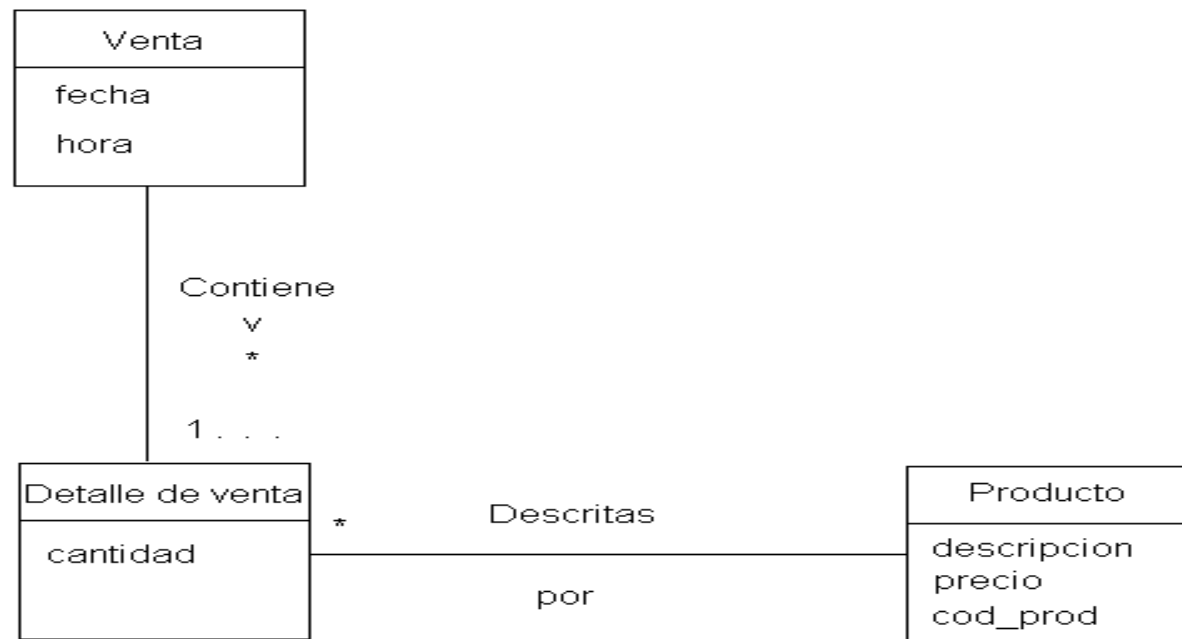
Se mantiene el encapsulamiento, los objetos utilizan su propia información para llevar a cabo sus tareas. Se distribuye el comportamiento entre las clases que contienen la información requerida. Son más fáciles de entender y mantener.



Se considera que más que patrones propiamente dichos, son una serie de "buenas prácticas"

EXPERTO EJEMPLO

Alguna clase necesita conocer el total de la venta.
¿Quién es el responsable de conocer el monto total de la venta?





BENEFICIOS

- Mantiene el encapsulamiento de la información
- Favorece el bajo acoplamiento .
- Son más fáciles de entender .

CREADOR

El patrón creador nos ayuda a identificar quién debe ser el responsable de la creación (o instanciación) de nuevos objetos o clases. La nueva instancia deberá ser creada por la clase que:

- Tiene la información necesaria para realizar la creación del objeto, o
- Usa directamente las instancias creadas del objeto, o
- Almacena o maneja varias instancias de la clase
- Contiene o agrega la clase

CREADOR

PROPÓSITO:

- El propósito fundamental de patrón creador es encontrar un creador que debemos conectar con el objeto producido en cualquier evento.

OBJETIVOS:

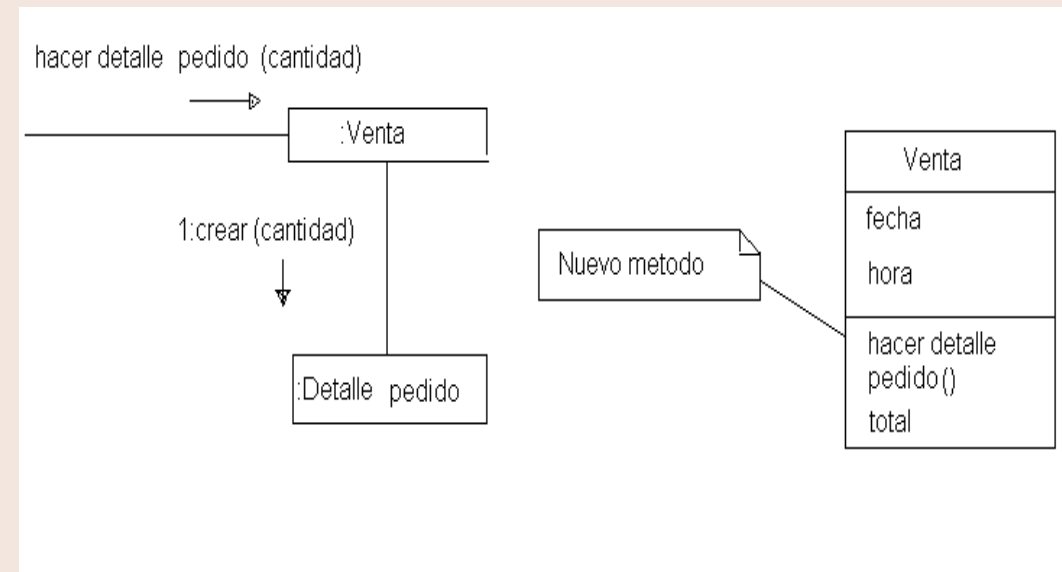
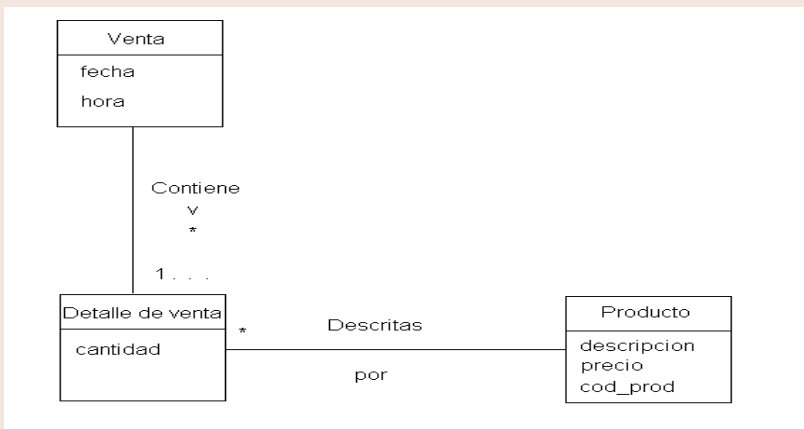
- La intención básica del patrón es encontrar un Creador que necesite conectarse al objeto creado en alguna situación
- Promueve el bajo acoplamiento, al hacer responsable a una clase de la creación de objetos que necesita referenciar

CREADOR

B es un Creador de A si se cumple uno o más de las siguientes condiciones:

- B agrega objetos de A
 - B contiene objetos de A
 - B registra objetos de A
 - B utiliza intensivamente objetos de A
 - B tiene los datos de inicialización de objetos de A
- (B es Experto en la creación de A)

EJEMPLO CREADOR



PATRON CONTROLADOR

El patrón controlador es un patrón que sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es la que recibe los datos del usuario y la que los envía a las distintas clases según el método llamado.

Este patrón sugiere que la lógica de negocios debe estar separada de la capa de presentación, esto para aumentar la reutilización de código y a la vez tener un mayor control.

Se recomienda dividir los eventos del sistema en el mayor número de controladores para poder aumentar la cohesión y disminuir el acoplamiento.

PATRON CONTROLADOR

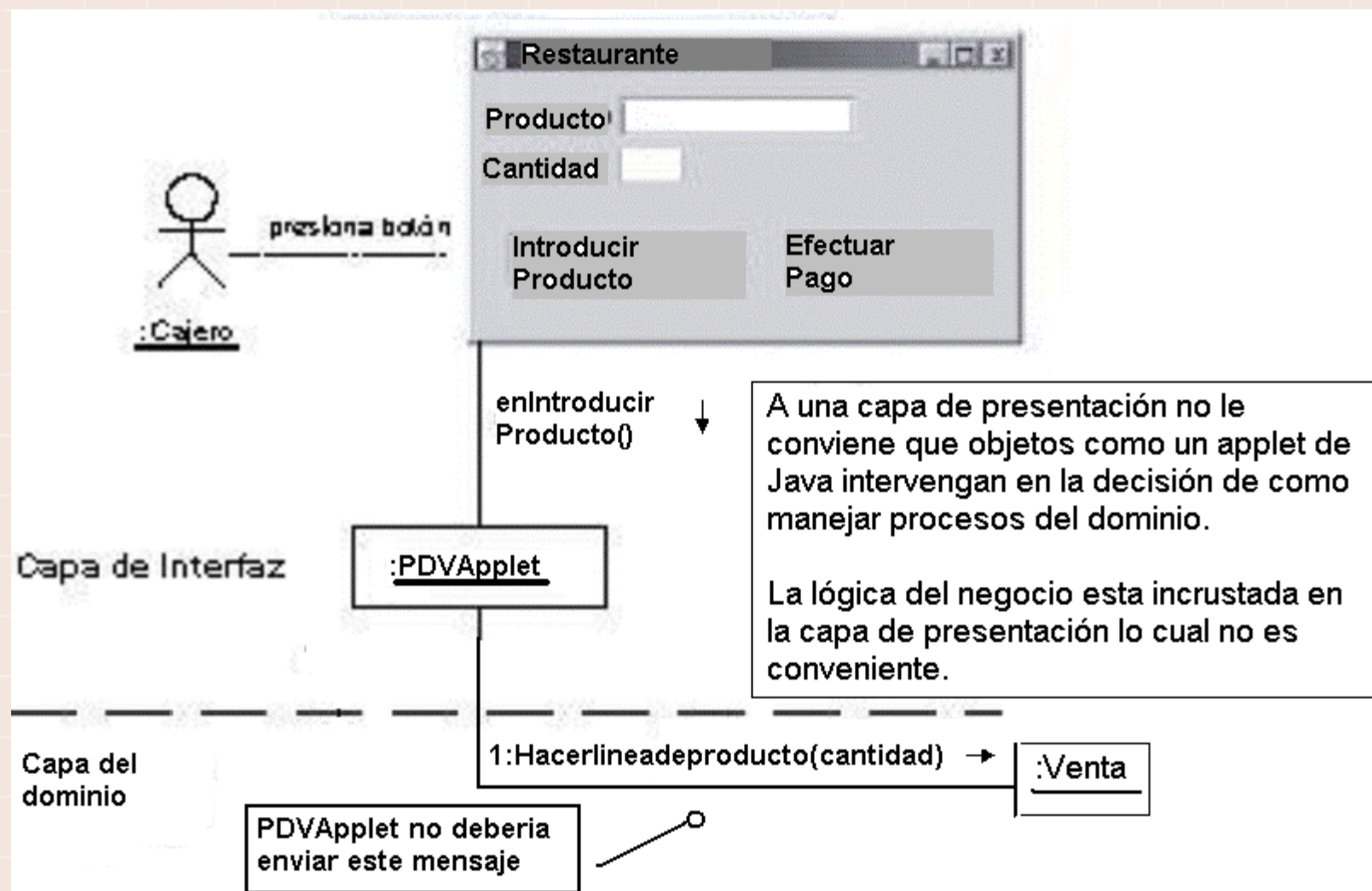
PROPÓSITO:

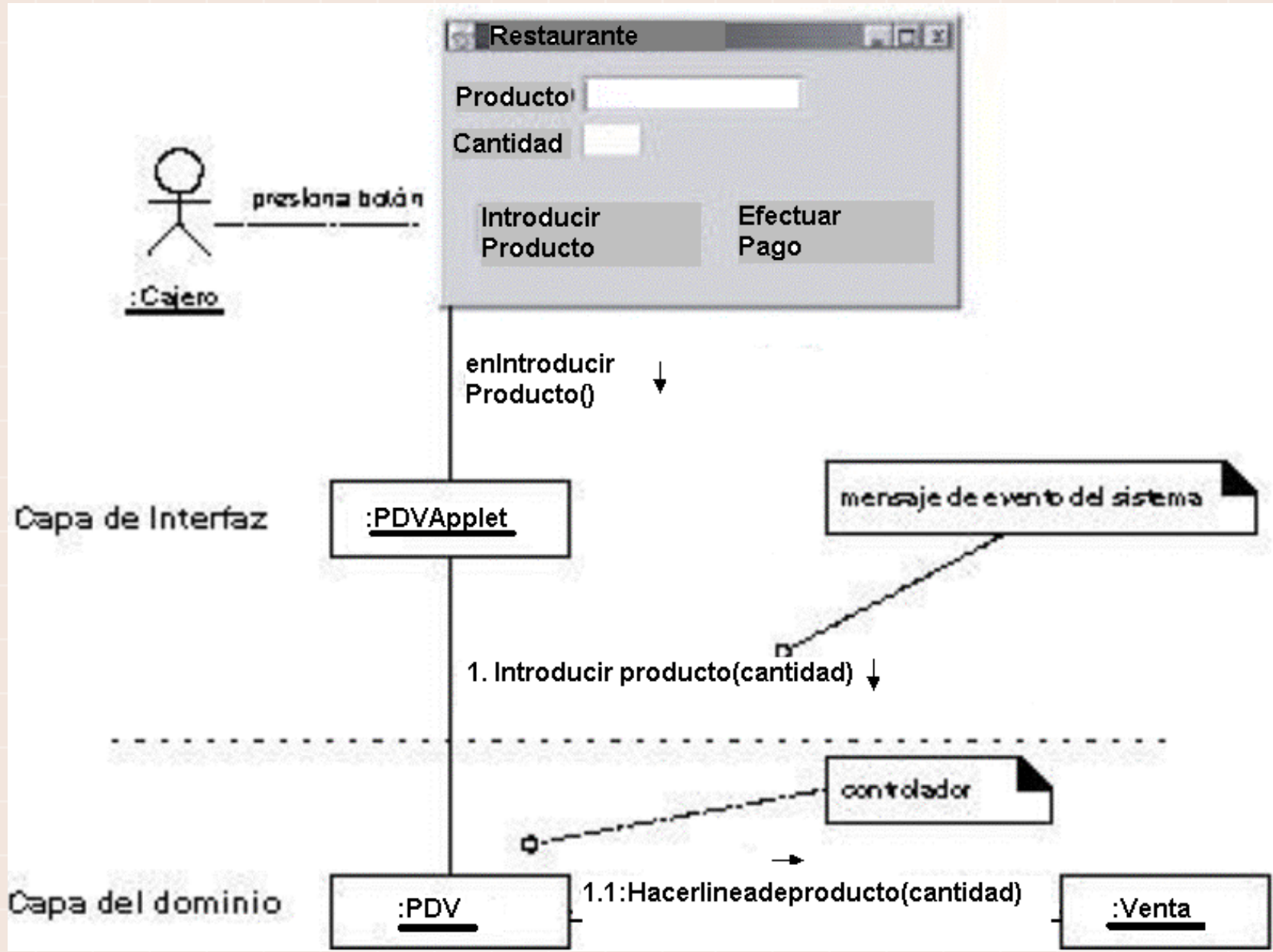
- Sirve como intermediario entre una determinada interfaz y el algoritmo que la implementa, de tal forma que es el controlador quien recibe los datos del usuario y quien los envía a las distintas clases según el método llamado.
- Se recomienda dividir los eventos del sistema en el mayor número de controladores para poder aumentar así la cohesión y disminuir el acoplamiento.

OBJETIVOS:

- Debería ser el primer objeto llamado después de un cambio en la interfaz de usuario.
- Controla/ejecuta un caso de uso. No hace demasiado por sí solo, controla, coordina.
- Pertenece a la capa de aplicación o a la de servicios.

EJEMPLO CONTROLADOR





ALTA COHESION BAJO ACOPLAMIENTO

- El **grado de acoplamiento** indica lo vinculadas que están unas clases con otras, es decir, lo que afecta un cambio en una clase a las demás y por tanto lo dependientes que son unas clases de otras.

- El **grado de cohesión** mide la coherencia de la clase, esto es, lo coherente que es la información que almacena una clase con las responsabilidades y relaciones que ésta tiene.

COHESION

1. Cohesión coincidente
2. Cohesión lógica
3. Cohesión temporal
4. Cohesión de procedimiento
5. Cohesión de comunicación
6. Cohesión de información
7. Cohesión funcional



ACOPLAMIENTO

1. Acoplamiento de contenido
2. Acoplamiento común
3. Acoplamiento de control




EJEMPLO

```
class AltoAcomplamiento
{
    public void SaludarEnIngles(string type)
    {
        switch (type)
        {
            case "GM":
                Console.WriteLine("Good Morning");
                break;
            case "GE":
                Console.WriteLine("Good Evening");
                break;
            case "GN":
                Console.WriteLine("Good Night");
                break;
        }
    }
}

class AltoAcoplamiento2
{
    public AltoAcoplamiento2()
    {
        var ejemplo = new AltoAcomplamiento();
        ejemplo.SaludarEnIngles("GM");
    }
}
```

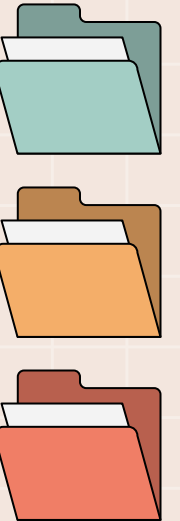
La clase
AltoAcoplamiento2
necesita conocer la lógica
interna de la clase
AltoAcomplamiento.

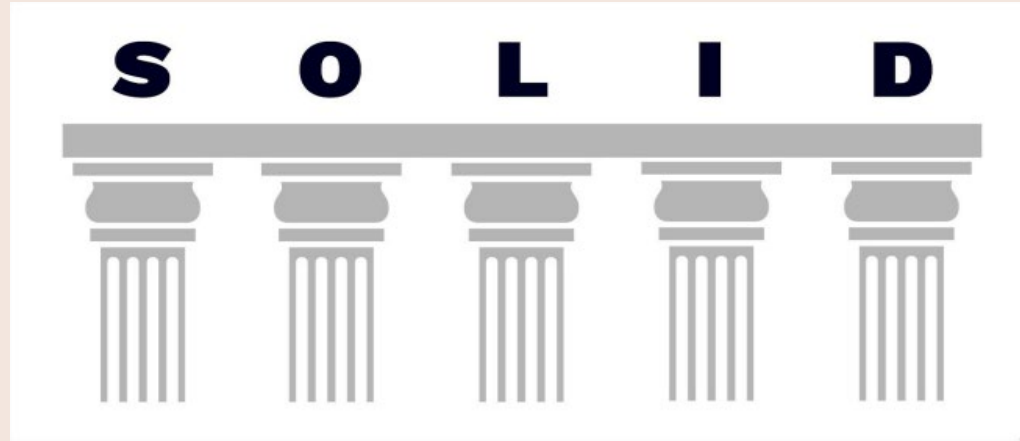


02

SOLID

Add a brief introduction of your section here: Let's dive in and get to know some interesting facts about animals!





SOLID es el acrónimo de cinco principios básicos de diseño que tienen como intención hacer que el diseño de software sea más simple y comprensible permitiendo reducir además los costos de mantenimiento.

SOLID apunta a lograr una alta cohesión y un bajo acoplamiento en nuestros programas.

S

Principio de
responsabilidad única

Se refiere a la responsabilidad única que debiera tener cada programa con una tarea bien específica y acotada

O

Principio abierto/cerrado

Toda clase, modulo, método, etc. debería estar abierto para extenderse pero debe estar cerrado para modificarse.

L

Principio de sustitución de
Liskov

Si la clase A es de un subtipo de la clase B, entonces deberíamos poder reemplazar B con A sin afectar el comportamiento de nuestro programa.

I

Principio de segregación de
interfaces

Ningún cliente debería estar obligado a depender de los métodos que no utiliza.

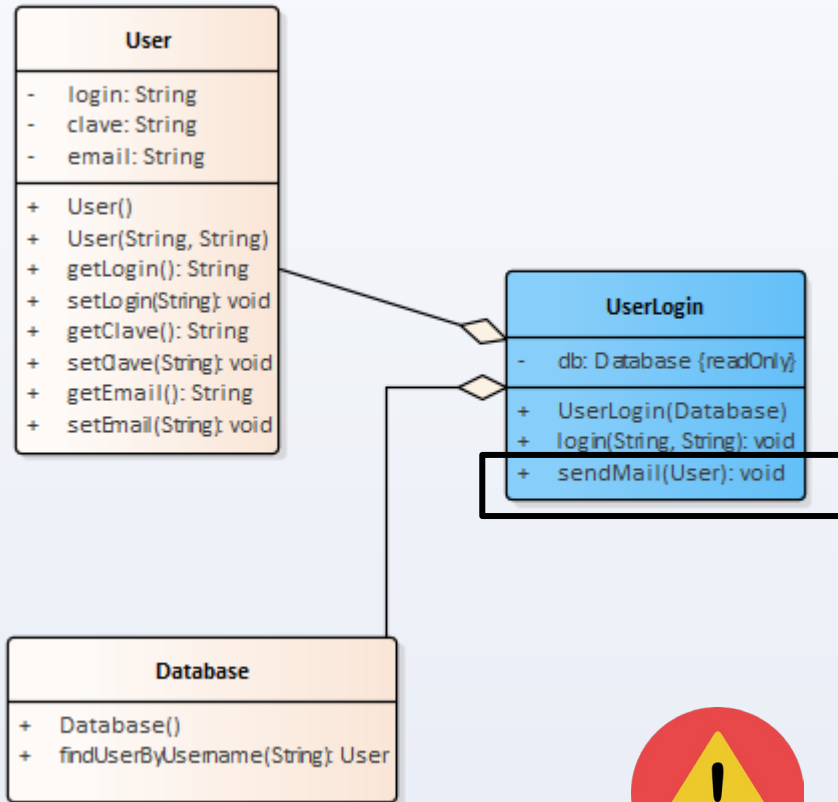
D

Principio de inversión de
dependencias

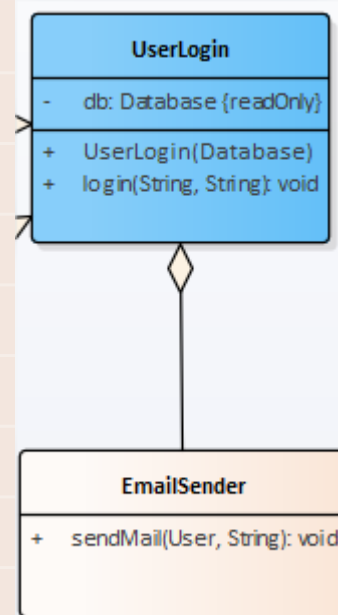
No deben existir dependencias entre los módulos, en especial entre módulos de bajo nivel y de alto nivel.

SOLID- Ejemplos

► S – Principio de Responsabilidad Única:



Esta clase `UserLogin` tiene como responsabilidad realizar el proceso de login pero además le dimos la responsabilidad de enviar mensajes al usuario. Este código viola el principio de responsabilidad única. Está haciendo dos cosas con objetivos diferentes.



Llevemos entonces el método `sendMail` a otra clase que tenga como responsabilidad el envío de mensajes.

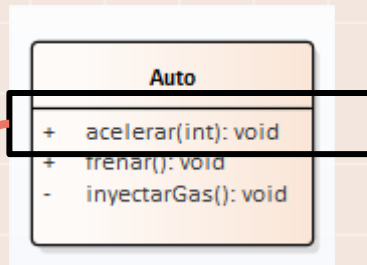
Este principio se refiere a la responsabilidad Única que debiera tener cada programa con una tarea bien específica y acotada.

SOLID- Ejemplos



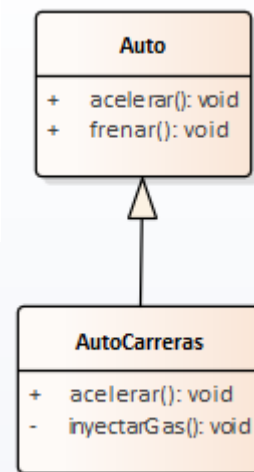
El principio de open/closed dice que toda clase, modulo, método, etc. debería estar abierto para extenderse pero debe estar cerrado para modificarse.

► O – Principio Abierto Cerrado:



```
public void acelerar(int tipo) {  
    System.out.println("Acelerando...");  
    //Si es un tipo auto de carreras  
    if (tipo == 1){  
        System.out.println("Activamos la Inyección a gas...");  
        inyectarGas();  
    }  
}
```

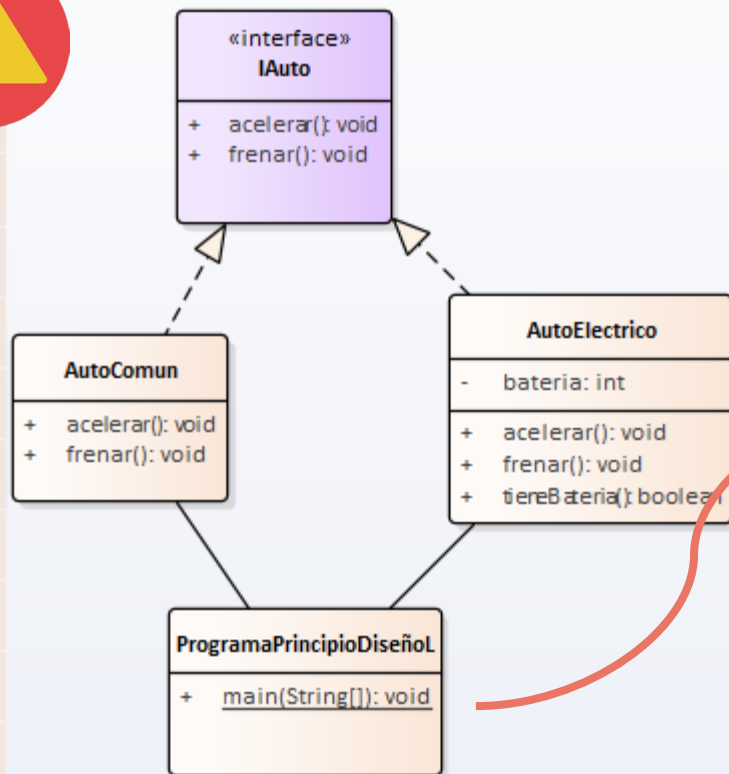
Esta clase *Auto* de acuerdo al tipo que se recibe en el método *acelerar* invoca a otra función. En vez de hacer esto, para cumplir con el principio de open/closed podemos extender nuestra clase creando otro tipo de clase derivada de la principal.



Creamos nuestra clase *AutoCarreras* que extiende *Auto* agregando aquí la funcionalidad. Extendiendo nuestra clase *Auto* estamos seguros que nuestra aplicación que usa *Auto* no se verá afectada de algún modo.

SOLID- Ejemplos

► L – Principio de Sustitución de Liskov:



```
public class ProgramaPrincipioDiseñoL {

    public static void main(String[] args) throws IOException {

        System.out.println("Ingrese el tipo de auto a crear....");
        System.out.println("1. Comun");
        System.out.println("2. Electrico");
        Scanner lector = new Scanner(System.in);
        System.out.println("Opción:");
        int tipo = lector.nextInt();

        if (tipo == 1) {
            AutoComun ac = new AutoComun();
            ac.acelerar();
        } else if (tipo == 2) {
            AutoElectrico ae = new AutoElectrico();
            if (ae.tieneBateria()) {
                ae.acelerar();
            } else {
                System.out.println("No hay bateria suficiente");
            }
        } else {
            throw new RuntimeException("Opción incorrecta");
        }
    }
}
```

El principio de sustitución de Liskov dice que si la clase A es de un subtipo de la clase B, entonces deberíamos poder reemplazar B con A sin afectar el comportamiento de nuestro programa.

SOLID- Ejemplos

► L – Principio de Sustitución de Liskov:

Observa cómo en el caso del auto eléctrico necesitamos invocar el método *tieneBateria()* para luego poder acelerar.

Con este diseño de clases estamos rompiendo el principio de sustitución porque necesitamos explícitamente conocer el tipo de vehículo y no podemos reemplazar la clase *AutoElectrico* con la interfaz *IAuto*.

Para poder cumplir con el principio, debemos realizar algunos cambios:

1. Modificar la clase *AutoElectrico* para que realice la validación en su propio método de aceleración:

```
@Override
public void acelerar() {
    // TODO Auto-generated method stub
    if (tieneBateria()) {
        System.out.println("Acelerando...");
    }else {
        System.out.println("No se puede acelerar");
    }
}
```



SOLID- Ejemplos

► L – Principio de Sustitución de Liskov:

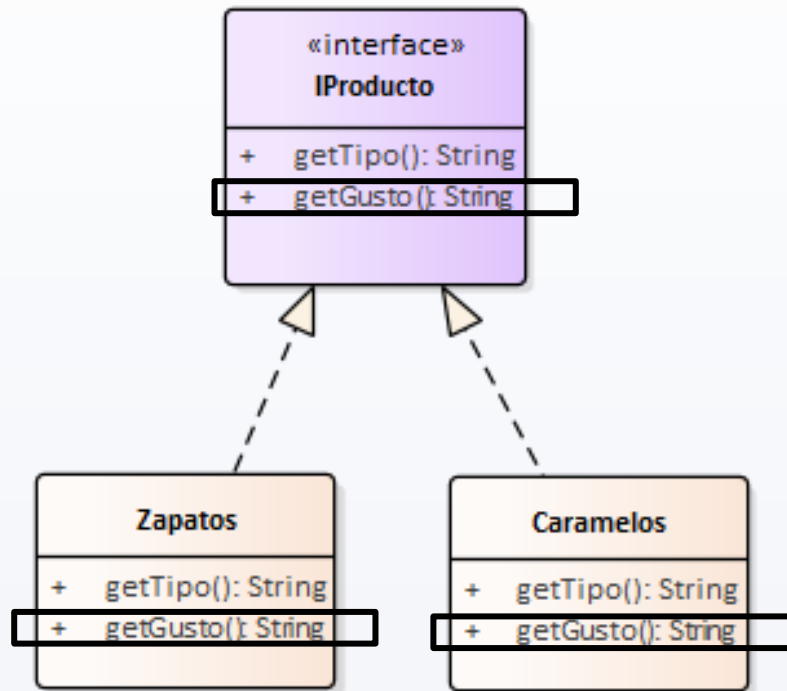
2. Modificar la clase del programa invocador:



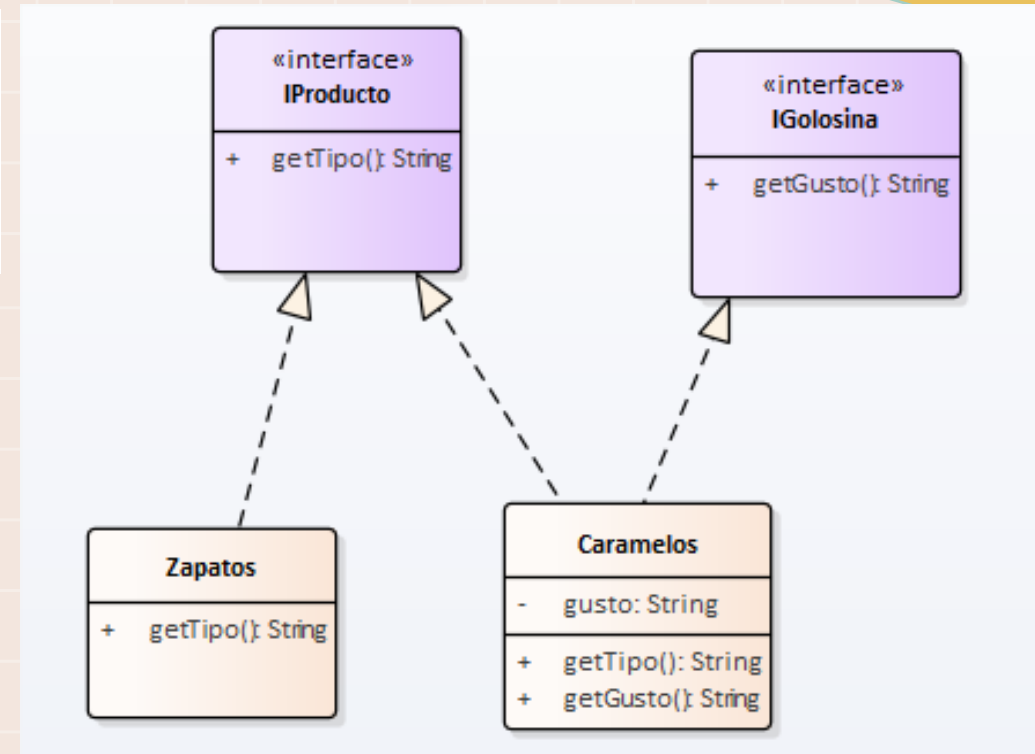
```
public class ProgramaPrincipioDiseñoL {  
  
    public static void main(String[] args) throws IOException {  
  
        IAuto auto;  
  
        System.out.println("Ingrese el tipo de auto a crear....");  
        System.out.println("1. Comun");  
        System.out.println("2. Electrico");  
        Scanner lector = new Scanner(System.in);  
        System.out.println("Opción:");  
        int tipo = lector.nextInt();  
  
        if (tipo == 1) {  
            auto = new AutoComun();  
        } else if (tipo == 2) {  
            auto = new AutoElectrico();  
        } else {  
            throw new RuntimeException("Opción incorrecta");  
        }  
  
        auto.acelerar();  
    }  
}
```

SOLID- Ejemplos

► I – Principio de Segregación de Interfaces



Las clases Zapatos y Caramelos son clientes de la interface Iproducto. Ambas deben implementar las funcionalidades definidas en la interface.



Creamos otra interface que contenga la funcionalidad que necesita el cliente Caramelos y de esta forma la clase Zapatos solo utiliza lo que necesita.

El principio de segregación de interfaz dice que ningún cliente debería estar obligado a depender de los métodos que no utiliza.

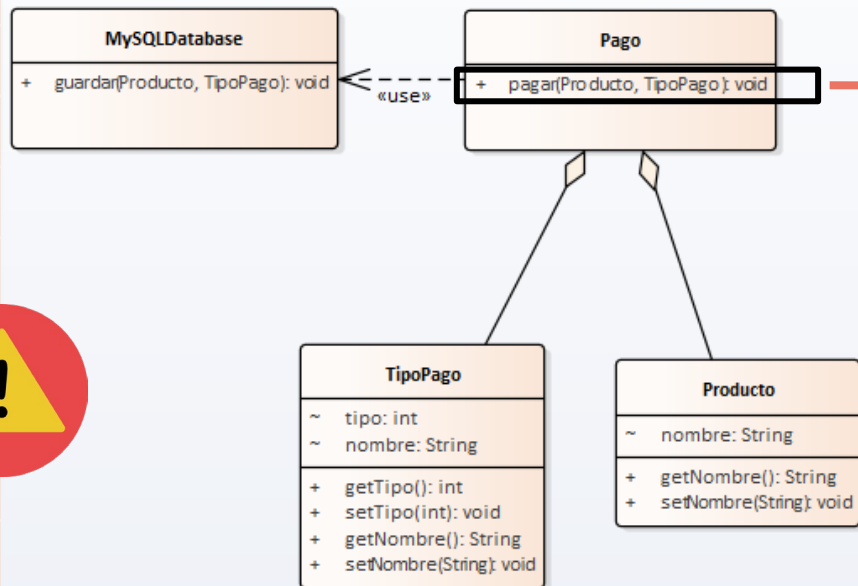
SOLID- Ejemplos

El principio nos indica que no deben existir dependencias entre los módulos, en especial entre módulos de bajo nivel y de alto nivel

► D – Principio de Inversión de Dependencias

Nuestro software no debería depender, por ejemplo, de cómo están implementados los frameworks para acceso a base de datos o conexiones con el servidor.

Para este fin debemos depender de interfaces sin conocer qué sucede exactamente en la implementación de dichas interfaces.



```
public class Pago {

    public void pagar(Producto producto, TipoPago tipoPago) {

        MySQLDatabase database = new MySQLDatabase();
        database.guardar(producto, tipoPago);

    }

}
```

Observa que estamos haciendo un uso directo de Database dependiendo directamente de su implementación. Estamos utilizando en nuestro código clases de bajo nivel, la clase `MySQLDatabase`.

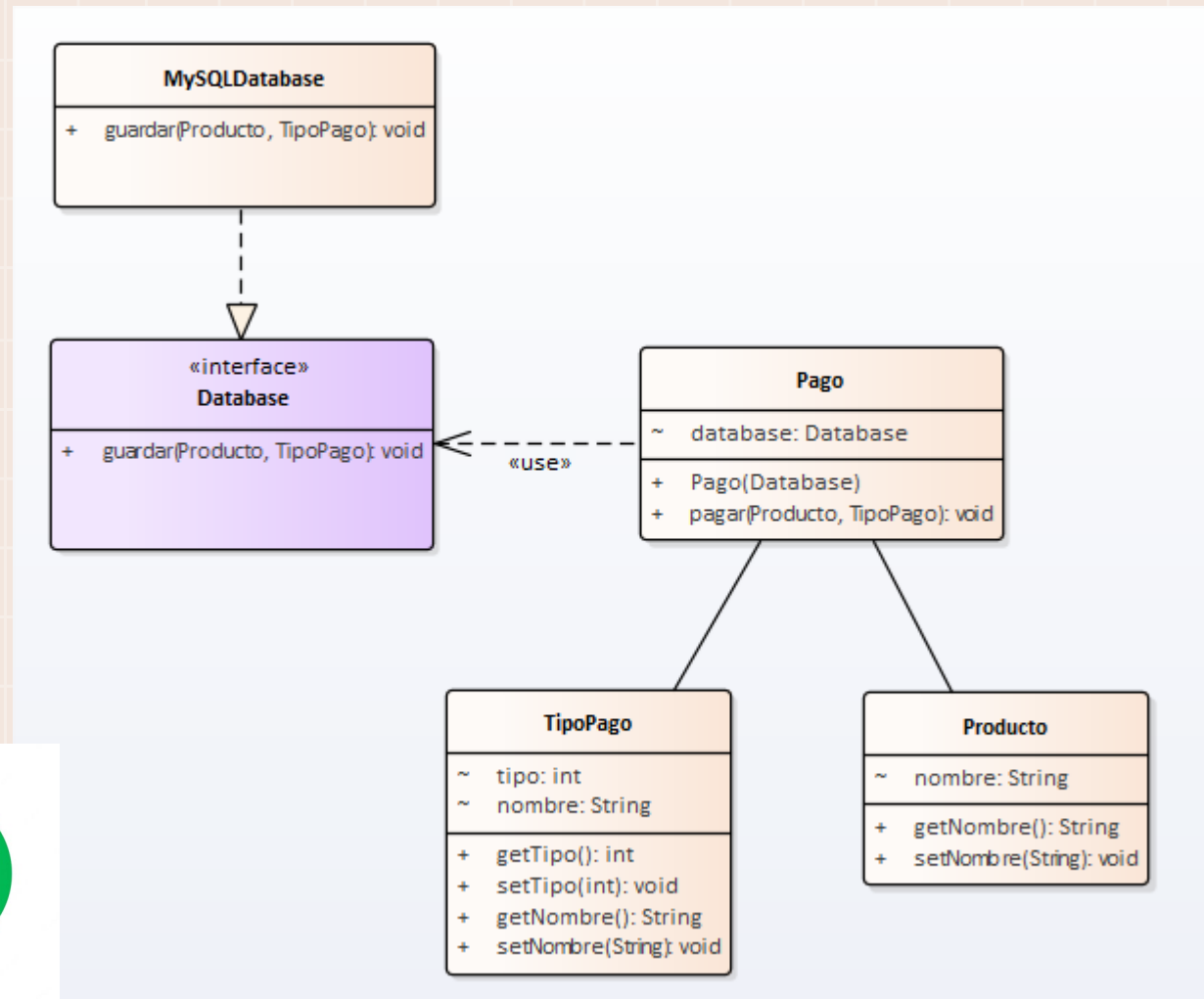
Si el día de mañana quisiéramos migrar a otra base de datos deberíamos modificar nuestro código directamente.



SOLID- Ejemplos

► D – Principio de Inversión de Dependencias

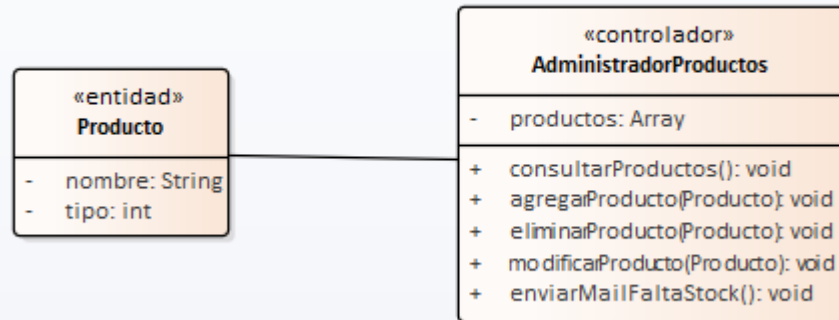
1. Lo primero que tenemos que hacer es dejar de depender directamente de la clase concreta MySQLDatabase por lo que vamos a crear una interfaz que nos desacople de la persistencia. No tenemos porqué saber en nuestro código como o donde se guarda el producto.
2. La clase Pago ahora recibirá la interfaz Database en el constructor y no tiene porque saber que motor de base de datos se encargará de guardar la información.
3. En la creación de la aplicación se creará el que corresponda.



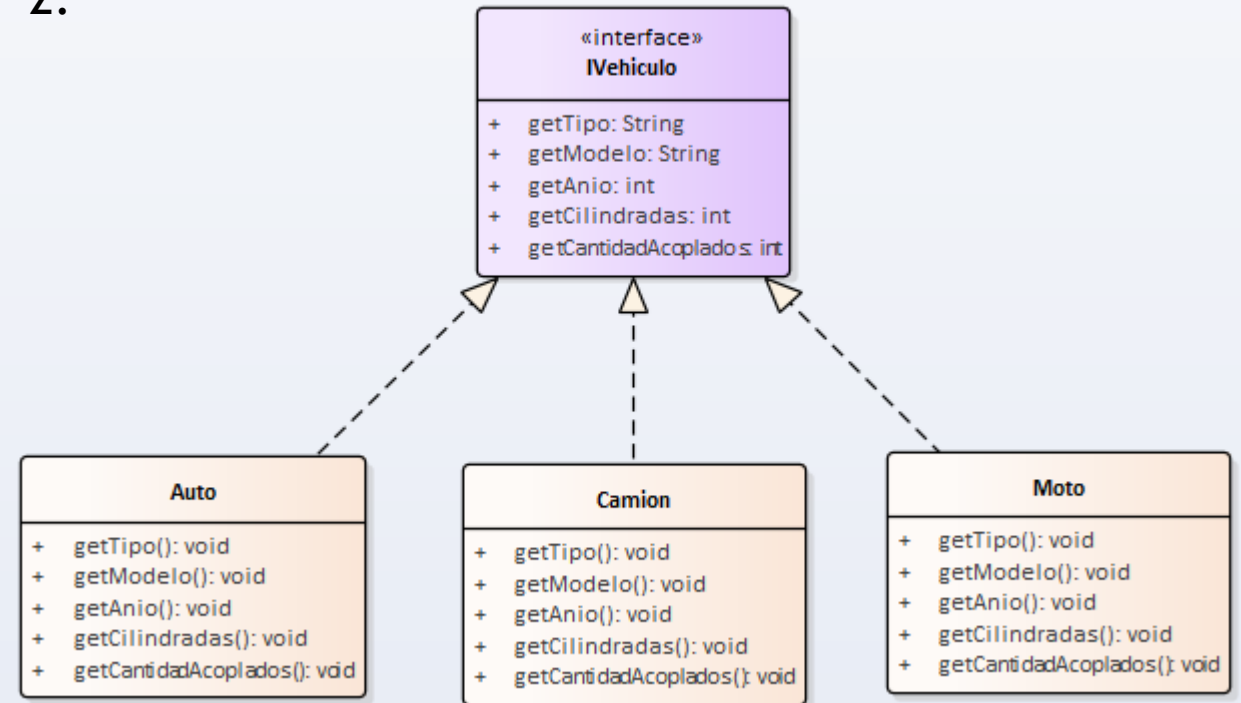
Actividades

- Indicar en los siguientes diagramas de clases que principio de diseño no cumple y como lo solucionaría.

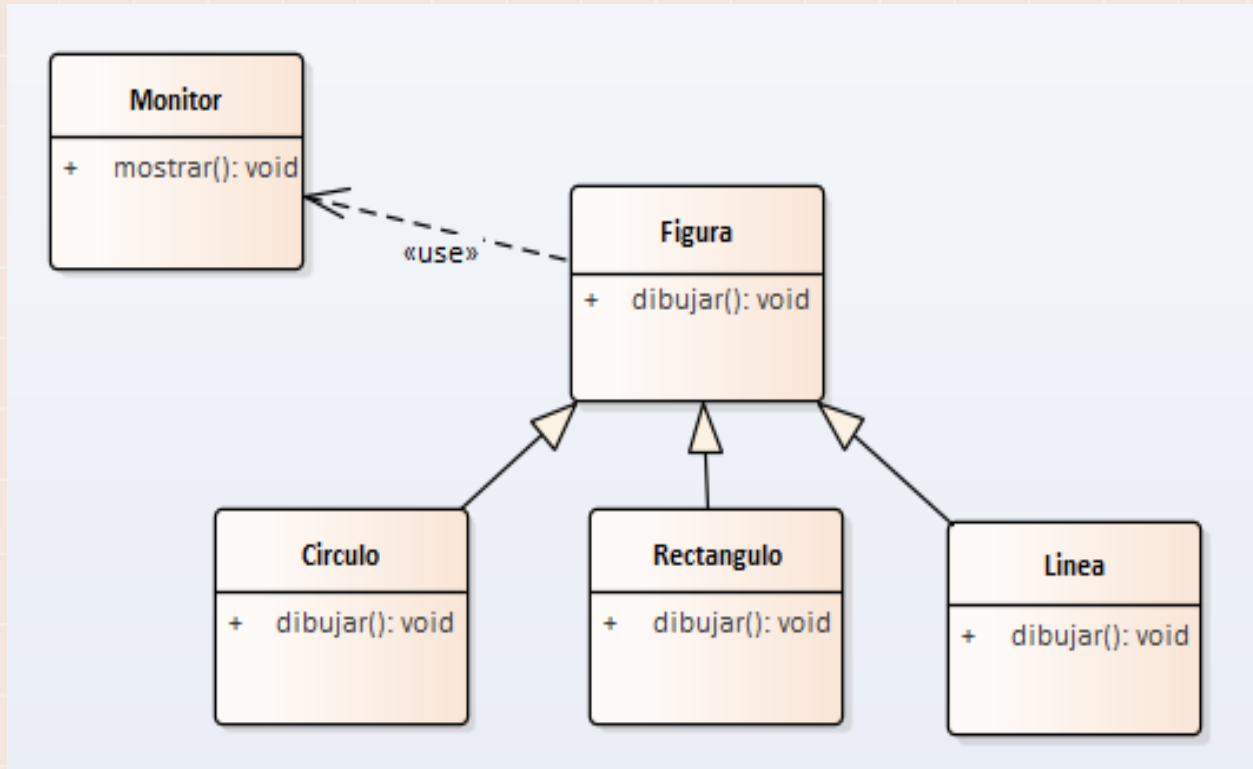
1.



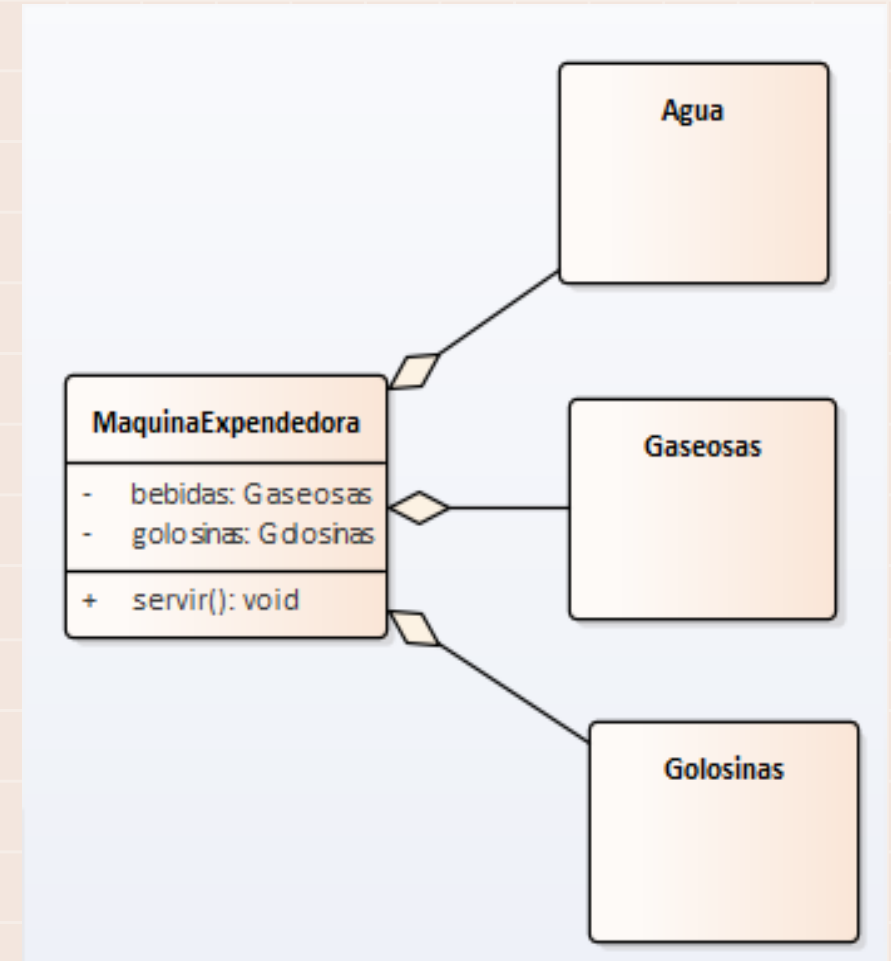
2.



3.



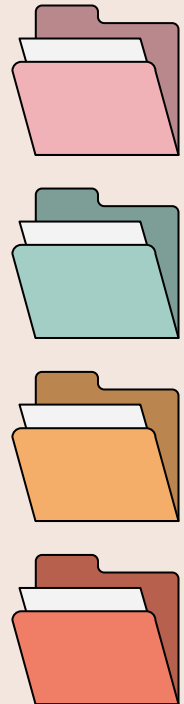
4.



SINTESIS



- Patrones de diseño
- Patrón GRASP
- Patrón SOLID





Bibliografía de la materia



Guardati, Silvia, E-book Estructuras de Datos Básicas Programación orientada a objetos con Java, México, Alfa omega, 20 15

FONTELA, Carlos, E-book Uml Modelado de Software para Profesionales, México: Alfa omega, 20 11

PRESSMAN, Roger. Ingeniería del software: un enfoque práctico. Edición: 7a ed. 20 10. México, D.F.: McGraw Hill.

Fontela, C. (20 10). Uml Modelado De Software Para Profesionales



CONSULTAS!

pahernadnez@uade.edu.ar
Canal de TEAMS

