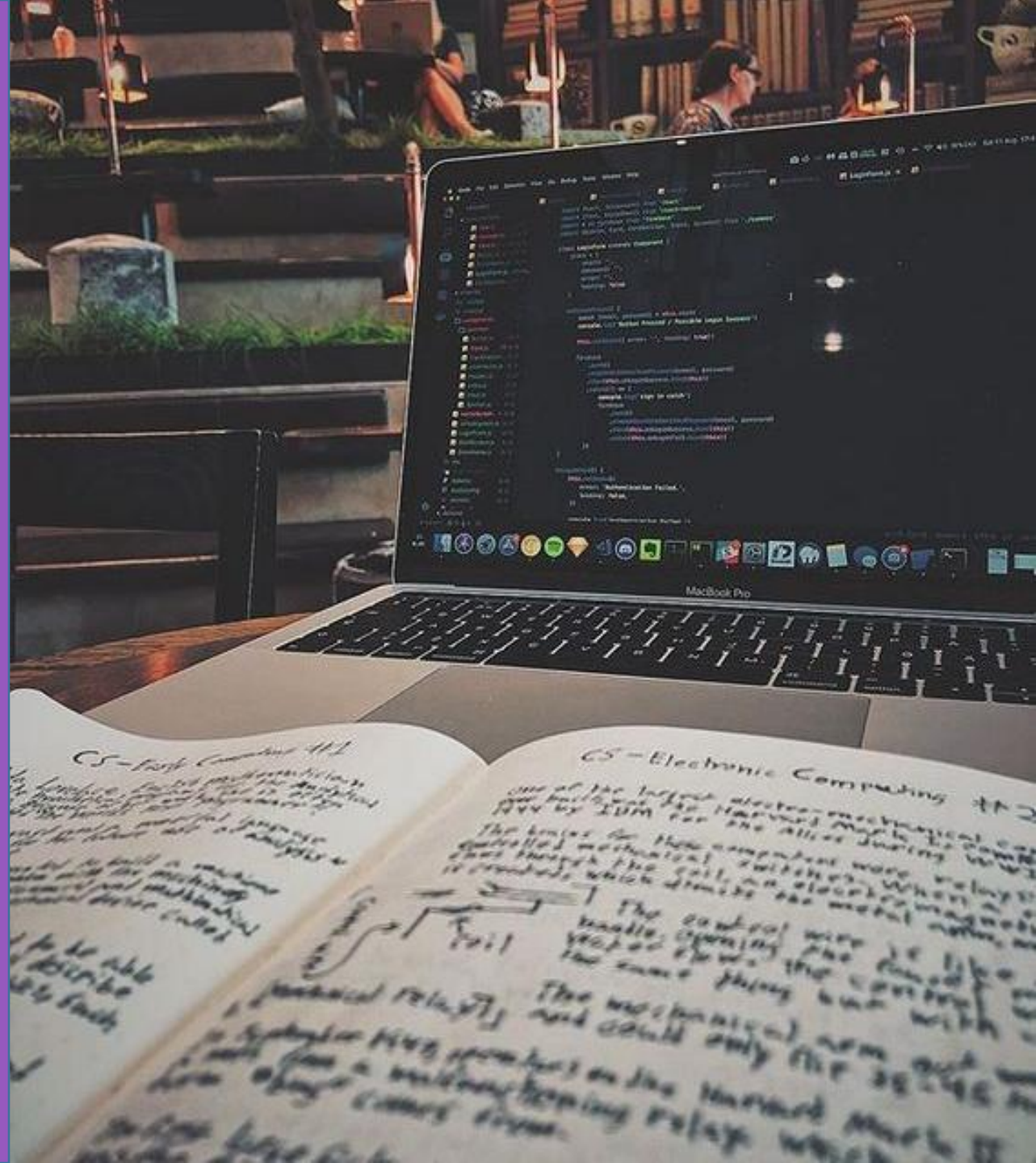
Principios S.O.L.I.D.

- Acomplamiento
- SOLID
- Definición
- Principio de Responsabilidad Única
- Liskov

POO

- Herencia
- Polimorfismo
- Sobreescritura de métodos
- Errores de abstracción



A tener en cuenta

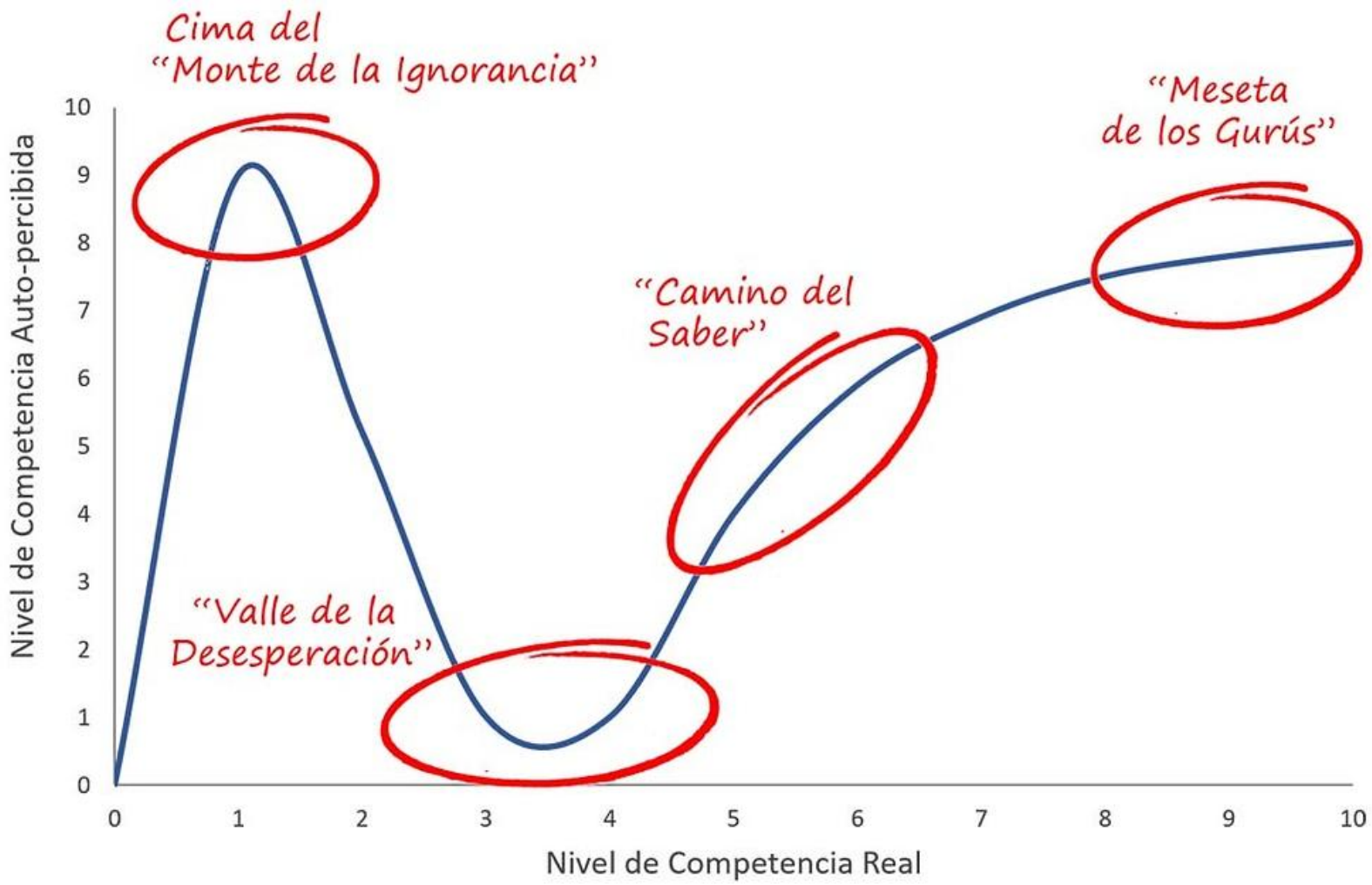
David Dunning y Justin Kruger

El efecto Dunning-Kruger es un **sesgo cognitivo**, según el cual los individuos con escasa habilidad o conocimientos sufren de un sentimiento de superioridad ilusorio considerándose más inteligentes que otras personas más preparadas, midiendo incorrectamente su habilidad por encima de lo real.

David Dunning y Justin Kruger de la Universidad de Cornell concluyeron que:

> La sobrevaloración del incompetente nace de la mala interpretación de la capacidad de uno mismo. La infravaloración del competente nace de la mala interpretación de la capacidad de los demás.

A tener en cuenta



Acoplamiento

¿Qué es el acoplamiento?

Se refiere al grado de interdependencia entre los módulos o clases de un sistema.

Alto acoplamiento	Módulos fuertemente relacionados y dependen mucho unos de otros	Dificulta el mantenimiento y la escalabilidad del sistema

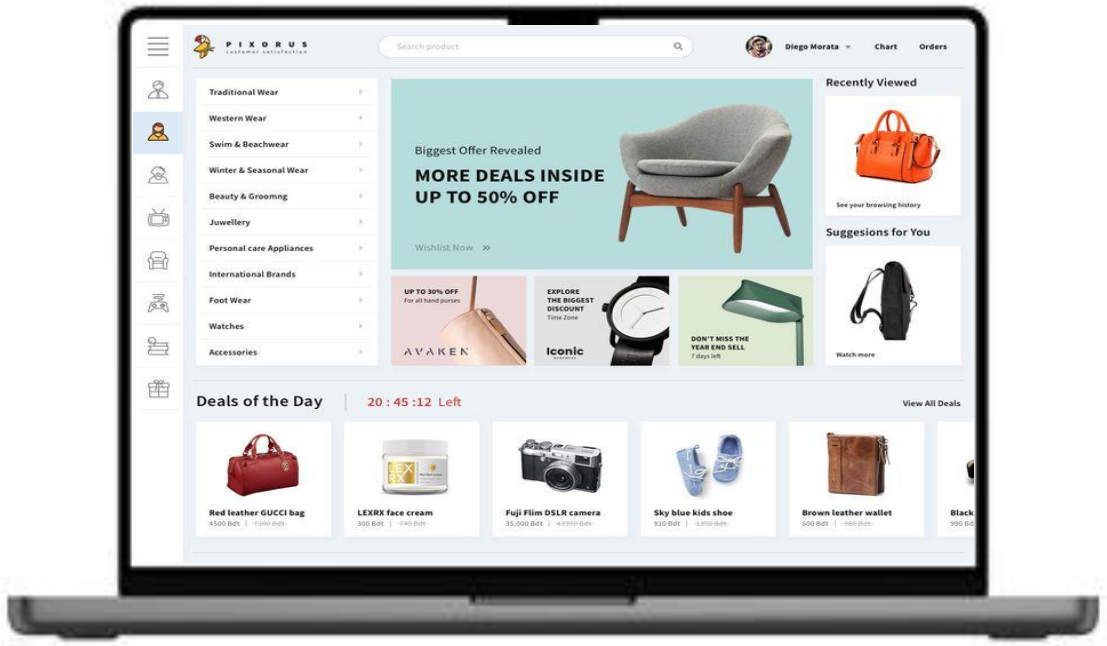
Bajo acoplamiento	Módulos más independientes	flexibles y fáciles de modificar.

Acoplamiento

Caso de Estudio

Supongamos los siguientes requisitos para un sistema de venta online:

Un sistema e-commerce que recibe órdenes de compra y gestiona múltiples tareas.
Entre las **RESPONSABILIDADES** de una clase **ORDEN** se identifica la capacidad de:
procesar *pagos, enviar correos de confirmación y guardar la orden en una base de datos.*



Acoplamiento

Caso de Estudio

```
public class Orden
{


}
```

RESPONSABILIDAD 1

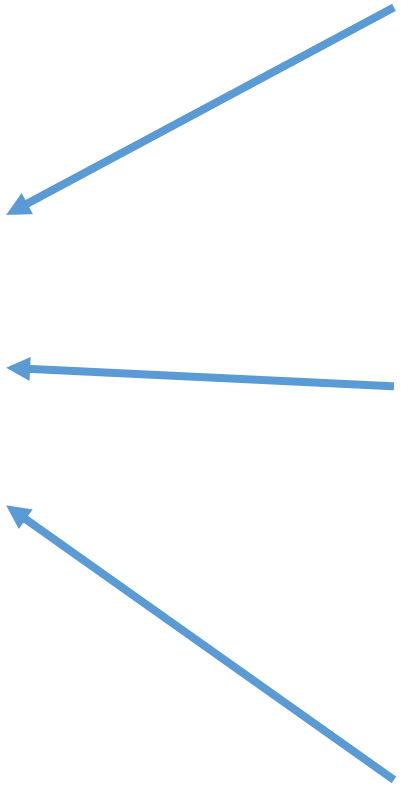
```
public void ProcesarPago(string numeroTarjeta)
{
        // Lógica para procesar el pago
}
```

RESPONSABILIDAD 2

```
public void EnviarCorreoConfirmacion(string correoCliente)
{
        // Lógica para enviar un correo electrónico
}
```

RESPONSABILIDAD 3

```
public void GuardarOrdenEnBaseDeDatos()
{
        // Lógica para guardar la orden en la base de datos
}
```



Acoplamiento

Caso de Estudio

```
public class Orden
{
   public void ProcesarPago(string numeroTarjeta)
   {
      // Lógica para procesar el pago
      Console.WriteLine("Procesando pago con la tarjeta: " +
numeroTarjeta);
   }

   public void EnviarCorreoConfirmacion(string correoCliente)
   {
      // Lógica para enviar un correo electrónico
      Console.WriteLine("Enviando correo de confirmación a: " +
correoCliente);
   }

   public void GuardarOrdenEnBaseDeDatos()
   {
      // Lógica para guardar la orden en la base de datos
      Console.WriteLine("Guardando la orden en la base de
datos.");
   }
}
```

Cosas que se pueden observar de la clase Orden:

Múltiples responsabilidades: procesar pagos, enviar correos y gestionar la base de datos.

Alto Acoplamiento: La clase Orden está acoplada directamente a los detalles de implementación del procesamiento de pagos, envío de correos y acceso a la base de datos. Si cualquiera de estos detalles cambia, la clase Orden debe modificarse, lo que complica el mantenimiento.

S.O.L.I.D.

¿Qué son los principios SOLID?

Los principios SOLID fueron popularizados por Robert C. Martin, conocido como "Uncle Bob", en la década de 2000. Aunque estos principios se basan en conceptos ya existentes en la programación orientada a objetos, Uncle Bob los agrupó y promovió como una guía esencial para el diseño de software robusto, escalable y fácil de mantener. El acrónimo SOLID fue acuñado por Michael Feathers

Objetivo

- Mejorar la Mantenibilidad.
- Promover la Reusabilidad.
- Reducir el Acoplamiento.
- Facilitar el Desarrollo Ágil.

S.O.L.I.D

El acrónimo proviene de:

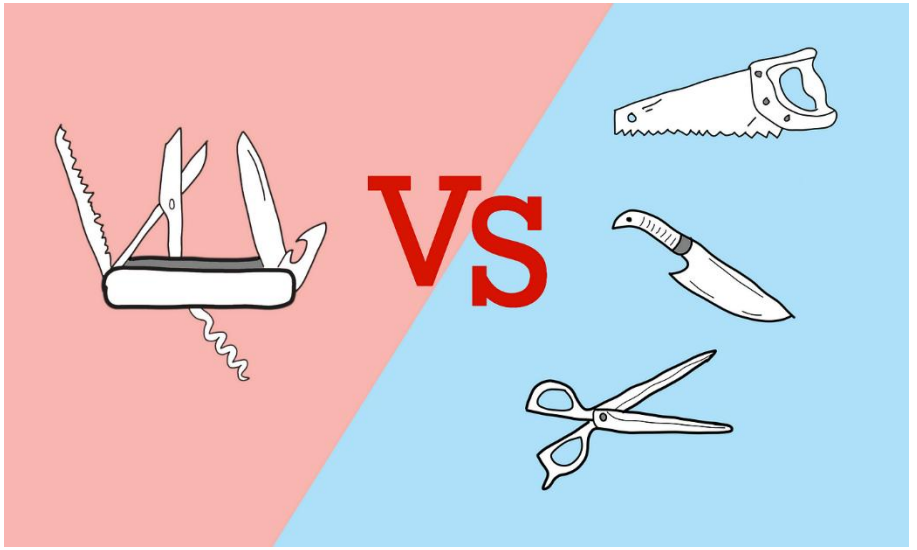
- **S**ingle Responsibility Principle (SRP): Principio de Responsabilidad Única
- **O**pen/Closed Principle (OCP): Principio de Abierto/Cerrado
- **L**iskov Substitution Principle (LSP): Principio de Sustitución de Liskov
- **I**nterface Segregation Principle (ISP): Principio de Segregación de Interfaces
- **D**ependency Inversion Principle (DIP): Principio de Inversión de Dependencias

S.O.L.I.D

Single Responsibility Principle (SRP) o Principio de Responsabilidad Única

El **SRP** establece que **una clase debe tener una única responsabilidad** o, en otras palabras, **una sola razón para cambiar**. Esto significa que cada clase debe centrarse en hacer solo una cosa y hacerlo bien.

Este principio ayuda a mantener el **código más limpio, modular y fácil de mantener**.



S.O.L.I.D

Single Responsibility Principle (SRP) o Principio de Responsabilidad Única

Caso de estudio

Supongamos que se nos presenta una biblioteca y nos dice que necesita un software para gestionar la información de libros que le pertenecen. Los Libros tienen Titulo, Autor, Paginas y necesita que recuperar los datos de un archivo CSV y a la vez poder guardar cuando se sume un nuevo libro a la biblioteca.

S.O.L.I.D

Single Responsibility Principle (SRP) o Principio de Responsabilidad Única

```
public class Libro
{
    public string Titulo { get; set; }
    public string Autor { get; set; }
    public int Paginas { get; set; }

    // Constructor
    public Libro(string titulo, string autor, int paginas)
    {
        Titulo = titulo; Autor = autor; Paginas = paginas;
    }

    public void GuardarEnArchivo(string rutaArchivo, Libro libro)
    {
         File.WriteAllText(rutaArchivo, contenido);
    }

   public static List<Libro> CargarDesdeArchivo(string rutaArchivo)
    {
         string contenido = string contenido = File.ReadAllText(rutaArchivo);
          List<Libro> Libros = new Libros<Libro>();
        foreach (var linea in lineas) {
                        string[] partes = Contenido.Split(',');
                        Libros.Add(new Libro(partes[0], partes[1], int.Parse(partes[2])));
         }
        return Libros;
    }
  // Enviar datos
  public void EnviarCorreoConfirmacion(string correoCliente, string contenido)
  {
                        // Lógica para enviar un correo electrónico
  }
}
```

Información de un Libro

Proceso de guardado y lectura de Libros

Envio de confirmación de email

S.O.L.I.D

Single Responsibility Principle (SRP) o Principio de Responsabilidad Única

```
public class Libro
{
    public string Titulo { get; set; }
    public string Autor { get; set; }
    public int Paginas { get; set; }

    // Constructor
    public Libro(string titulo, string autor, int paginas)
    {
        Titulo = titulo;
        Autor = autor;
        Paginas = paginas;
    }
}
```

Información de **UN** Libro

La clase que solo maneja la información del libro

S.O.L.I.D

Single Responsibility Principle (SRP) o Principio de Responsabilidad Única

```
public class GestorLibros
{
   // Guarda una lista de libros en un archivo
   public void GuardarEnArchivo(List<Libro> libros, string rutaArchivo)
   {
      var lineas = new List<string>();
      foreach (var libro in libros)
      {
         lineas.Add(libro.ToString());
      }
      File.WriteAllLines(rutaArchivo, lineas);
   }

   // Carga una lista de libros desde un archivo
   public List<Libro> CargarDesdeArchivo(string rutaArchivo)
   {
      var libros = new List<Libro>();
      var lineas = File.ReadAllLines(rutaArchivo);

      foreach (var linea in lineas)
      {
         var partes = linea.Split(',');
         if (partes.Length == 3 && int.TryParse(partes[2], out int paginas))
         {
            libros.Add(new Libro(partes[0], partes[1], paginas));
         }
      }
      return libros;
   }
}
```

Proceso de guardado y lectura de Libros

La clase que solo maneja la persistencia de los libros

S.O.L.I.D

Single Responsibility Principle (SRP) o Principio de Responsabilidad Única

```
public class EmailServices
{
  //Constructor de el servicio
  public EmailService(string credenciales)
  {
  }

  // Enviar datos
  public void EnviarCorreoConfirmacion(string correoCliente, string contenido)
  {
                    // Lógica para enviar un correo electrónico
  }
}
```

Proceso de envíos de emails

La clase que solo maneja lo relacionado al envío de emails

S.O.L.I.D

Single Responsibility Principle (SRP) o Principio de Responsabilidad Única

```
public class Orden
{
  //Constructor de el servicio
  public ProcesarOrden()
  {

  }

  // Enviar datos
  public void ProcesarOrden()
  {
          Libro = new Libro("libro","autor", "Paginas" );
          GestorLibros.GuardarEnArchivo(Libro);
          EmailService email = new EmailService();
          email. EnviarCorreoConfirmacion(Usuario@usuario.com, "Se agregó un nuevo libro a la biblioteca");
   }

}
```

Proceso el conjunto de tareas

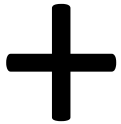
La clase conoce el proceso general y los objetos necesarios pero no como conoce resuelve cada clase su tarea.

S.O.L.I.D

Single Responsibility Principle (SRP) o Principio de Responsabilidad Única

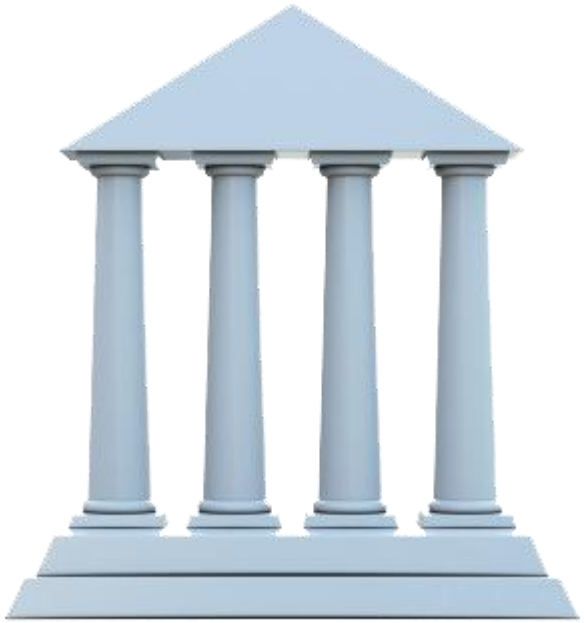
Una clase tiene que tener una sola razón para cambiar

Una clase o método solo tiene que tener una responsabilidad



Principio de Responsabilidad Única

Pilares de la POO



Abstracción

Encapsulamiento

Herencia

Polimorfismo

Herencia

¿Que es herencia en la POO?

Es un tipo de relación donde un objeto es creado a partir de otros objetos ya existentes, obteniendo todas las características (métodos y atributos) de la clase base.

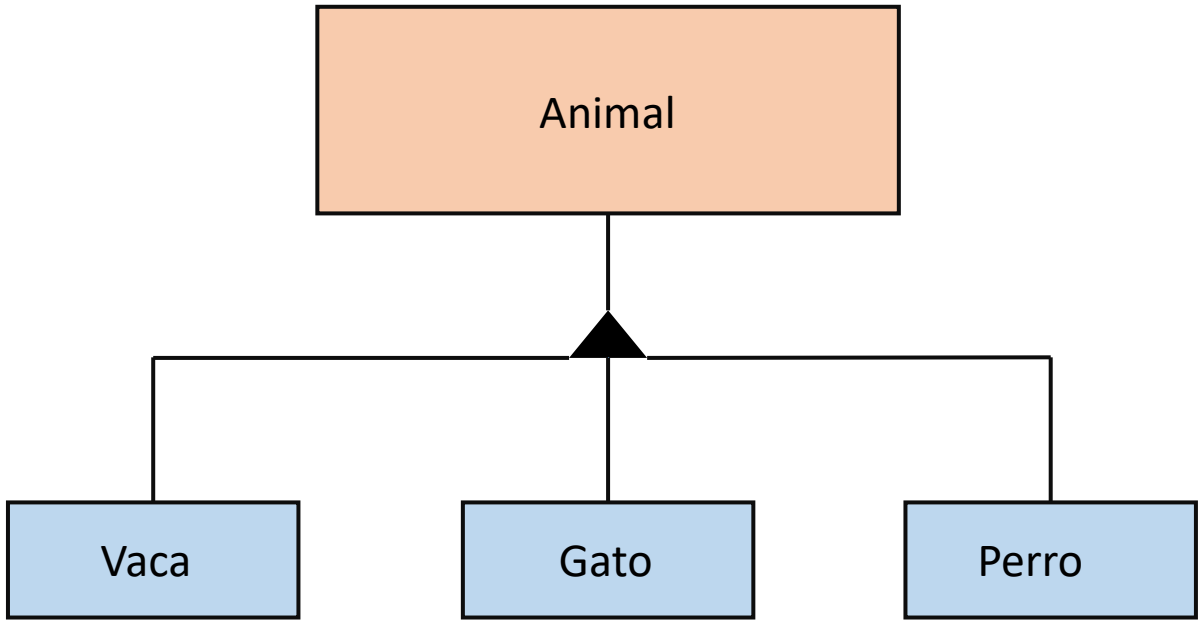
Normalmente, nos referimos a este tipo de relación con la expresión: “Es un”

Ej: Auto es un Vehiculo

Ej: Vaca es un Animal

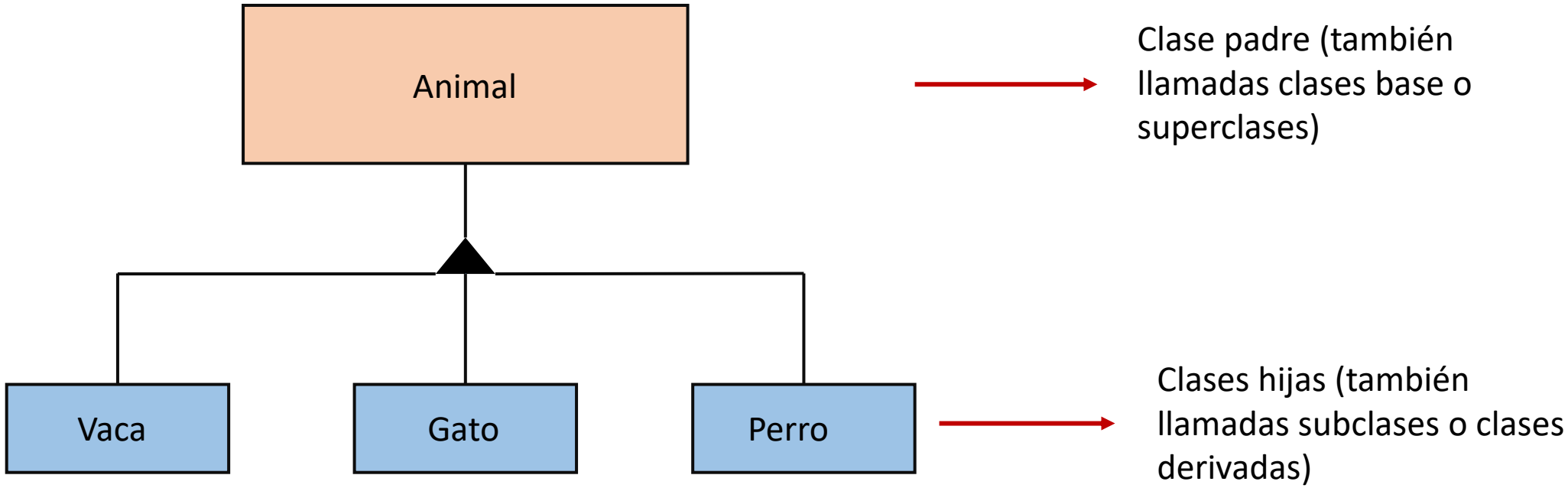
Beneficios

- Reutilización
- Polimorfismo



Herencia

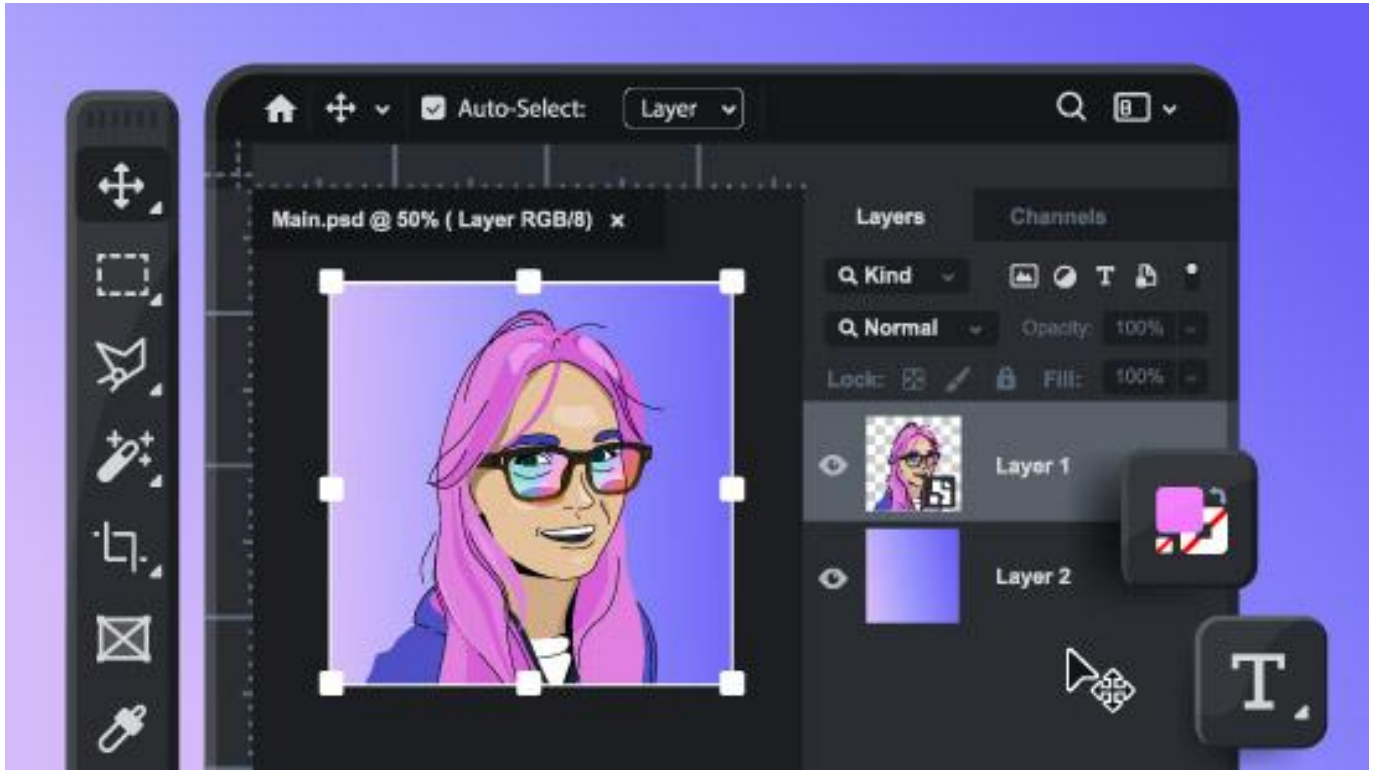
¿Que es herencia en la POO?



Herencia

Caso de ejemplo de herencia

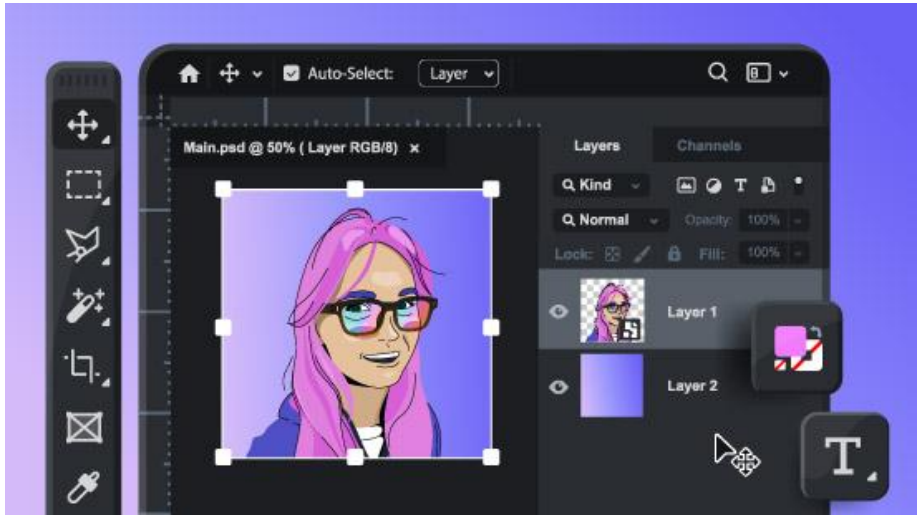
Supongamos una herencia para un sistema que permita la edición de contenidos gráficos, como puede ser por ejemplo: CorelDraw, PowerPoint o Adobe Illustrator

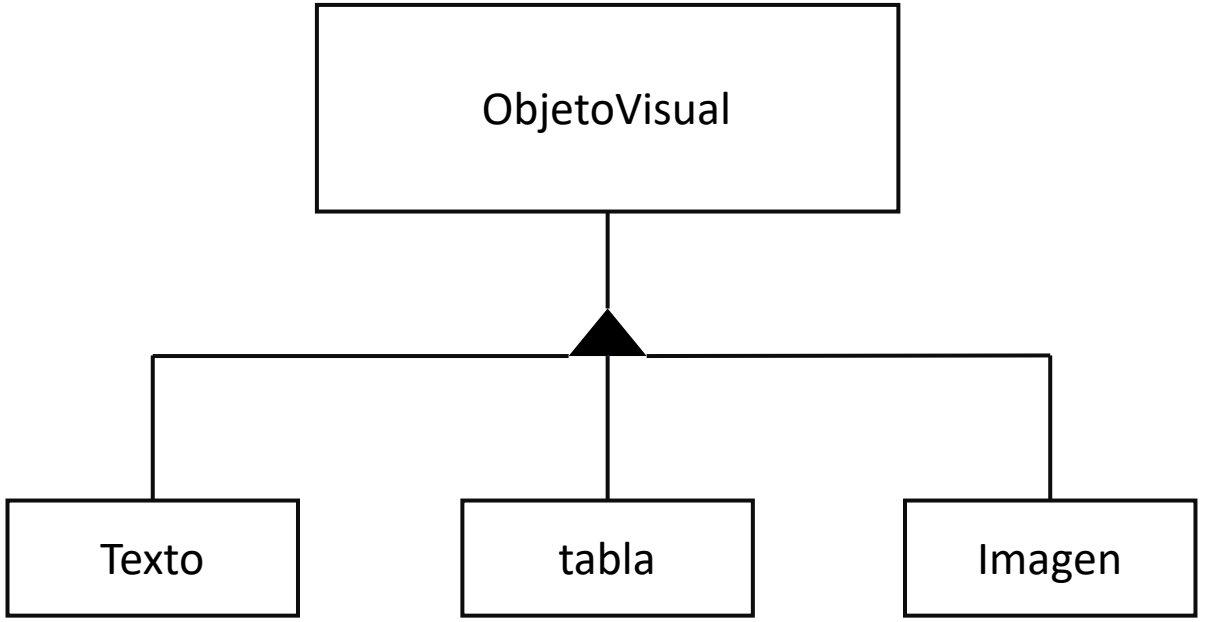


Herencia

Caso de ejemplo de herencia

La herencia quedaría planteada en los términos de diagrama de la izquierda donde **ObjetoVisual** es el objeto base del que herendan las clases **Texto**, **Tabla** e **Imagen, etc**.





Herencia

Herencia en C#

En C# la herencia se especifica de la *Clase Base* a la *Clase derivada* utilizando el operador ":" (dos puntos).

Class **ClaseDerivada** : **ClaseBase**
{
 […] // procesos de la clase derivada
}

En el siguiente ejemplo se puede ver:
La clase **Texto** hereda de la clase **ObjetoVisual**
La clase **Imagen** hereda de la clase **ObjetoVisual**

También podríamos leer:

- Texto **es un** ObjetoVisual
- Imagen **es un** ObjetoVisual

```
public class ObjetoVisual //Clase base
{
        public int X { get; set; }
        public int Y { get; set; }
        public ObjetoVisual()
        {
                […]
        }
}

public class Texto : ObjetoVisual  // Hereda de clase ObjetoVisual
{
        public Texto() :base()
        {
                […]
        }
}

public class Imagen : ObjetoVisual  // Hereda de clase ObjetoVisual
{
        public Imagen() :base()
        {
                […]
        }
}                                […]

ObjetoVisual Figura1 = new Texto();
Figura1.X = 10;
ObjetoVisual Figura2= new Imagen();
Figura2.X = 10;
```

Polimorfismo

¿Qué es el polimorfismo en la POO?

El polimorfismo suele considerarse el cuarto pilar de la programación orientada a objetos, después de la encapsulación y la herencia. Polimorfismo es una palabra griega que significa "con muchas formas" y tiene dos aspectos diferentes:

- En tiempo de ejecución, los objetos de una clase derivada pueden ser tratados como objetos de una clase base en lugares como parámetros de métodos y colecciones o matrices.

- Las clases base pueden definir e implementar *métodos* virtuales, y las clases derivadas pueden invalidarlos, lo que significa que pueden proporcionar su propia definición e implementación

Polimorfismo

Caso de estudio

En a la clase Figura se crea un método virtual llamado Dibujar (marcada con virtual).

Este método puede ser sobrescrito (override) en cada clase derivada para dibujar la forma determinada que la clase que corresponda.

```
public class ObjetoVisual
{
        public int X { get; set; }
        public int Y { get; set; }
        public ObjetoVisual ()
        {
                […]
        }
        public virtual void Dibujar()
        {
                Console.WriteLine(“método para dibujar”);
        }
}

public class Texto : ObjetoVisual
// hereda de clase figura
{
        Public override void Dibujar()
        {
                Console.WriteLine(“muestro texto”);
        }
}

public class Imagen : ObjetoVisual
// hereda de clase figura
{
        public override void Dibujar()
        {
                Console.WriteLine(“muestro Imagen”);
        }
}
```

Encapsulamiento

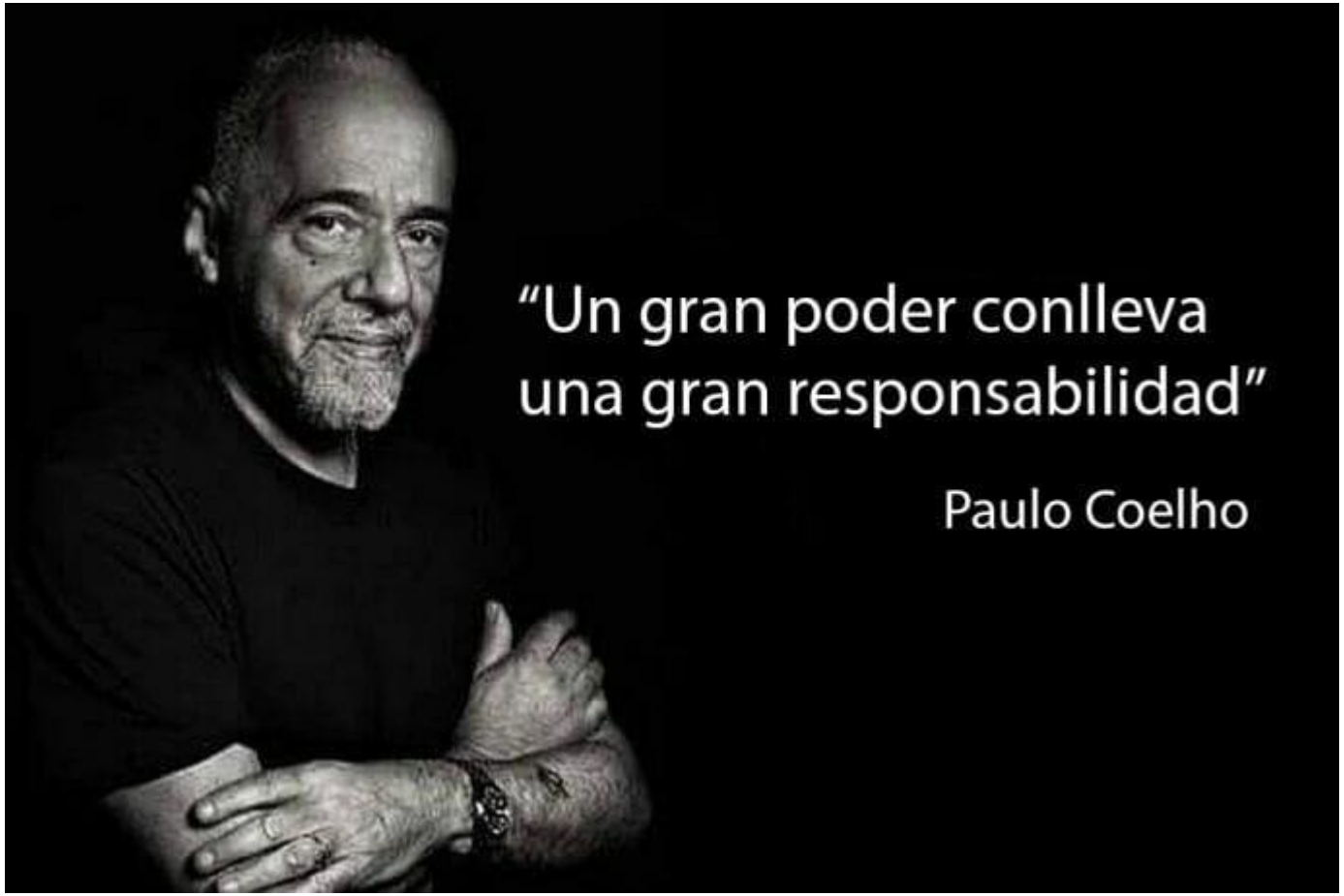
Modificador de acceso

Los principales modificadores de acceso son los siguiente:

- **public** : Accesible desde cualquier lugar del programa.
- **private**: Accesible desde la propia clase.
- **protected**: Accesible desde la propia clase y desde las clases derivadas.

Nota:
Existen otros modificadores de acceso (internal, protected internal, private protected), pero solo nos centraremos en los anteriores.

Problemas de abstracciones



Problemas de abstracciones

Liskov Substitution Principle

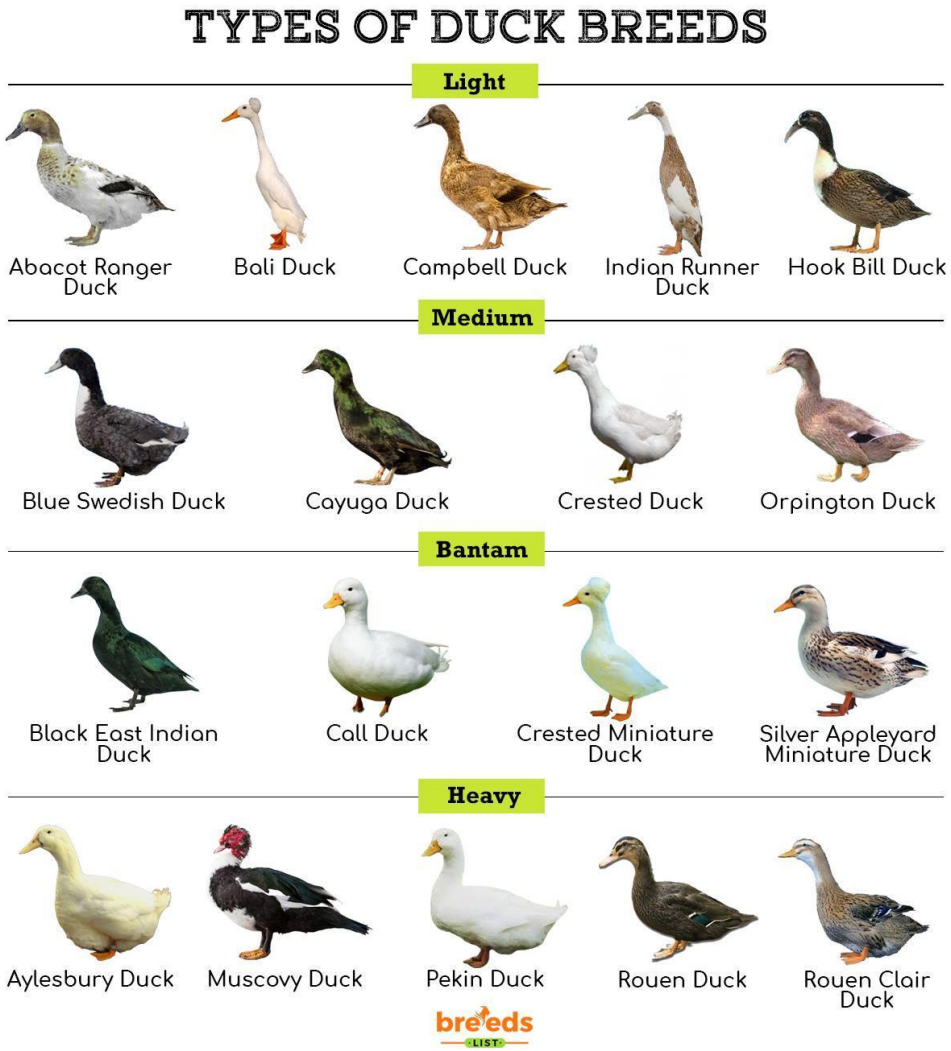
Si **S** es un subtipo de **T**, entonces los objetos de tipo **T** en un programa de computadora pueden ser sustituidos por objetos de tipo **S** (es decir, los objetos de tipo S pueden sustituir objetos de tipo T), sin alterar ninguna de las propiedades deseables de ese programa (la corrección, la tarea que realiza, etc.).

Esto implica que:

- Los objetos de la clase derivada deben comportarse de una manera coherente con las promesas hechas en el contrato de la clase base.
- Las clases pueden extenderse más fácilmente sin riesgo de romper el comportamiento del sistema.
- La clase derivada puede reimplementar solamente los métodos públicos de la clase padre.
- Diseño en el que las dependencias entre clases son más débiles, lo que reduce el acoplamiento

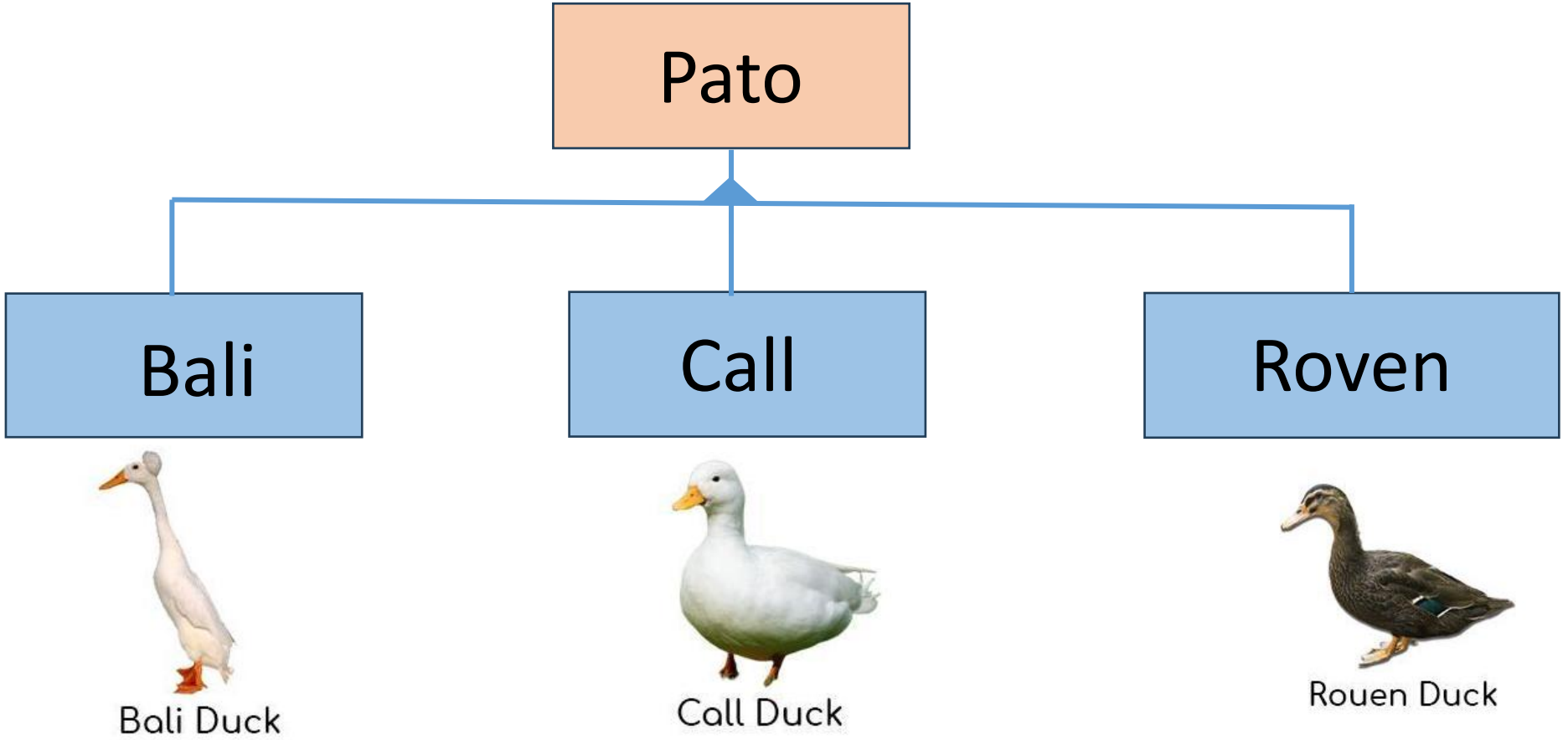
Problemas de abstracciones

Liskov Substitution Principle



Problemas de abstracciones

Liskov Substitution Principle



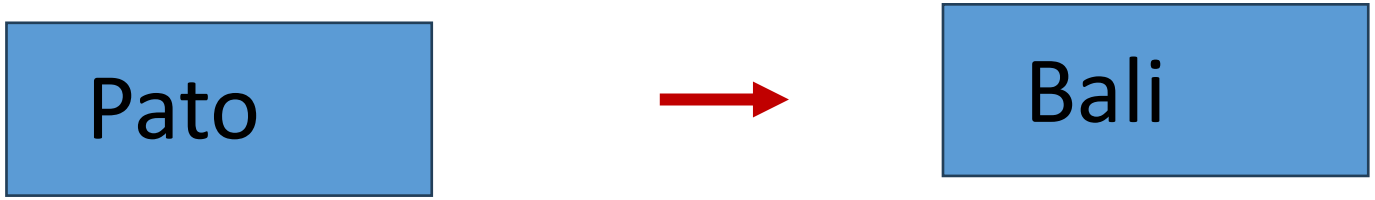
https://www.youtube.com/watch?v=etqJV_ZHPbQ

https://www.youtube.com/watch?v=0Da8ZhKcNKQ

https://www.youtube.com/watch?v=m0EuUjqBM1w

Problemas de abstracciones

Liskov Substitution Principle

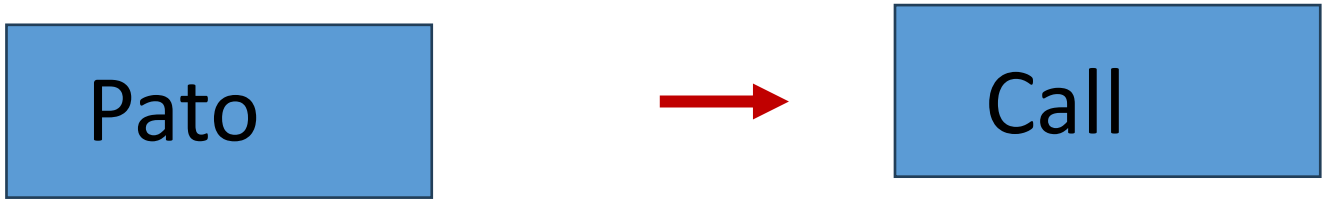



```
class Pato
{
    public virtual void Quack(){
    }

    public virtual void Volar(){
    }
}
```

```
class Bali : Pato
{
    public override void Quack(){
        Console.WriteLine("Quack quack!");
    }

    public override void Volar(){
        Console.WriteLine(“Volando como un campeón.");
    }
}
```

Problemas de abstracciones

Liskov Substitution Principle




```
class Pato
{
    public virtual void Quack(){
    }

    public virtual void Volar(){
    }
}
```

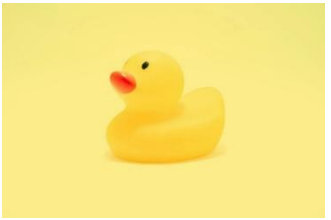
```
class Call : Pato
{
    public override void Quack(){
        Console.WriteLine("Quack quackock!");
    }

    public override void Volar(){
        Console.WriteLine(“Vuelo en manada y despacito como
un campeón.");
    }
}
```

Problemas de abstracciones

Liskov Substitution Principle

```
class PatoBali : Pato
{
    public override void Quack(){
        Console.WriteLine("Quack quack!");
    }

    public override void Volar(){
        Console.WriteLine(“Volando.");
    }
}
```

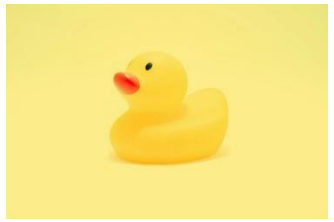
≠



```
class PatoJuguete : Pato
{
    public override void Quack() {
        Console.WriteLine(""Squeak squeak!"!");
    }

    public override void Volar() {
        Console.WriteLine(“NO VUELO.");
    }
    public override void CargarPilas(){
        Console.WriteLine(“Cargando Pilas.");
    }
}
```

Problemas de abstracciones

Liskov Substitution Principle



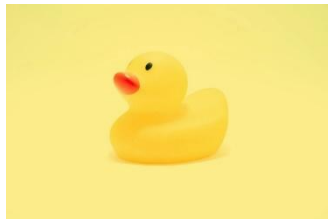
```
class PatoJuguete : Pato
{
    public override void Quack() {
        Console.WriteLine(""Squeak squeak!"!");
    }

    public override void Volar() {
        Console.WriteLine("NO VUELO.");
    }
    public override void CargarPilas(){
        Console.WriteLine("Cargando Pilas.");
    }
}
```

¿Cómo CARGO LAS PILAS?

¿Puedo usar Volar?

Problemas de abstracciones

Liskov Substitution Principle



¿Cómo CARGO LAS PILAS?

```
class PatoJuguete : Pato
{
    public override void Quack() {
        Console.WriteLine("Squeak squeak!!");
    }

    public override void Volar() {
        Console.WriteLine(“NO VUELO.");
    }
    public override void CargarPilas(){
        Console.WriteLine(“Cargando Pilas.");
    }
}
```

```
Pato patito = new PatoJuguete();
patito.Quack();
patito.CargarPilas();
```

¡NO LO PUEDO USAR!

¡NO HACE NADA AQUÍ!

Problemas de abstracciones

Liskov Substitution Principle

La clave para evitar el "Problema de la Abstracción Incorrecta" es crear abstracciones que se ajusten de manera natural y coherente a la lógica del problema que estás resolviendo.

En algunos casos, puede ser más apropiado crear abstracciones separadas para casos diferentes, en lugar de forzar una única jerarquía de clases.

> "Los objetos de una superclase deben poder ser reemplazados por objetos de sus subclases sin romper la aplicación."

Problemas de abstracciones

Liskov Substitution Principle

En el ejemplo de los patos, podría ser más claro y sencillo tratar los patos de goma como objetos independientes sin intentar forzarlos a encajar en la misma jerarquía que los patos reales.

```
public class Juguete
{
    public virtual void Sonido() {}

    public virtual void HacerLuces() {}

    public virtual void Flotar() {}

    public virtual void CargarPilas(){}
}
```

```
public class PatoDeGoma : Juguete
{
    public override void Sonido() {
        Console.WriteLine("Squeak squeak!!");
    }
    public override void HacerLuces() {
        Console.WriteLine(“¡¡Lucecitas!!");
    }
    public override void Flotar() {
        Console.WriteLine(“flotando libremente");
    }
    public override void CargarPilas(){
        Console.WriteLine(“Cargando Pilas.");
    }
}
```

Problemas de abstracciones

Conclusión de Liskov Substitution Principle

Esto implica que:

- Aplicar Liskov asegura que las subclases pueden ser sustituidas por sus superclases sin alterar la funcionalidad del sistema.

- Los objetos de la clase derivada deben comportarse de una manera coherente con las promesas hechas en el contrato de la clase base.

Recordando

Para referirnos a las relaciones de las clases diremos que:

- La **Herencia** nos dice *es un*.
- La **Composición** nos dice *es parte de*.
- La **Agregación** nos dice *tiene un*.

"Los objetos de una superclase deben poder ser reemplazados por objetos de sus subclases sin romper la aplicación."

Ejercicio: Simulando construcción de vehículos

Suponga que se requiere modelar objetos vehículos considerando:

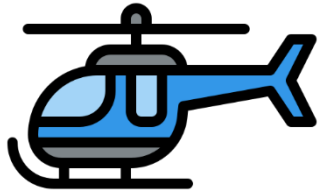
- Un Avión
- Un Auto
- Una Bicicleta

Considere los siguientes métodos:

- Motor(string sonido),
- Conducir()
- ApagarMotor(string sonido)
- Volar()

Ejercicio: Simulando construcción de vehículos

- EncenderMotor(string sonido),
- Conducir()
- ApagarMotor(string sonido)
- Volar()



Tiene motor	**No Tiene Motor**	**Tiene Motor**
Conduce	Se puede Conducir	Se Puede conducir
No Puede volar	No puede volar	Puede volar