[7507 / 9502]

Algoritmos y Programación III

Principios SOLID

Diseño orientado a objetos

*Tomás Bustamante

Principios de diseño

Conjunto de reglas que favorecen y/o fomentan la escritura de código extensible y mantenible, minimizando los posibles códigos afectados. Algunos ejemplos:

- General Responsibility Assignment Software Patterns (GRASP).
- Tell-Don't-Ask.
- Don't repeat yourself (DRY).
- **S.O.L.I.D.**

Principios SOLID

SRP: Principio de responsabilidad única

OCP: Principio de abierto/cerrado

LSP: Principio de sustitución de Liskov

ISP: Principio de segregación de la interfaz

DIP: Principio de inversión de dependencias

Principio de responsabilidad única

Una clase debe tener una única razón para cambiar.

A module should be responsible to one, and only one, **actor.*

"Todo gran poder conlleva una gran responsabilidad"

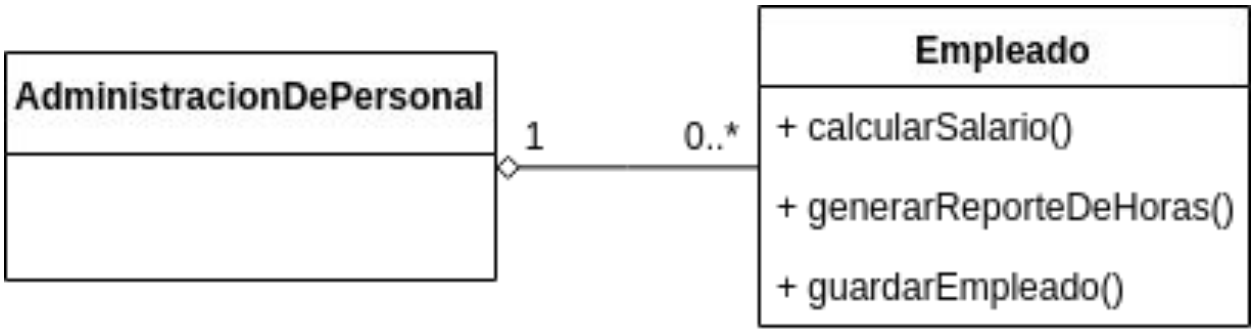
- Gandalf

- Una refactorización o un arreglo de un bug no es una razón de cambio.
- Las razones de cambio son cambios de los ~~requerimientos~~ requisitos.

**SRP: THE SINGLE RESPONSIBILITY PRINCIPLE.*
Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series)

Principio de responsabilidad única

Ejemplo: ¿Cuántas razones de cambio tiene la clase Empleado?

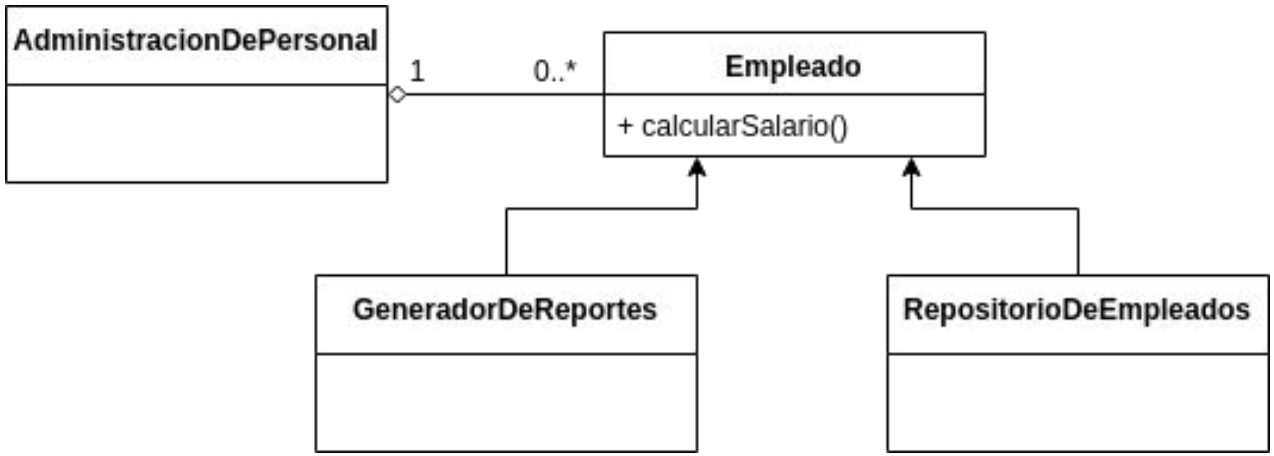


Principio de responsabilidad única

¿Cómo se soluciona?

Creando más clases para cada responsabilidad (hay muchas maneras de hacerlo).

Una posible solución:



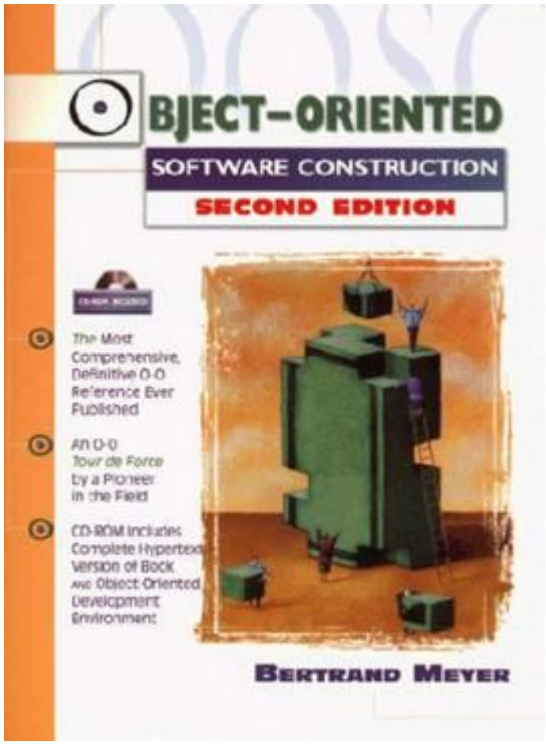
Más información:

- Paper original de Robert Martin
- Entrada en el blog de Robert Martin

Principio de abierto/cerrado

Las clases deben estar abiertas para la extensión pero cerradas para su modificación. (Bertrand Meyer)

- Se debe poder cambiar el comportamiento sin modificar el código ya existente.
- Se puede lograr utilizando herencia o delegación.



Más información:

- Artículo original de Robert Martin
- Entrada en el blog de Robert Martin

Principio de abierto/cerrado

```
public enum TipoFigura {
  CIRCULO, CUADRADO
}
```

```
public abstract class Figura
{
  TipoDeFigura tipo;
  // ...
}
```

```
public class Circulo extends Figura {
  TipoFigura tipo = TipoFigura.CIRCULO;
  double radio;
  Punto centro;
  // ...
}
```

```
public class Cuadrado extends Figura {
  TipoFigura tipo = TipoFigura.CUADRADO;
  double lado;
  Punto verticeSuperiorIzquierdo;
  // ...
}
```

```
public class Dibujante {

  void dibujarFiguras(Figura[] figuras, int n) {
    for(int i = 0; i < n; i++) {
      Figura figura = figuras[i];
      switch (figura.tipo) {
        case Figura.CIRCULO:
          dibujarCirculo((Circulo) figura);
          break;
        case Figura.CUADRADO:
          dibujarCuadrado((Cuadrado) figura);
          break;
      }
    }
  }

  void dibujarCirculo(Circulo circulo) {
    // ...
  }

  void dibujarCuadrado(Cuadrado cuadrado) {
    // ...
  }
}
```

Principio de abierto/cerrado

Una posible solución:

```
public interface Figura {
  void dibujar();
  // ...
}
```

```
public class Circulo implements Figura {
  @Override
  public void dibujar() {
    // ...
  }
}
```

```
public class Cuadrado implements Figura {
  @Override
  public void dibujar() {
    // ...
  }
}
```

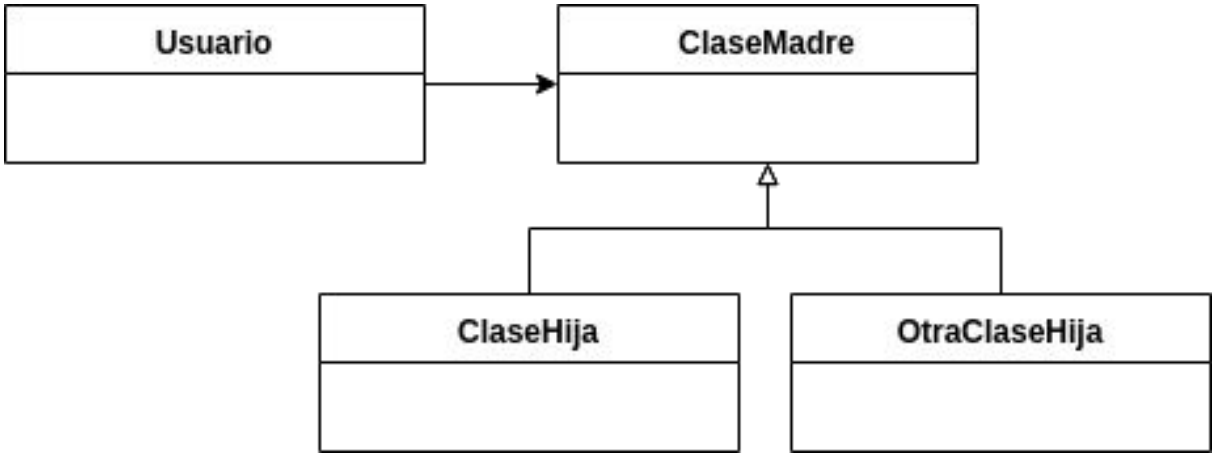
```
import java.util.List;

public class Dibujante {

  void dibujarFiguras(List<Figura> figuras) {
    for(Figura figura : figuras) {
      figura.dibujar();
    }
  }
}
```

```
public class Triangulo implements Figura {
  @Override
  public void dibujar() {
    // ...
  }
}
```

Principio de sustitución de Liskov

Las clases heredadas deben poder ser utilizadas a través de su clase madre sin la necesidad de que el usuario sepa la diferencia.

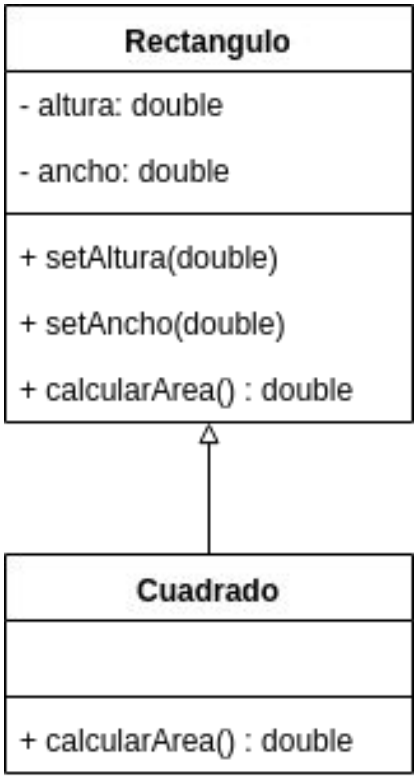
- Se debe cumplir la condición “es un” al aplicar herencia.



Más información:

- Artículo original de Robert Martin

Principio de sustitución de Liskov




```
public class Cuadrado extends Rectangulo {

  @Override
  public void setAltura(double altura) {
    this.altura = altura;
    this.ancho = altura;
  }

  @Override
  public void setAncho(double ancho) {
    this.altura = ancho;
    this.ancho = ancho;
  }
}
```

¿Qué pasa si a este método le pasamos un cuadrado en lugar de un rectángulo?

```
// ...
public double modificarRectangulo(Rectangulo rectangulo) {
  rectangulo.setAncho(4);
  rectangulo.setAltura(5);
  return rectangulo.calcularArea();
}
// ...
```

FACULTAD DE INGENIERIA
Universidad de Buenos Aires

algo3

Principio de segregación de la interfaz

Los clientes no deben ser forzados a depender de métodos que no utilizan.

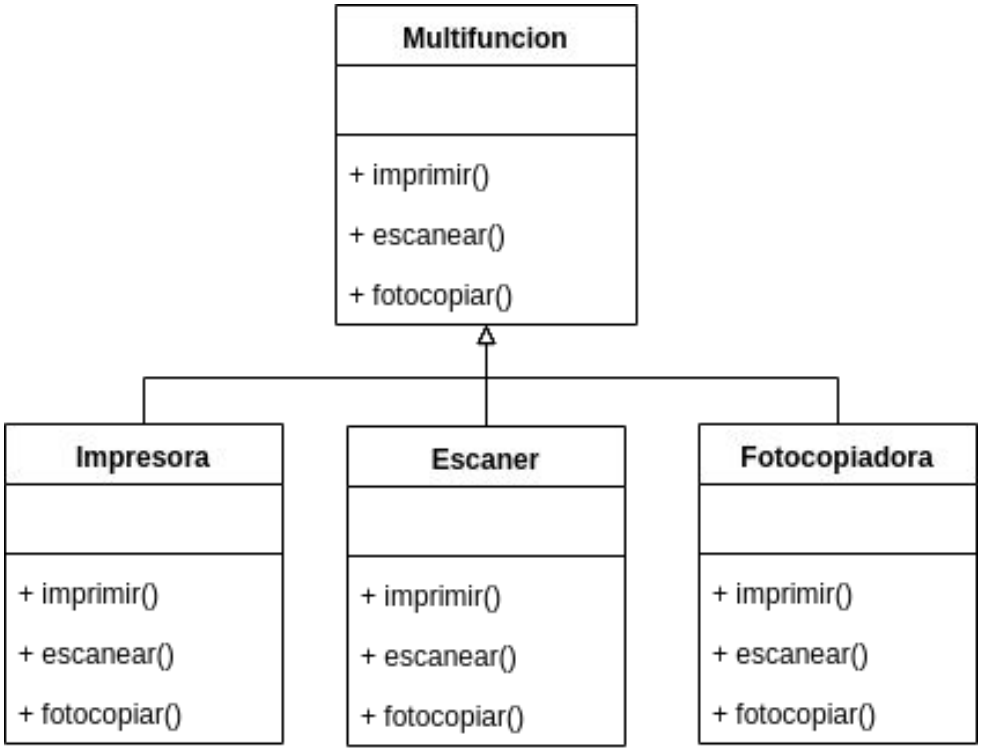
- Muchas interfaces específicas son mejores que una interfaz de propósito general.
- Es necesario aplicarlo cuando se tiene una clase con varios métodos, de los cuales solamente me interesan algunos.

Más información:

- Artículo original de Robert Martin

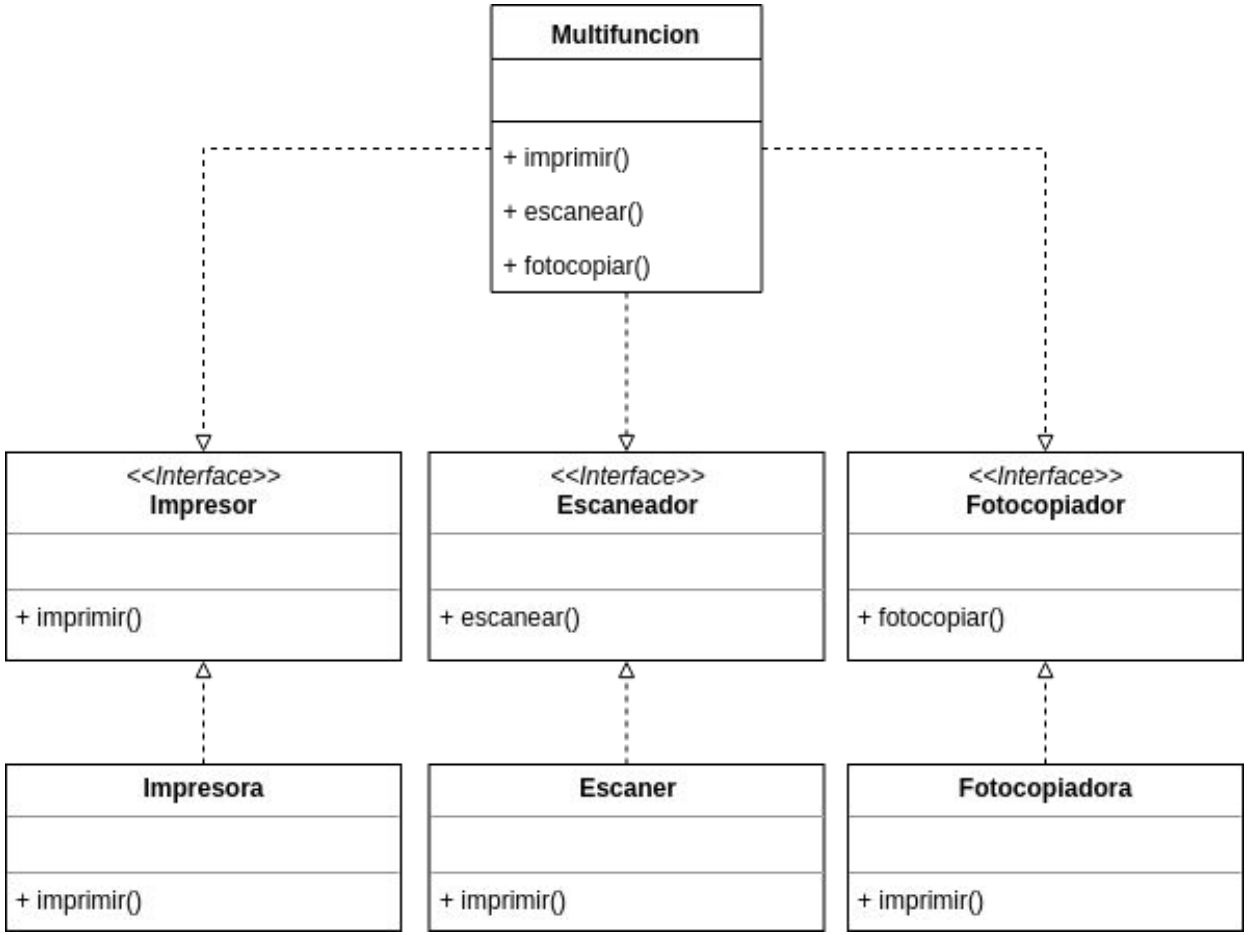
Principio de segregación de la interfaz

Un mal ejemplo:



Principio de segregación de la interfaz

Una posible solución:

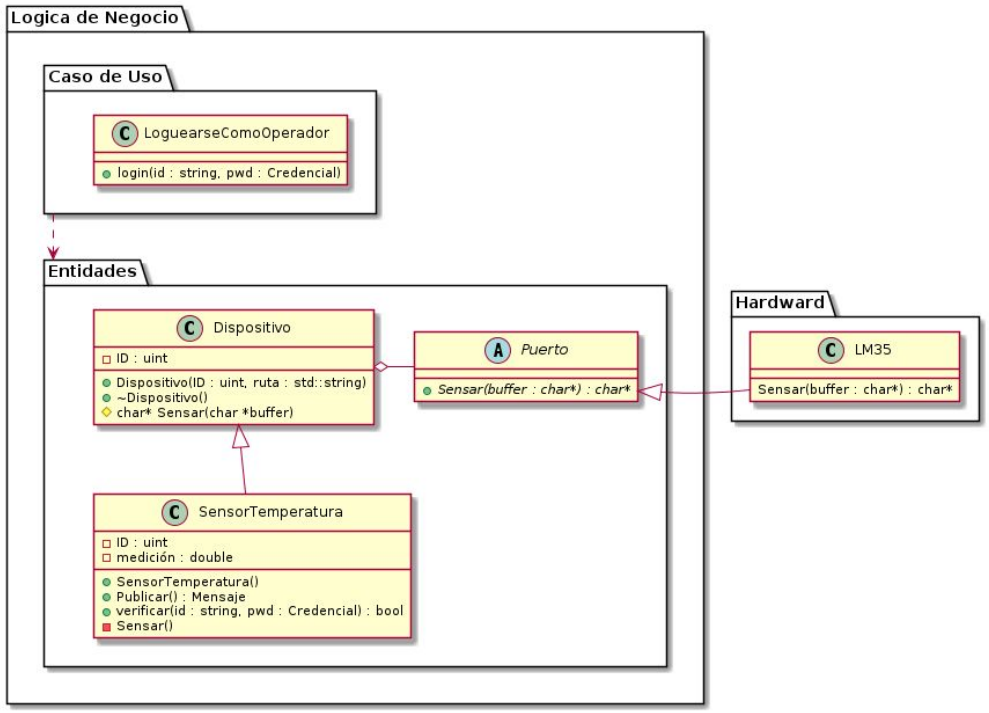


Principio de inversión de dependencia

Se debe depender de las abstracciones y no de las implementaciones.

Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.

Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.



Más información:

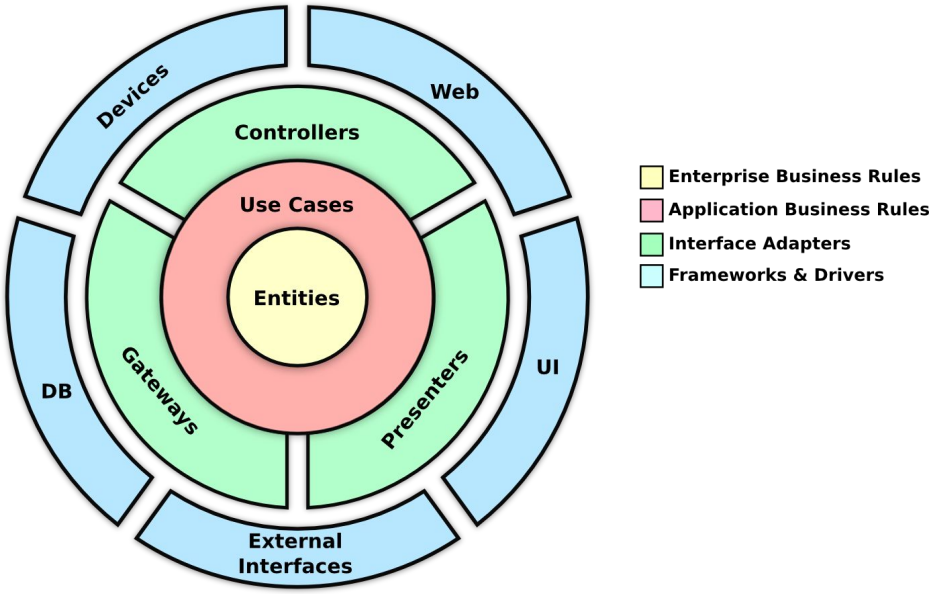
- Artículo de Robert Martin

Principio de inversión de dependencia

Se debe depender de las abstracciones y no de las implementaciones.

Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de abstracciones.

Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.



Más información:

- Artículo de Robert Martin