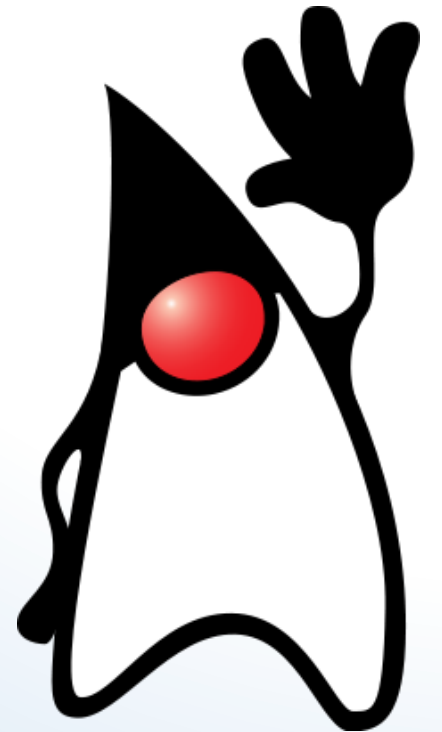
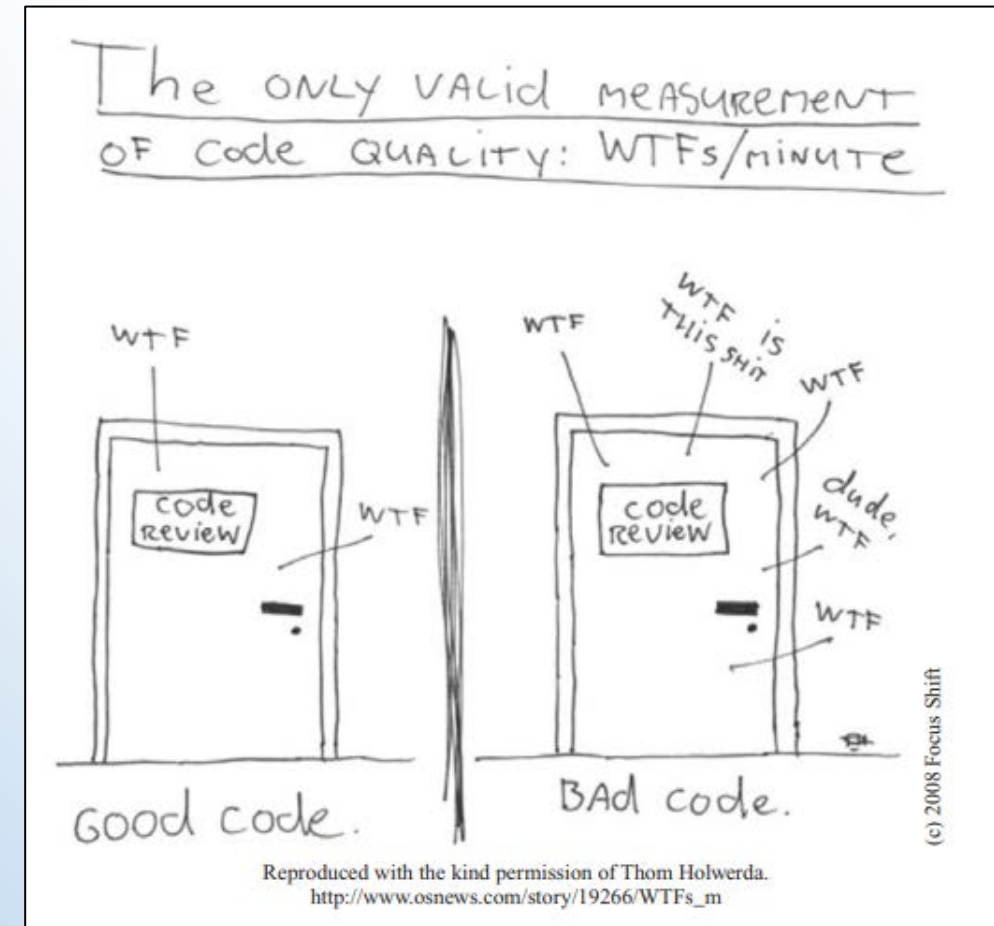


Principios de diseño



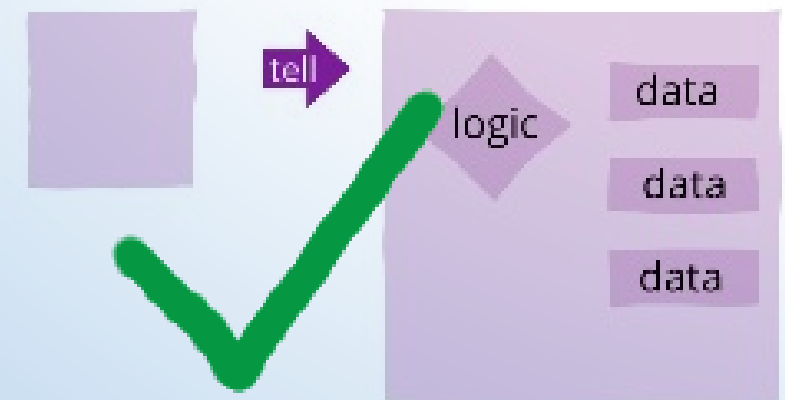
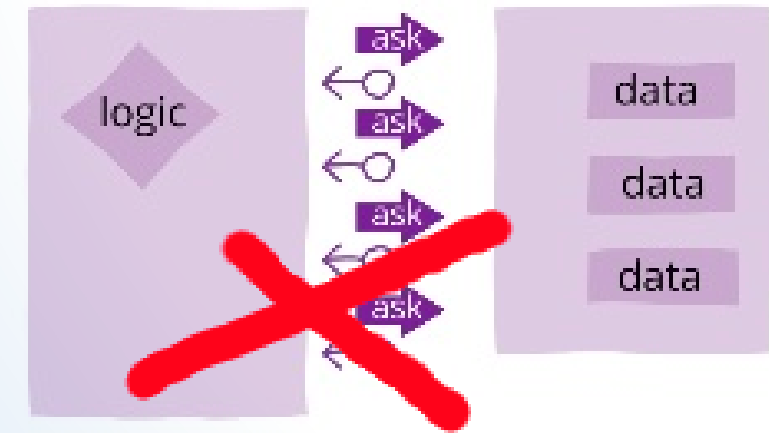
- TDA (*Tell, Don't Ask!*)
- PoLC (*Principle of Least Commitment*)
- PoLA (*Principle of Least Astonishment*)
- PoLK (*Principle of Least Knowledge*)
- DRY (*Don't Repeat Yourself*)
- YAGNI (*You Ain't Gonna Need It*)
- KISS (*Keep It Simple, Stupid!*)
- EDP (*Explicit Dependencies Principle*)
- KOP (*Knuth's Optimization Principle*)
- SoC (*Separation of Concerns Principle*)
- SOLID (*SRP, OCP, LSP, ISP, DIP*)
- FIRST (*Fast, Independent, Repeatable, Self-checking, Timely*)



TDA: Tell, don't ask!

¡Di qué hacer, no preguntes!

Principio de diseño orientado a objetos en el que se basa la **buena práctica** de evitar solicitarle a un objeto que indique su estado y luego llevar a cabo una acción basándose en este, en lugar de solicitarle al objeto que lleve a cabo la acción él mismo.

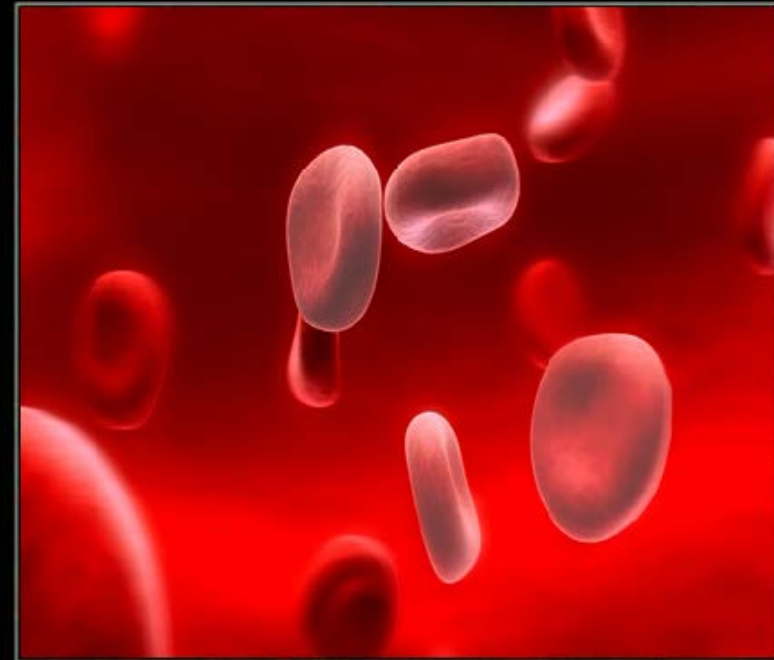


¿Cuándo se viola el principio TDA?



FLAGS OVER OBJECTS

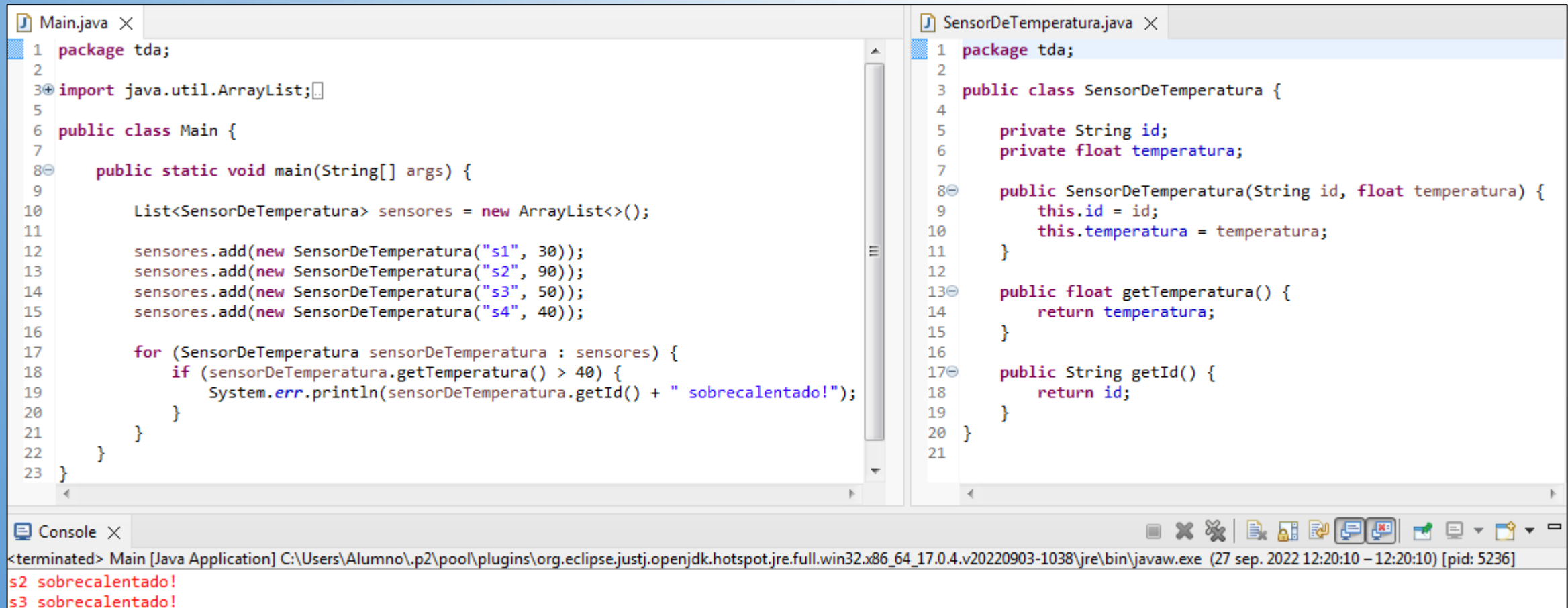
So easy you can add another twenty options.



Anemic Domain Model

Domain Model where Domain Objects contain little or no business logic

Ejemplo donde se viola el principio TDA:



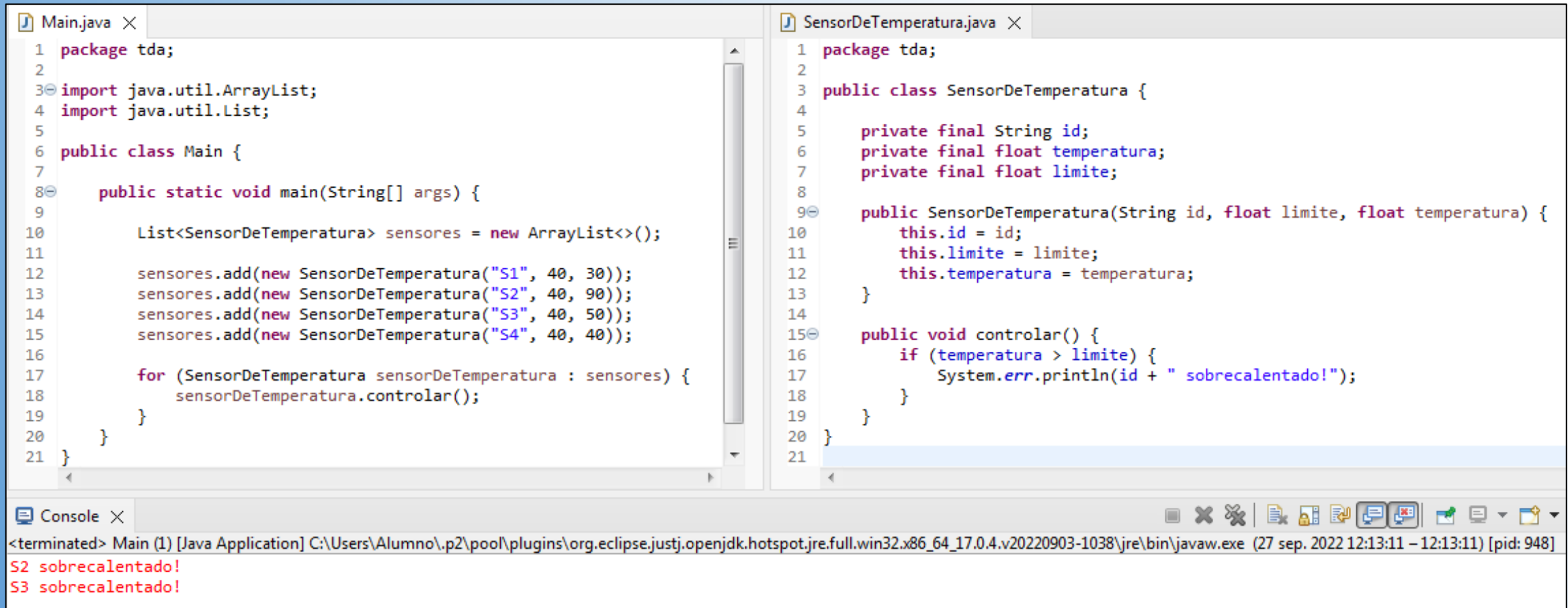
```

Main.java
1 package tda;
2
3 import java.util.ArrayList;
4
5
6 public class Main {
7
8     public static void main(String[] args) {
9
10         List<SensorDeTemperatura> sensores = new ArrayList<>();
11
12         sensores.add(new SensorDeTemperatura("s1", 30));
13         sensores.add(new SensorDeTemperatura("s2", 90));
14         sensores.add(new SensorDeTemperatura("s3", 50));
15         sensores.add(new SensorDeTemperatura("s4", 40));
16
17         for (SensorDeTemperatura sensorDeTemperatura : sensores) {
18             if (sensorDeTemperatura.getTemperatura() > 40) {
19                 System.err.println(sensorDeTemperatura.getId() + " sobrecalentado!");
20             }
21         }
22     }
23 }

SensorDeTemperatura.java
1 package tda;
2
3 public class SensorDeTemperatura {
4
5     private String id;
6     private float temperatura;
7
8     public SensorDeTemperatura(String id, float temperatura) {
9         this.id = id;
10        this.temperatura = temperatura;
11    }
12
13    public float getTemperatura() {
14        return temperatura;
15    }
16
17    public String getId() {
18        return id;
19    }
20 }

Console
<terminated> Main [Java Application] C:\Users\Alumno\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20220903-1038\jre\bin\javaw.exe (27 sep. 2022 12:20:10 - 12:20:10) [pid: 5236]
s2 sobrecalentado!
s3 sobrecalentado!
```

Ejemplo donde se respeta el principio TDA:



```

Main.java
1 package tda;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Main {
7
8     public static void main(String[] args) {
9
10         List<SensorDeTemperatura> sensores = new ArrayList<>();
11
12         sensores.add(new SensorDeTemperatura("S1", 40, 30));
13         sensores.add(new SensorDeTemperatura("S2", 40, 90));
14         sensores.add(new SensorDeTemperatura("S3", 40, 50));
15         sensores.add(new SensorDeTemperatura("S4", 40, 40));
16
17         for (SensorDeTemperatura sensorDeTemperatura : sensores) {
18             sensorDeTemperatura.controlar();
19         }
20     }
21 }

SensorDeTemperatura.java
1 package tda;
2
3 public class SensorDeTemperatura {
4
5     private final String id;
6     private final float temperatura;
7     private final float limite;
8
9     public SensorDeTemperatura(String id, float limite, float temperatura) {
10         this.id = id;
11         this.limite = limite;
12         this.temperatura = temperatura;
13     }
14
15     public void controlar() {
16         if (temperatura > limite) {
17             System.err.println(id + " sobrecalentado!");
18         }
19     }
20 }

Console
<terminated> Main (1) [Java Application] C:\Users\Alumno\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20220903-1038\jre\bin\javaw.exe (27 sep. 2022 12:13:11 - 12:13:11) [pid: 948]
S2 sobrecalentado!
S3 sobrecalentado!
```

PoLC (Principle of Least Commitment)

Principio de mínimo compromiso

Principio de diseño orientado a objetos que propone que la interfaz de un objeto solo proporcione su comportamiento esencial, nada más.

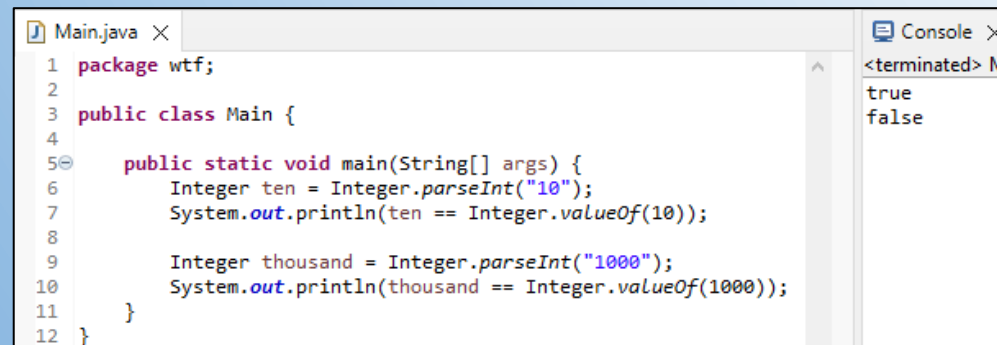


Por ejemplo: Si la interfaz `Vehiculo` declara el método `estacionar` y la clase `Auto` define además los métodos `arrancar`, `frenar`, `transferir`, `lavar`, etc., al Valet de un hotel le pasaríamos un `Vehiculo`, no un `Auto`.

PoLA (Principle of Least Astonishment)

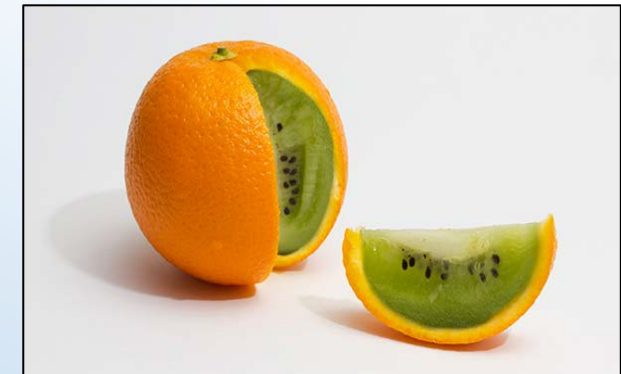
Principio del menor asombro

Principio de diseño orientado a objetos que propone que una abstracción capture todo el comportamiento de algún objeto, ni más ni menos, y no ofrezca sorpresas ni efectos secundarios que vayan más allá del alcance de la abstracción.



```
1 package wtf;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         Integer ten = Integer.parseInt("10");
7         System.out.println(ten == Integer.valueOf(10));
8
9         Integer thousand = Integer.parseInt("1000");
10        System.out.println(thousand == Integer.valueOf(1000));
11    }
12 }
```

Console X
<terminated> Ma
true
false

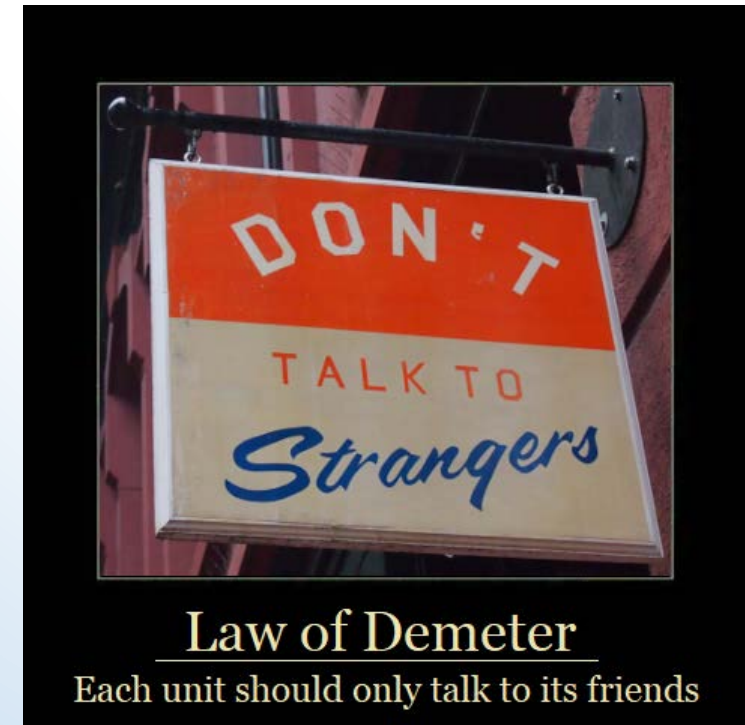


PoLK (Principle of Least Knowledge)

Principio del menor conocimiento

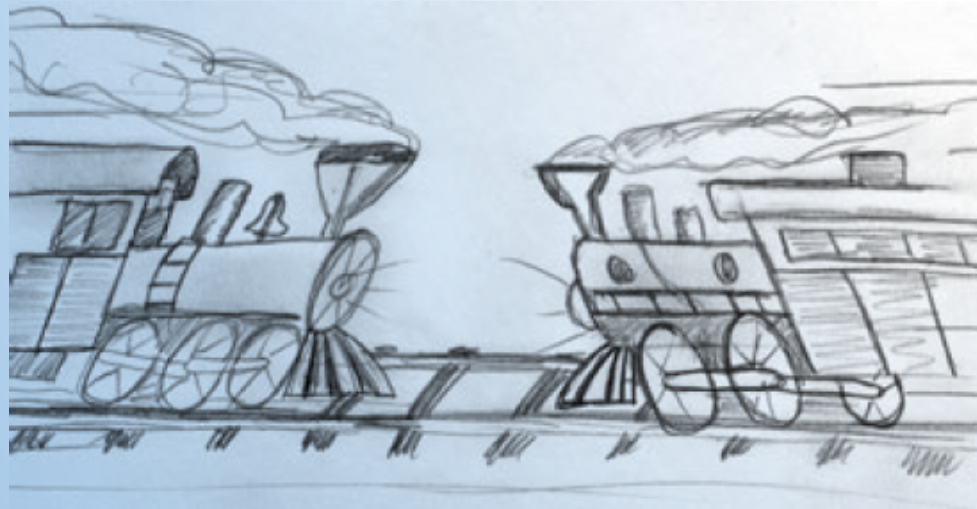
Principio de diseño orientado a objetos que propone que un método **f** de una clase **C** solo debería invocar métodos de:

- la propia clase **C**;
- los objetos que son atributos de **C**;
- los objetos recibidos por **f** como argumentos;
- los objetos instanciados en **f**.



(No se deberían invocar los métodos de un objeto retornado por otro método).

¿Cuándo se viola el principio PoLK?



```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

If `ctxt`, `Options`, and `ScratchDir` are objects, then their internal structure should be hidden rather than exposed, and so knowledge of their innards is a clear violation of the Law of Demeter.

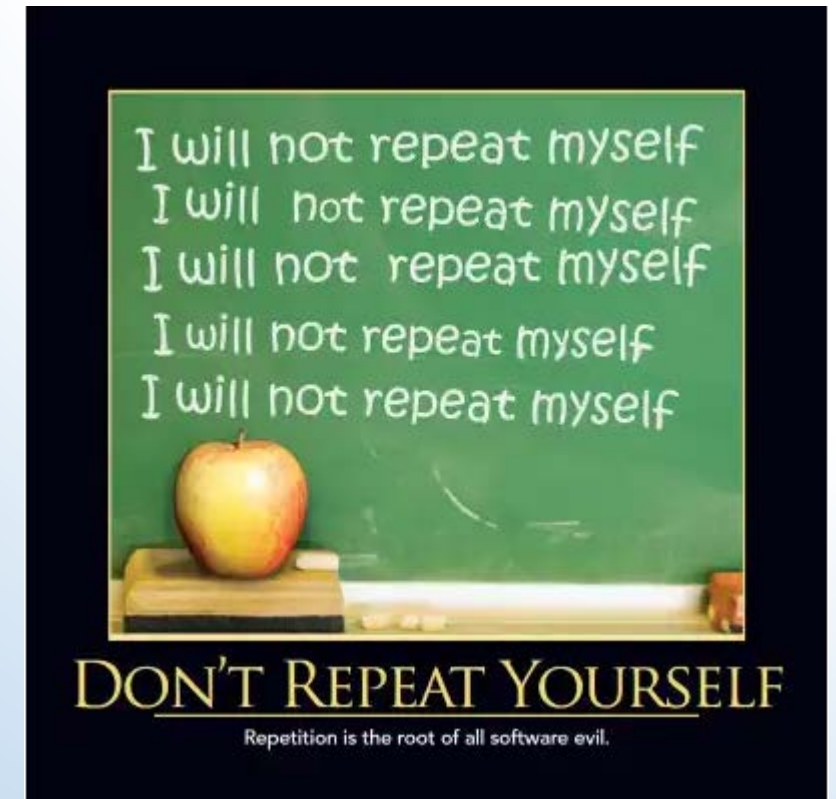
Fuente: Martin (2009). *Clean Code* (p. 98)

DRY (Don't Repeat Yourself) **(¡No te repitas!)**

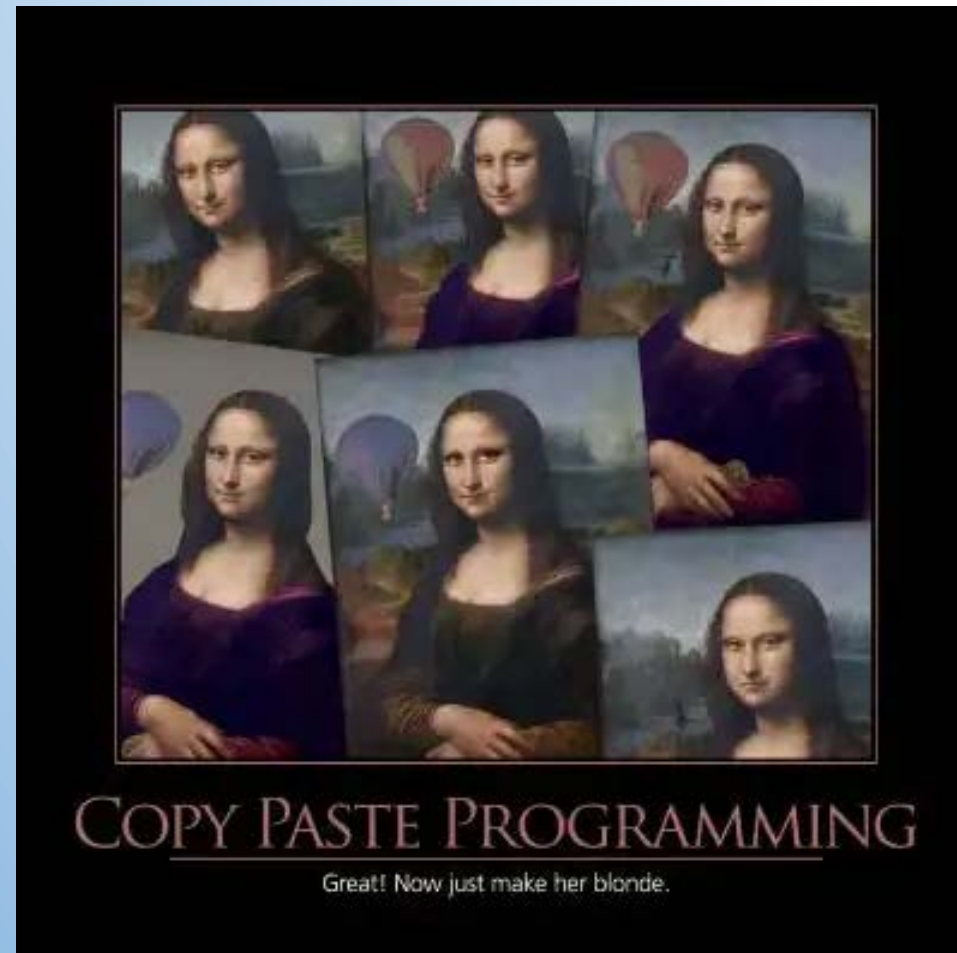
Principio de diseño que promueve la reducción de la duplicación de toda "pieza de información", abarcando:

- datos almacenados en una base de datos;
- código fuente de un programa de software;
- información textual o documentación.

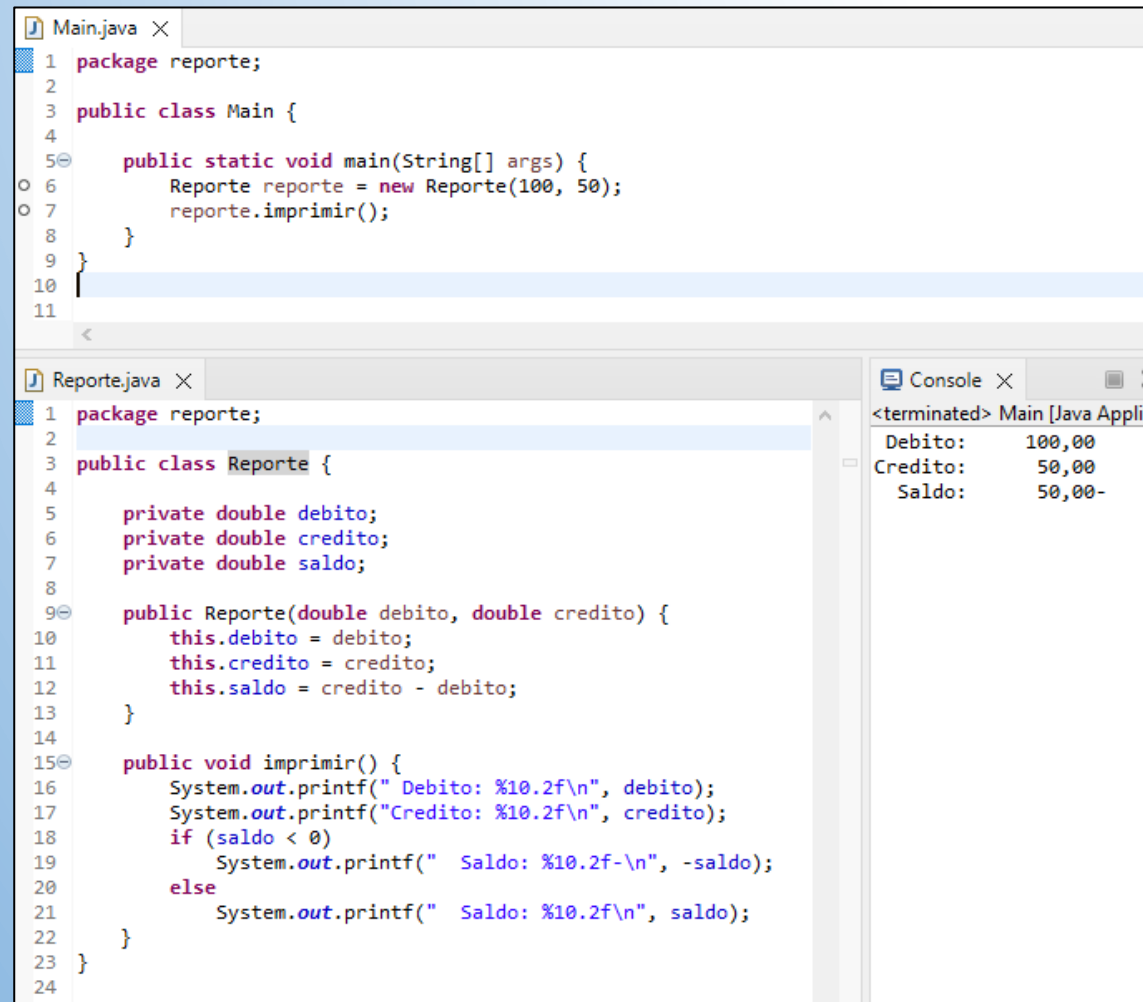
Cuando el principio DRY se aplica de forma eficiente, los cambios en cualquier parte del proceso requieren ediciones en un único lugar.



¿Cuándo se viola el principio DRY?



Ejemplo donde se viola el principio DRY:



```
1 package reporte;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         Reporte reporte = new Reporte(100, 50);
7         reporte.imprimir();
8     }
9 }
10
11
```

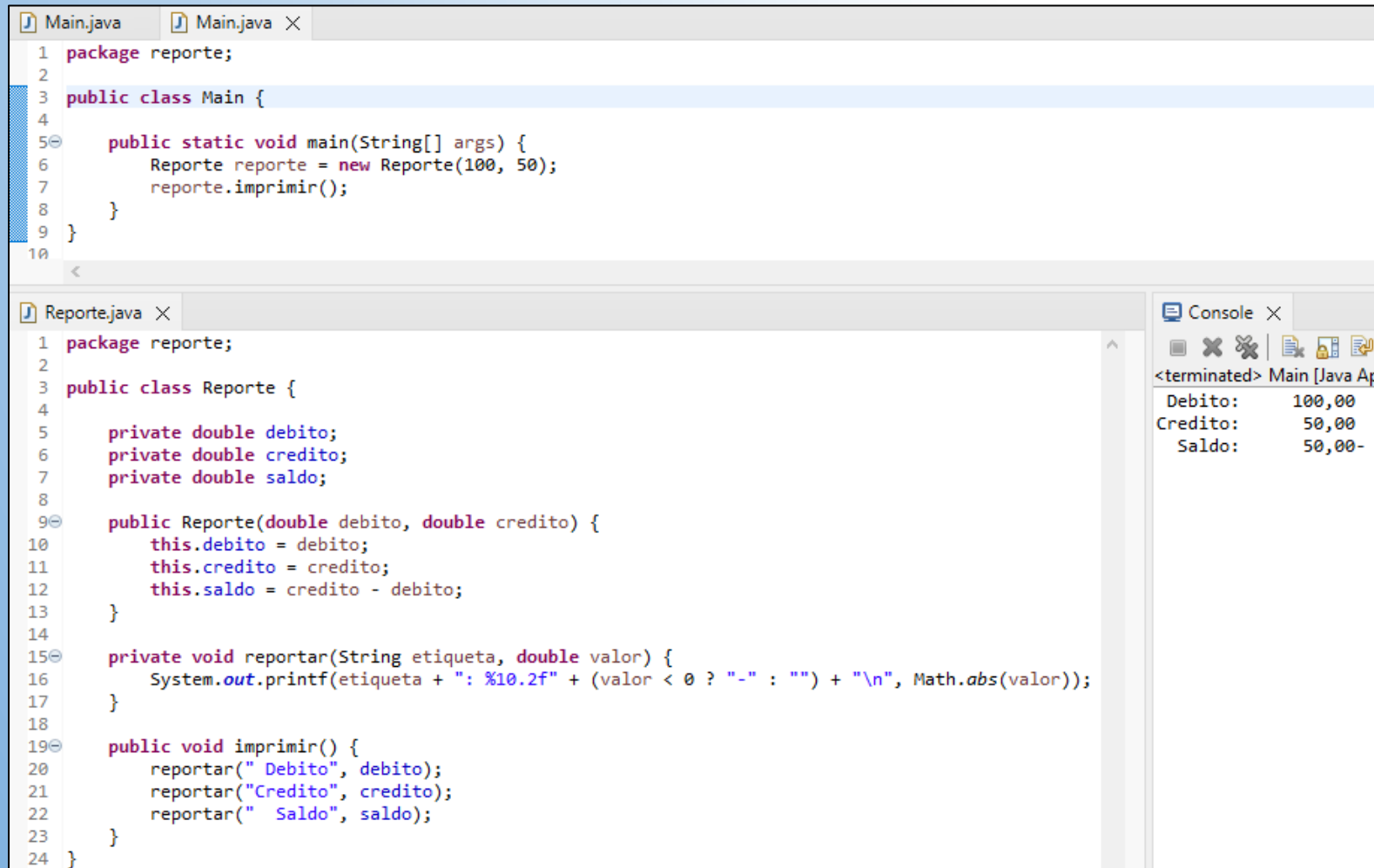
```
1 package reporte;
2
3 public class Reporte {
4
5     private double debito;
6     private double credito;
7     private double saldo;
8
9     public Reporte(double debito, double credito) {
10         this.debito = debito;
11         this.credito = credito;
12         this.saldo = credito - debito;
13     }
14
15     public void imprimir() {
16         System.out.printf(" Debito: %10.2f\n", debito);
17         System.out.printf("Credito: %10.2f\n", credito);
18         if (saldo < 0)
19             System.out.printf(" Saldo: %10.2f-\n", -saldo);
20         else
21             System.out.printf(" Saldo: %10.2f\n", saldo);
22     }
23 }
24
```

Console X

<terminated> Main [Java Applic

Debito:	100,00
Credito:	50,00
Saldo:	50,00-

Ejemplo donde se respeta el principio DRY:



```
1 package reporte;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         Reporte reporte = new Reporte(100, 50);
7         reporte.imprimir();
8     }
9 }
10
```

```
1 package reporte;
2
3 public class Reporte {
4
5     private double debito;
6     private double credito;
7     private double saldo;
8
9     public Reporte(double debito, double credito) {
10         this.debito = debito;
11         this.credito = credito;
12         this.saldo = credito - debito;
13     }
14
15     private void reportar(String etiqueta, double valor) {
16         System.out.printf(etiqueta + ": %10.2f" + (valor < 0 ? "-" : "") + "\n", Math.abs(valor));
17     }
18
19     public void imprimir() {
20         reportar(" Debito", debito);
21         reportar("Credito", credito);
22         reportar(" Saldo", saldo);
23     }
24 }
```

Console X

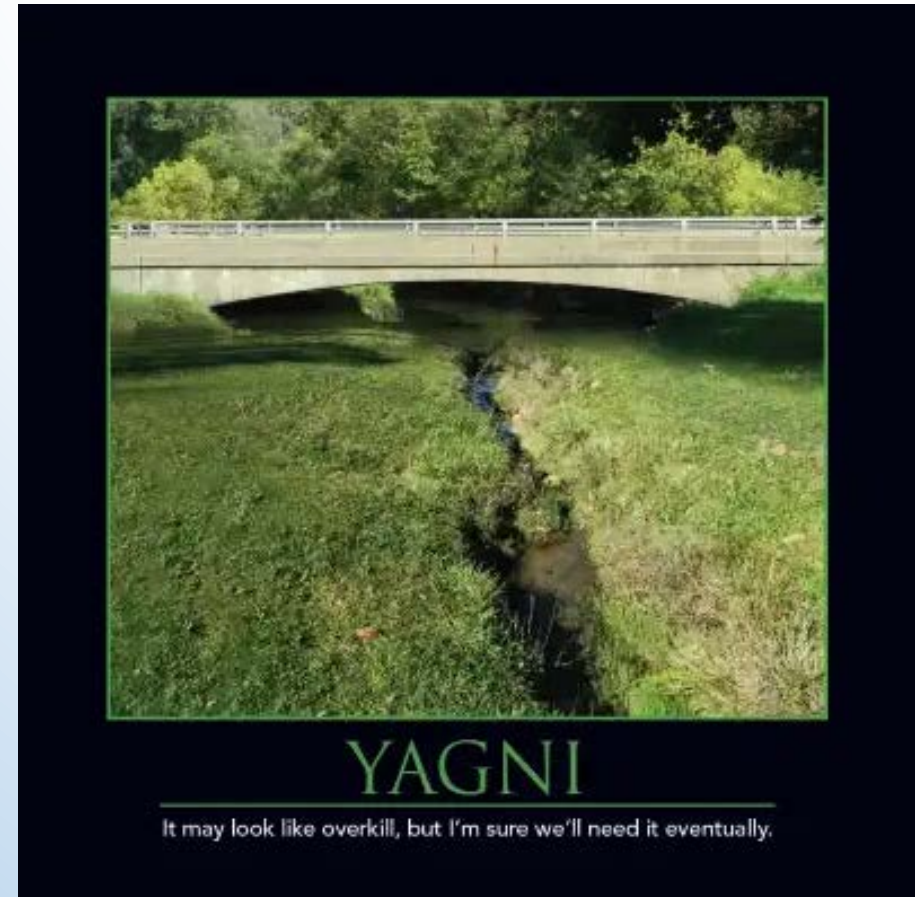
<terminated> Main [Java Ap

Debito:	100,00
Credito:	50,00
Saldo:	50,00-

YAGNI (You Ain't Gonna Need It) **(No lo vas a necesitar)**

Principio de diseño que propone no agregar nunca una funcionalidad excepto cuando sea necesaria.

Es fundamental resistir la tentación de escribir código que no es necesario, pero que podría serlo en el futuro.



¿Cuándo se viola el principio YAGNI?



YOU AREN'T GONNA NEED IT

Don't waste resources on what you *might* need.



FEATURE CREEP

Just one more feature and it's done.

KISS (Keep It Simple, Stupid!) **¡Mantenlo sencillo, estúpido!**

Principio de diseño que propone que la mayoría de sistemas funcionan mejor si se mantienen simples que si se hacen complejos.

Por ello, la simplicidad debe ser mantenida como un objetivo clave del diseño, y cualquier complejidad accidental debe ser evitada.



Keep it simple, stupid!

¿Cuándo se viola el principio KISS?



FRANKENCODE

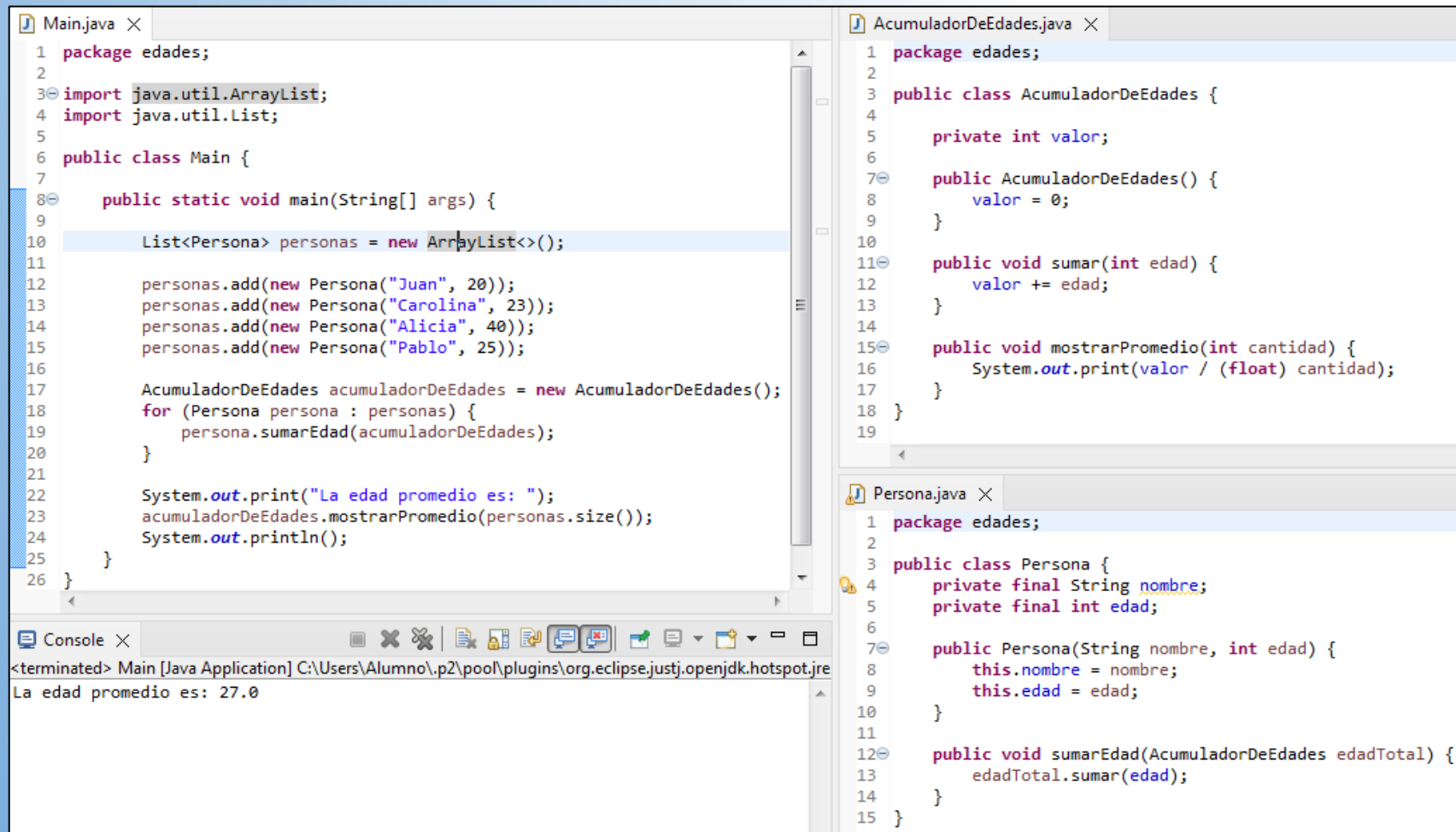
It's alive! What could go wrong?



SPAGHETTI CODE

Maintenance is easy with everything in one place.

Ejemplo donde se viola el principio KISS:



```
1 package edades;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Main {
7
8     public static void main(String[] args) {
9
10         List<Persona> personas = new ArrayList<>();
11
12         personas.add(new Persona("Juan", 20));
13         personas.add(new Persona("Carolina", 23));
14         personas.add(new Persona("Alicia", 40));
15         personas.add(new Persona("Pablo", 25));
16
17         AcumuladorDeEdades acumuladorDeEdades = new AcumuladorDeEdades();
18         for (Persona persona : personas) {
19             persona.sumarEdad(acumuladorDeEdades);
20         }
21
22         System.out.print("La edad promedio es: ");
23         acumuladorDeEdades.mostrarPromedio(personas.size());
24         System.out.println();
25     }
26 }
```

```
1 package edades;
2
3 public class AcumuladorDeEdades {
4
5     private int valor;
6
7     public AcumuladorDeEdades() {
8         valor = 0;
9     }
10
11     public void sumar(int edad) {
12         valor += edad;
13     }
14
15     public void mostrarPromedio(int cantidad) {
16         System.out.print(valor / (float) cantidad);
17     }
18 }
19
```

```
1 package edades;
2
3 public class Persona {
4     private final String nombre;
5     private final int edad;
6
7     public Persona(String nombre, int edad) {
8         this.nombre = nombre;
9         this.edad = edad;
10     }
11
12     public void sumarEdad(AcumuladorDeEdades edadTotal) {
13         edadTotal.sumar(edad);
14     }
15 }
```

Console ×

<terminated> Main [Java Application] C:\Users\Alumno\p2\poo\plugins\org.eclipse.justj.openjdk.hotspot.jre

La edad promedio es: 27.0

The image shows the Eclipse IDE with two Java files open: **Main.java** and **Persona.java**.

Main.java code:

```
1 package edades;
2
3 import java.util.ArrayList;
4
5 public class Main {
6
7     public static void main(String[] args) {
8
9         List<Persona> personas = new ArrayList<>();
10
11         personas.add(new Persona("Juan", 20));
12         personas.add(new Persona("Carolina", 23));
13         personas.add(new Persona("Alicia", 40));
14         personas.add(new Persona("Pablo", 25));
15
16         float acumuladorDeEdades = 0;
17         for (Persona persona : personas) {
18             acumuladorDeEdades += persona.getEdad();
19         }
20
21         System.out.println("La edad promedio es: " + acumuladorDeEdades / personas.size());
22     }
23 }
24
```

Persona.java code:

```
1 package edades;
2
3 public class Persona {
4     private final String nombre;
5     private final int edad;
6
7     public Persona(String nombre, int edad) {
8         this.nombre = nombre;
9         this.edad = edad;
10    }
11
12    public int getEdad() {
13        return edad;
14    }
15 }
16
17
```

The **Console** at the bottom shows the output of the program:

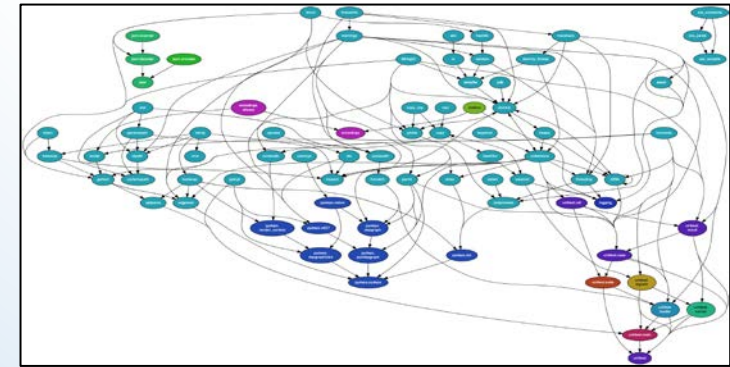
```
<terminated> Main (1) [Java Application] C:\Users\Alumno\p2\pool\plugins\org.eclipse.justi.openjdk.hotspot.jre.full.win32.x86_64_17.0.10\jre\bin\java.exe
La edad promedio es: 27.0
```


EDP (Explicit Dependencies Principle)

Principio de dependencias explícitas

Principio de diseño orientado a objetos que propone que los métodos y las clases deben requerir **explícitamente** (generalmente a través de parámetros de método o parámetros de constructor) cualquier objeto de colaboración que necesiten para funcionar correctamente.

Cuesta más mantener las clases con **dependencias implícitas** (que usan objetos que existen solo en el código dentro de sus métodos y no en su interfaz pública) que aquellas con **dependencias explícitas**.



KOP (Knuth's Optimization Principle) Principio de optimización de Knuth

Principio de diseño que propone que no se refactorice el código que aún no funciona completamente.



*The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; **premature optimization is the root of all evil** (or at least most of it) in programming.*

Fuente: Knuth, d. (1974). *Computer Programming as an Art* (p. 671)

SoC (Separation of Concerns)

Principio de separación de preocupaciones

Principio de diseño que propone separar un sistema informático en partes distintas, tal que cada parte se enfoca una funcionalidad o un interés delimitado. Por ejemplo, la "lógica de negocios" del software es una preocupación, y la interfaz a través de la cual una persona utiliza esta lógica es otra. Cambiar una no debe requerir cambiar la otra.



SEPARATION OF CONCERNS

Don't let your plumbing code pollute your software.



VERTICAL SLICES

Start with the icing - we can make the cake later.

¿Cuándo se viola el principio SoC?



ICEBERG CLASS

It's cool to hide your code.



GOLDEN HAMMER

When you have a golden hammer, everything looks like a nail.

SOLID: SRP (Single Responsibility Principle)

Principio de responsabilidad única

Principio de diseño orientado a objetos que propone que cada clase debe tener un único motivo para cambiar.

No es posible bañarse dos veces en el mismo río, porque nuevas aguas corren siempre sobre ti.

Heráclito de Éfeso (c. 540 a. C.- c. 480 a. C.)

Un gran número de métodos públicos es un indicio de que una clase hace muchas cosas. Generalmente conviene separar los métodos públicos en distintas clases.



Single Responsibility Principle

Just because you can doesn't mean you should.


Ejemplo donde se viola el principio SRP:

```
Main.java x
1 package solid;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         LlamadaHaciaFijoCABA llamada = new LlamadaHaciaFijoCABA();
7         llamada.efectuar(43112700);
8         llamada.efectuar(1530366486);
9     }
10 }

LlamadaHaciaFijoCABA.java x
1 package solid;
2
3 public class LlamadaHaciaFijoCABA {
4
5     public void efectuar(int numero) {
6         if (validar(numero)) {
7             System.out.println("Llamando al +54 11 " + numero + "...");
8         } else {
9             System.err.println("Error: " + numero + " no es un numero fijo valido...");
10        }
11    }
12
13    public boolean validar(int numero) {
14        return numero > 19999999 && numero < 70000000;
15    }
16 }

Console x
<terminated> Main (2) [Java Application] C:\Users\Alumno\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86
Error: 1530366486 no es un numero fijo valido...
Llamando al +54 11 43112700...
```

Ejemplo donde se respeta el principio SRP:



```

Main.java
1 package solid;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         LlamadaHaciaFijoCABA llamada = new LlamadaHaciaFijoCABA();
7         llamada.efectuar(43112700);
8         llamada.efectuar(1530366486);
9     }
10 }

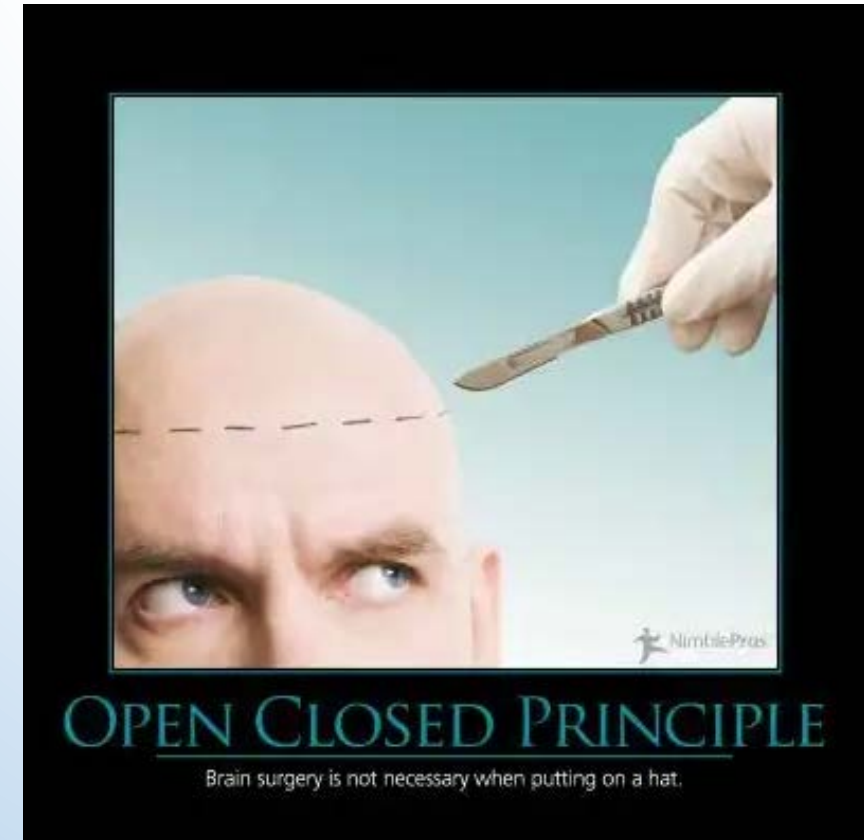
ServicioDeValidacion.java
1 package solid;
2
3 public class ServicioDeValidacion {
4
5     public boolean validar(int numero) {
6         return numero > 19999999 && numero < 70000000;
7     }
8 }

LlamadaHaciaFijoCABA.java
1 package solid;
2
3 public class LlamadaHaciaFijoCABA {
4
5     public void efectuar(int numero) {
6         ServicioDeValidacion v = new ServicioDeValidacion();
7         if (v.validar(numero)) {
8             System.out.println("Llamando al +54 11 " + numero + "...");
9         } else {
10            System.err.println("Error: " + numero + " no es un numero fijo valido...");
11        }
12    }
13 }

Console
<terminated> Main (2) [Java Application] C:\Users\Alumno\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20220903-1038\jre\bin\javaw.exe (28 sep. 2022)
Error: 1530366486 no es un numero fijo valido...
Llamando al +54 11 43112700...
```

SOLID: OCP (Open/Closed Principle) **Principio Abierto/Cerrado**

Principio de diseño orientado a objetos que propone que el comportamiento de una clase debe estar **abierto a la extensión**, pero **cerrado a la modificación** (se debe poder extender sin modificar el código existente).



Ejemplo donde se viola el principio OCP:

```

Main.java
1 package solid;
2
3 import java.util.ArrayList;
4
5 public class Main {
6
7     public static void main(String[] args) {
8         ArrayList c = new ArrayList();
9         c.add(new TrianguloEquilatero(5));
10        c.add(new Cuadrado(5));
11        c.add(new PentagonoRegular(5));
12        CalculadoraDeAreas calculo = new CalculadoraDeAreas(c);
13        calculo.calcular();
14    }
15 }
16

CalculadoraDeAreas.java
1 package solid;
2
3 import java.util.ArrayList;
4
5 public class CalculadoraDeAreas {
6
7     private ArrayList c;
8
9     public CalculadoraDeAreas(ArrayList c) {
10         this.c = c;
11     }
12
13     public void calcular() {
14         for (int i = 0; i < c.size(); i++) {
15             Object p = c.get(i);
16             if (p instanceof TrianguloEquilatero) {
17                 TrianguloEquilatero t = (TrianguloEquilatero) p;
18                 System.out.println(t.getNombre() + ": " + Math.floor(t.calcular() * 100) / 100);
19             } else if (p instanceof Cuadrado) {
20                 Cuadrado q = (Cuadrado) p;
21                 System.out.println(q.getNombre() + ": " + Math.floor(q.calcularArea() * 100) / 100);
22             } else if (p instanceof PentagonoRegular) {
23                 PentagonoRegular z = (PentagonoRegular) p;
24                 System.out.println(z.getNombre() + ": " + Math.floor(z.calcularSup() * 100) / 100);
25             }
26         }
27     }
28 }

TrianguloEquilatero.java
1 package solid;
2
3 public class TrianguloEquilatero {
4
5     private double lado;
6     private String nombre;
7
8     public TrianguloEquilatero(double lado) {
9         this.lado = lado;
10        nombre = "Triangulo equilatero";
11    }
12
13    public String getNombre() {
14        return nombre;
15    }
16
17    public double calcular() {
18        return lado * lado * Math.sqrt(3) / 4;
19    }
20 }

Cuadrado.java
1 package solid;
2
3 public class Cuadrado {
4
5     private double lado;
6     private String nombre;
7
8     public Cuadrado(double lado) {
9         this.lado = lado;
10        nombre = "Cuadrado";
11    }
12
13    public String getNombre() {
14        return nombre;
15    }
16
17    public double calcularArea() {
18        return lado * lado;
19    }
20 }

PentagonoRegular.java
1 package solid;
2
3 public class PentagonoRegular {
4
5     private double lado;
6     private String nombre;
7
8     public PentagonoRegular(double lado) {
9         this.lado = lado;
10        nombre = "Pentagono regular";
11    }
12
13    public String getNombre() {
14        return nombre;
15    }
16
17    public double calcularSup() {
18        return 5 * lado * lado / (4 * Math.tan(Math.PI / 5.0));
19    }
20 }
    
```

```

Console
<terminated> Main (4) [Java Application]
Triangulo equilatero: 10.82
Cuadrado: 25.0
Pentagono regular: 43.01
    
```

Ejemplo donde se respeta el principio OCP:

```
Main.java X
1 package solid;
2
3 import java.util.ArrayList;
4
5 public class Main {
6
7     public static void main(String[] args) {
8         ArrayList<PoligonoRegular> c = new ArrayList<>();
9         c.add(new TrianguloEquilatero(5));
10        c.add(new Cuadrado(5));
11        c.add(new PoligonoRegular(5));
12        CalculadoraDeAreas calcul = new CalculadoraDeAreas(c);
13        calcul.calcular();
14    }
15 }
16
```

```
CalculadoraDeAreas.java X
3 import java.util.ArrayList;
4
5 public class CalculadoraDeAreas {
6
7     private ArrayList<PoligonoRegular> c;
8
9     public CalculadoraDeAreas(ArrayList<PoligonoRegular> c) {
10         this.c = c;
11     }
12
13     public void calcular() {
14         for (int i = 0; i < c.size(); i++) {
15             PoligonoRegular p = c.get(i);
16             System.out.println(p.getNombre() + ": " + Math.floor(p.calcularArea() * 100) / 100);
17         }
18     }
19 }
```

```
*PoligonoRegular.java X
1 package solid;
2
3 public abstract class PoligonoRegular {
4
5     private String Nombre;
6     private double lado;
7
8     public PoligonoRegular(String Nombre, double lado) {
9         this.Nombre = Nombre;
10        this.lado = lado;
11    }
12
13     public String getNombre() {
14         return Nombre;
15     }
16
17     public double getLado() {
18         return lado;
19     }
20
21     public abstract double calcularArea();
22
23 }
24
```

```
TrianguloEquilatero.java X
1 package solid;
2
3 public class TrianguloEquilatero extends PoligonoRegular {
4
5     public TrianguloEquilatero(double lado) {
6         super("Triangulo equilatero", lado);
7     }
8
9     @Override
10    public double calcularArea() {
11        return getLado() * getLado() * Math.sqrt(3) / 4;
12    }
13 }
14
```

```
Cuadrado.java X
1 package solid;
2
3 public class Cuadrado extends PoligonoRegular {
4
5     public Cuadrado(double lado) {
6         super("Cuadrado", lado);
7     }
8
9     @Override
10    public double calcularArea() {
11        return getLado() * getLado();
12    }
13 }
14
```

```
PentagonoRegular.java X
1 package solid;
2
3 public class PentagonoRegular extends PoligonoRegular {
4
5     public PentagonoRegular(double lado) {
6         super("Pentagono regular", lado);
7     }
8
9     @Override
10    public double calcularArea() {
11        return 5*getLado()*getLado()/(4*Math.tan(Math.PI/5.0));
12    }
13 }
14
```

```
Console X
<terminated> Main (5) [Java Application]
Triangulo equilatero: 10.82
Cuadrado: 25.0
Pentagono regular: 43.01
```

SOLID: LSP (Liskov Substitution Principle)

Principio de sustitución de Liskov

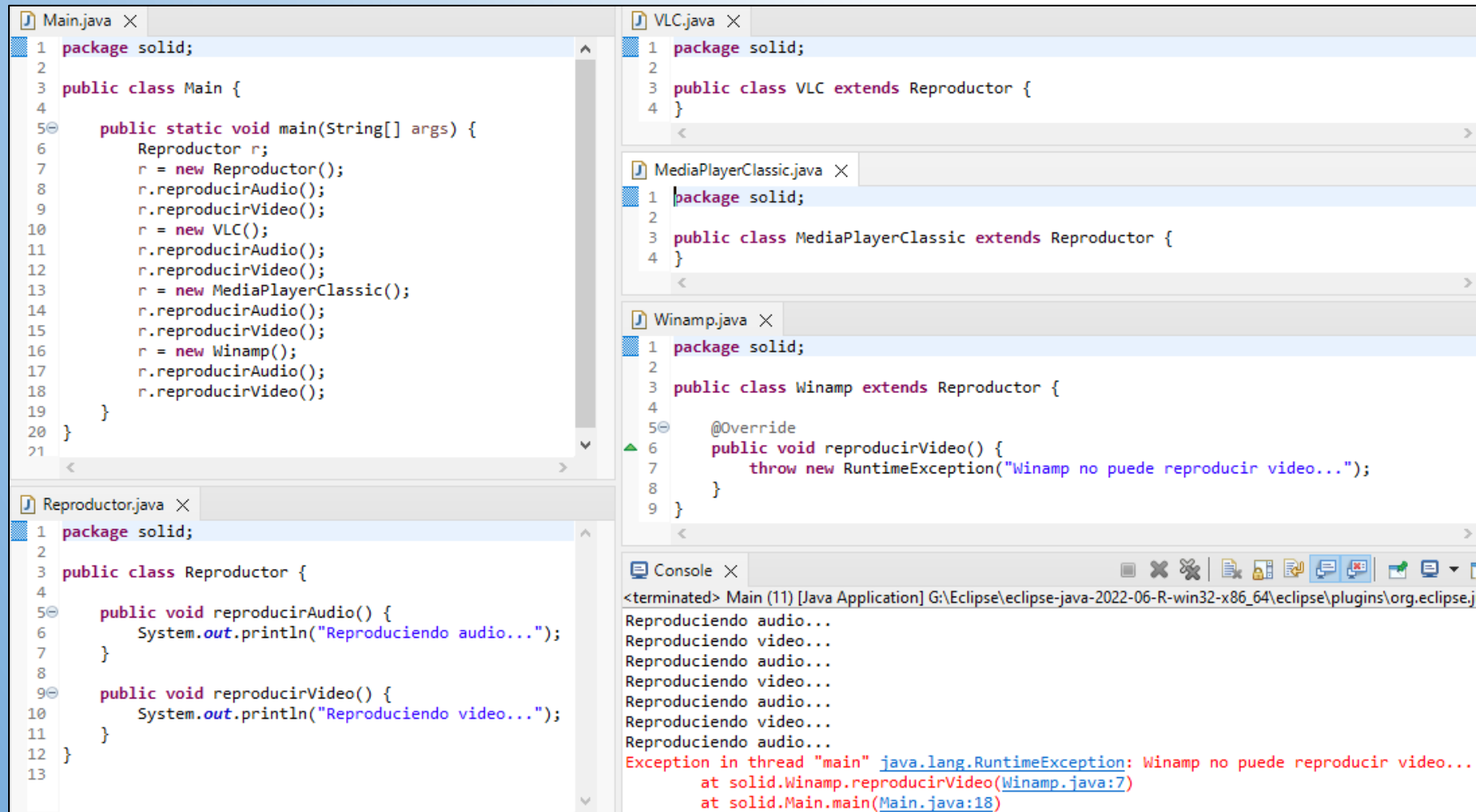
Principio de diseño orientado a objetos que propone que los objetos de las subclases deben poder sustituir a las instancias de las superclases sin alterar el correcto funcionamiento del programa.



Barbara Liskov



Ejemplo donde se viola el principio LSP:



```
1 package solid;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         Reproductor r;
7         r = new Reproductor();
8         r.reproducirAudio();
9         r.reproducirVideo();
10        r = new VLC();
11        r.reproducirAudio();
12        r.reproducirVideo();
13        r = new MediaPlayerClassic();
14        r.reproducirAudio();
15        r.reproducirVideo();
16        r = new Winamp();
17        r.reproducirAudio();
18        r.reproducirVideo();
19    }
20 }
21
```

```
1 package solid;
2
3 public class Reproductor {
4
5     public void reproducirAudio() {
6         System.out.println("Reproduciendo audio...");
7     }
8
9     public void reproducirVideo() {
10        System.out.println("Reproduciendo video...");
11    }
12 }
13
```

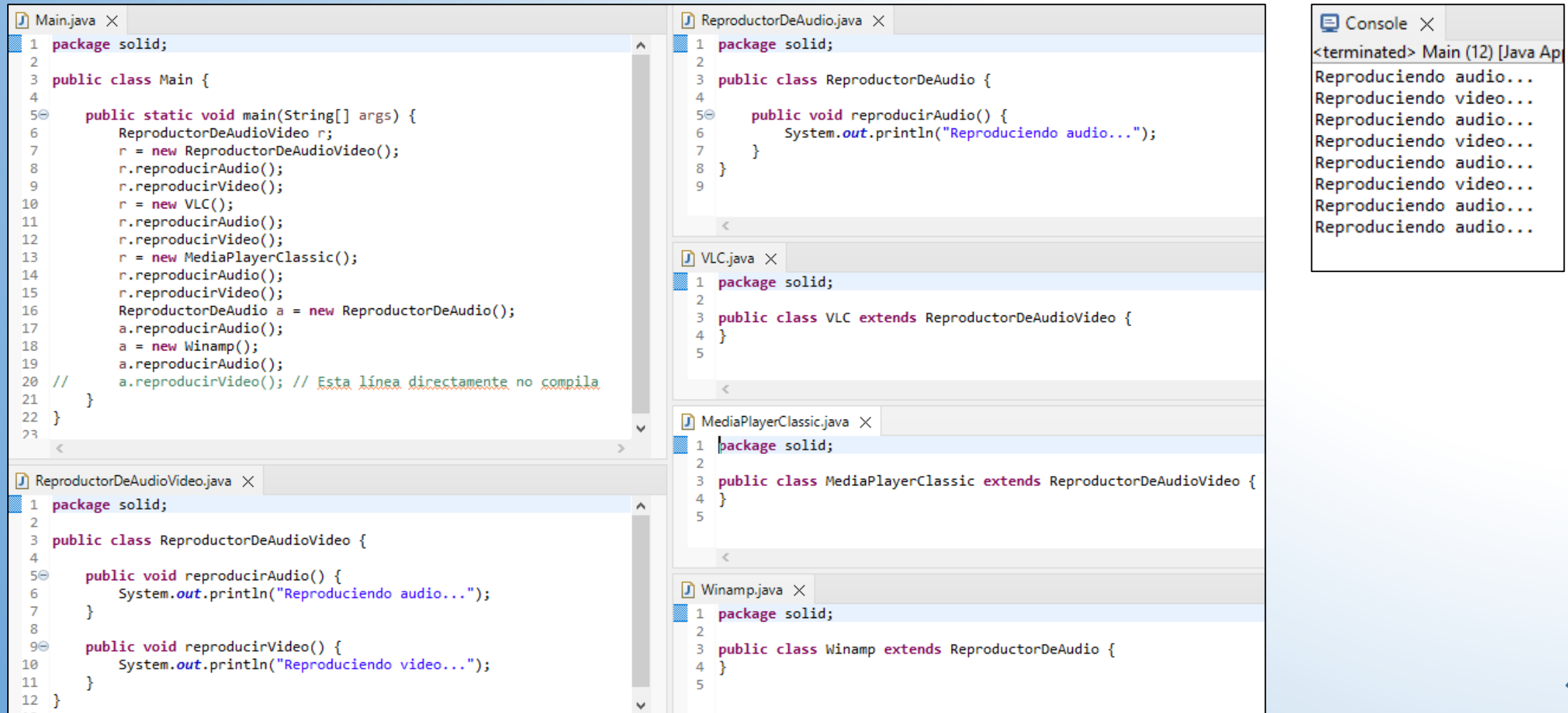
```
1 package solid;
2
3 public class VLC extends Reproductor {
4 }
5
```

```
1 package solid;
2
3 public class MediaPlayerClassic extends Reproductor {
4 }
5
```

```
1 package solid;
2
3 public class Winamp extends Reproductor {
4
5     @Override
6     public void reproducirVideo() {
7         throw new RuntimeException("Winamp no puede reproducir video...");
8     }
9 }
10
```

```
<terminated> Main (11) [Java Application] G:\Eclipse\eclipse-java-2022-06-R-win32-x86_64\eclipse\plugins\org.eclipse.jdt.core\bin\org.eclipse.jdt.core.jar
Reproduciendo audio...
Reproduciendo video...
Reproduciendo audio...
Reproduciendo video...
Reproduciendo audio...
Reproduciendo video...
Reproduciendo audio...
Exception in thread "main" java.lang.RuntimeException: Winamp no puede reproducir video...
    at solid.Winamp.reproducirVideo(Winamp.java:7)
    at solid.Main.main(Main.java:18)
```


Ejemplo donde se respeta el principio LSP:



The screenshot displays an IDE with several Java files open, illustrating the Liskov Substitution Principle (LSP) in a media player application. The files and their contents are as follows:

- Main.java**:

```
1 package solid;  
2  
3 public class Main {  
4  
5     public static void main(String[] args) {  
6         ReproductorDeAudioVideo r;  
7         r = new ReproductorDeAudioVideo();  
8         r.reproducirAudio();  
9         r.reproducirVideo();  
10        r = new VLC();  
11        r.reproducirAudio();  
12        r.reproducirVideo();  
13        r = new MediaPlayerClassic();  
14        r.reproducirAudio();  
15        r.reproducirVideo();  
16        ReproductorDeAudio a = new ReproductorDeAudio();  
17        a.reproducirAudio();  
18        a = new Winamp();  
19        a.reproducirAudio();  
20        // a.reproducirVideo(); // Esta línea directamente no compila  
21    }  
22 }  
23
```
- ReproductorDeAudioVideo.java**:

```
1 package solid;  
2  
3 public class ReproductorDeAudioVideo {  
4  
5     public void reproducirAudio() {  
6         System.out.println("Reproduciendo audio...");  
7     }  
8  
9     public void reproducirVideo() {  
10        System.out.println("Reproduciendo video...");  
11    }  
12 }
```
- ReproductorDeAudio.java**:

```
1 package solid;  
2  
3 public class ReproductorDeAudio {  
4  
5     public void reproducirAudio() {  
6         System.out.println("Reproduciendo audio...");  
7     }  
8 }  
9
```
- VLC.java**:

```
1 package solid;  
2  
3 public class VLC extends ReproductorDeAudioVideo {  
4 }  
5
```
- MediaPlayerClassic.java**:

```
1 package solid;  
2  
3 public class MediaPlayerClassic extends ReproductorDeAudioVideo {  
4 }  
5
```
- Winamp.java**:

```
1 package solid;  
2  
3 public class Winamp extends ReproductorDeAudio {  
4 }  
5
```

The **Console** window on the right shows the output of the program, demonstrating that the subclasses (VLC and MediaPlayerClassic) can be used interchangeably with the base class (ReproductorDeAudioVideo) without any issues, thus respecting the LSP.

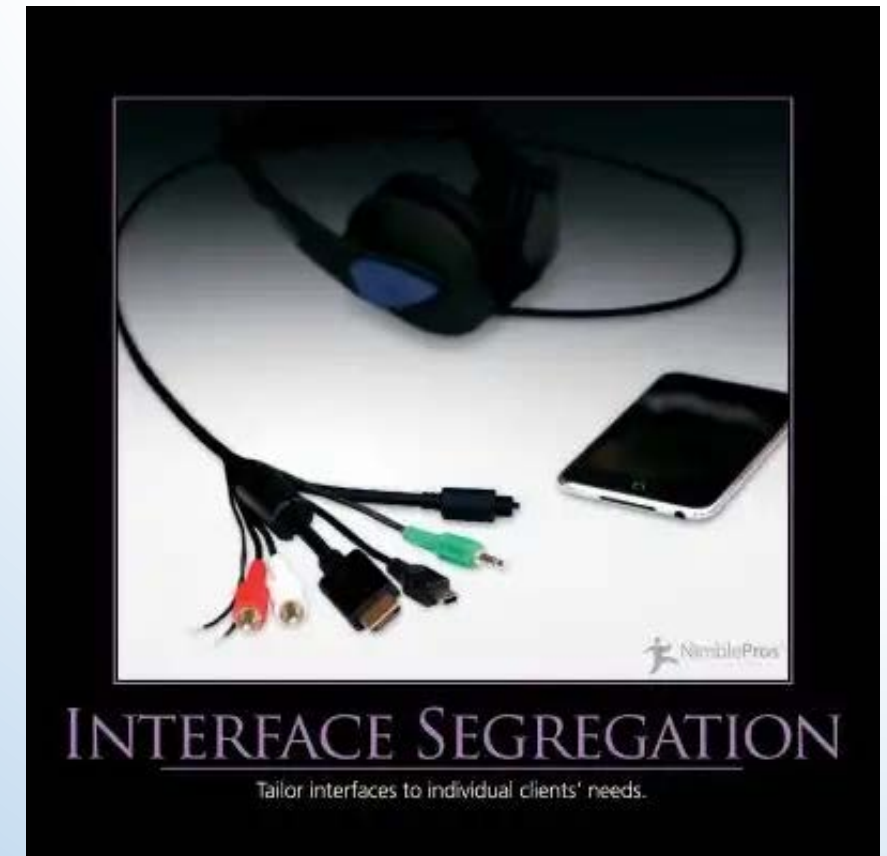
```
<terminated> Main (12) [Java Ap  
Reproduciendo audio...  
Reproduciendo video...  
Reproduciendo audio...  
Reproduciendo video...  
Reproduciendo audio...  
Reproduciendo video...  
Reproduciendo audio...  
Reproduciendo audio...
```

SOLID: ISP (Interface Segregation Principle)

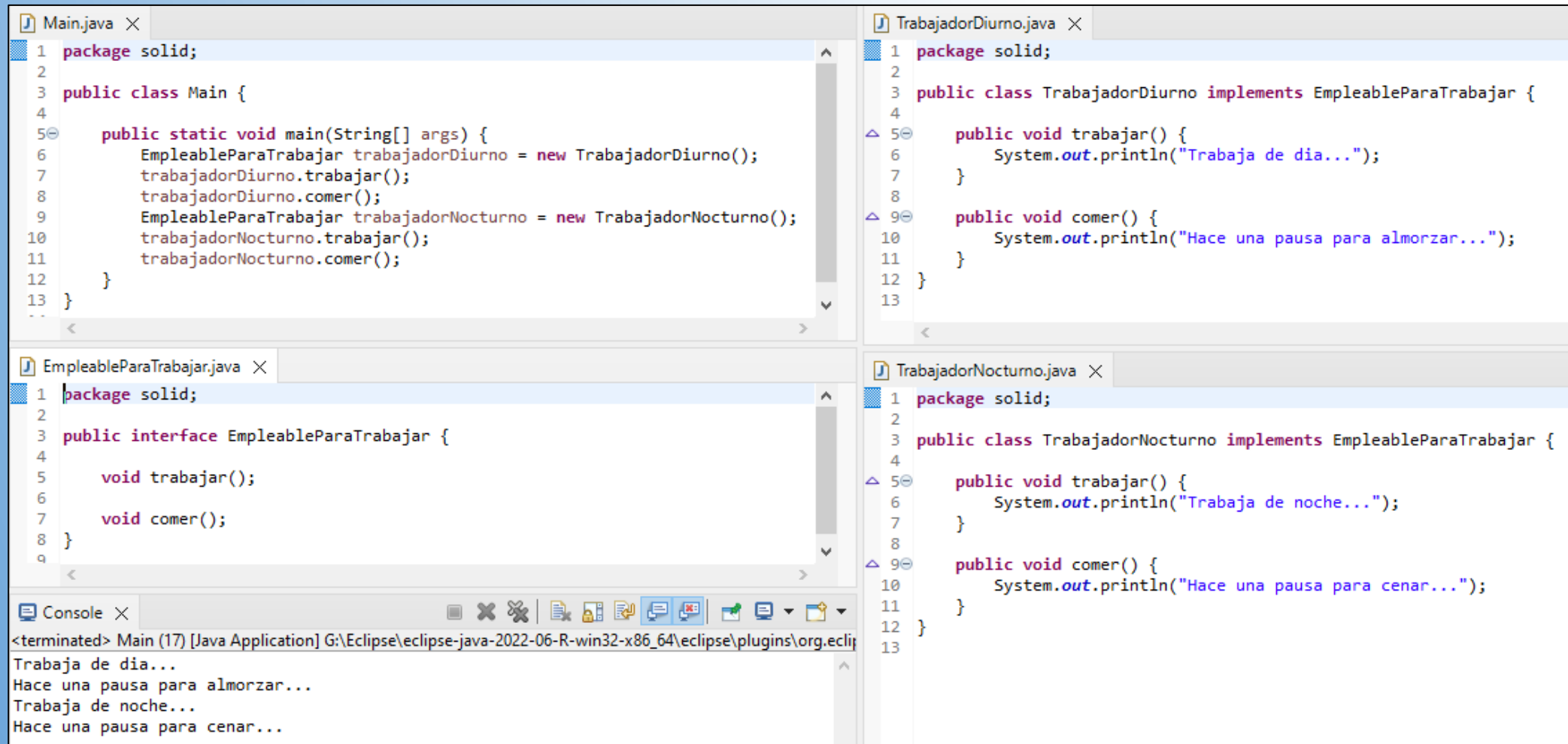
Principio de segregación de la interfaz

Principio de diseño orientado a objetos que propone que deben crearse pequeñas interfaces específicas para los clientes en lugar de una general.

De este modo, quienes implementen una interfaz no se verán forzados a implementar partes (métodos o propiedades) que no les sean útiles.



Ejemplo donde se viola el principio ISP:



```
1 package solid;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         EmpleableParaTrabajar trabajadorDiurno = new TrabajadorDiurno();
7         trabajadorDiurno.trabajar();
8         trabajadorDiurno.comer();
9         EmpleableParaTrabajar trabajadorNocturno = new TrabajadorNocturno();
10        trabajadorNocturno.trabajar();
11        trabajadorNocturno.comer();
12    }
13 }
```

```
1 package solid;
2
3 public class TrabajadorDiurno implements EmpleableParaTrabajar {
4
5     public void trabajar() {
6         System.out.println("Trabaja de dia...");
7     }
8
9     public void comer() {
10        System.out.println("Hace una pausa para almorzar...");
11    }
12 }
13
```

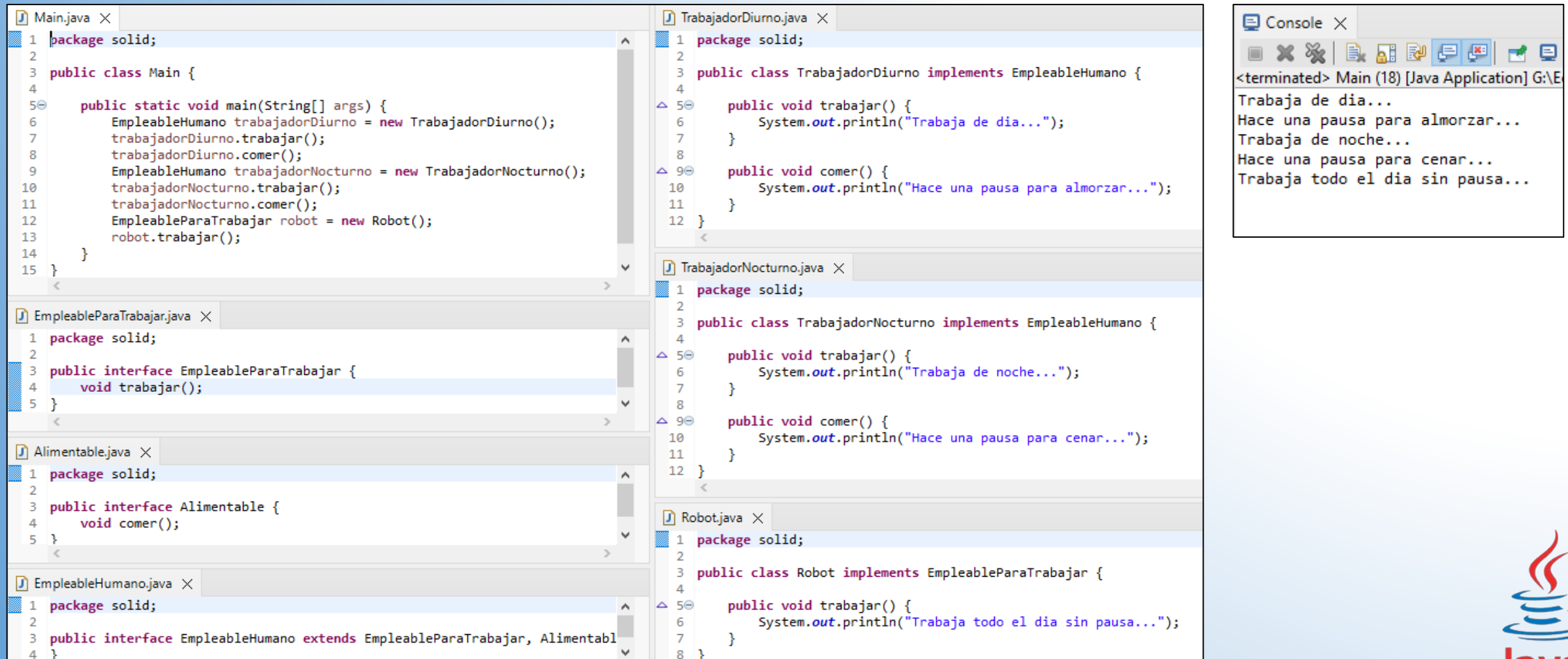
```
1 package solid;
2
3 public interface EmpleableParaTrabajar {
4
5     void trabajar();
6
7     void comer();
8 }
9
```

```
1 package solid;
2
3 public class TrabajadorNocturno implements EmpleableParaTrabajar {
4
5     public void trabajar() {
6         System.out.println("Trabaja de noche...");
7     }
8
9     public void comer() {
10        System.out.println("Hace una pausa para cenar...");
11    }
12 }
13
```

Console X

```
<terminated> Main (17) [Java Application] G:\Eclipse\eclipse-java-2022-06-R-win32-x86_64\eclipse\plugins\org.eclips
Trabaja de dia...
Hace una pausa para almorzar...
Trabaja de noche...
Hace una pausa para cenar...
```

Ejemplo donde se respeta el principio ISP:



```
1 package solid;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         EmpleadoHumano trabajadorDiurno = new TrabajadorDiurno();
7         trabajadorDiurno.trabajar();
8         trabajadorDiurno.comer();
9         EmpleadoHumano trabajadorNocturno = new TrabajadorNocturno();
10        trabajadorNocturno.trabajar();
11        trabajadorNocturno.comer();
12        EmpleadoParaTrabajar robot = new Robot();
13        robot.trabajar();
14    }
15 }
```

```
1 package solid;
2
3 public interface EmpleadoParaTrabajar {
4     void trabajar();
5 }
```

```
1 package solid;
2
3 public interface Alimentable {
4     void comer();
5 }
```

```
1 package solid;
2
3 public interface EmpleadoHumano extends EmpleadoParaTrabajar, Alimentable {
4 }
```

```
1 package solid;
2
3 public class TrabajadorDiurno implements EmpleadoHumano {
4
5     public void trabajar() {
6         System.out.println("Trabaja de dia...");
7     }
8
9     public void comer() {
10        System.out.println("Hace una pausa para almorzar...");
11    }
12 }
```

```
1 package solid;
2
3 public class TrabajadorNocturno implements EmpleadoHumano {
4
5     public void trabajar() {
6         System.out.println("Trabaja de noche...");
7     }
8
9     public void comer() {
10        System.out.println("Hace una pausa para cenar...");
11    }
12 }
```

```
1 package solid;
2
3 public class Robot implements EmpleadoParaTrabajar {
4
5     public void trabajar() {
6         System.out.println("Trabaja todo el dia sin pausa...");
7     }
8 }
```

Console X

<terminated> Main (18) [Java Application] G:\E

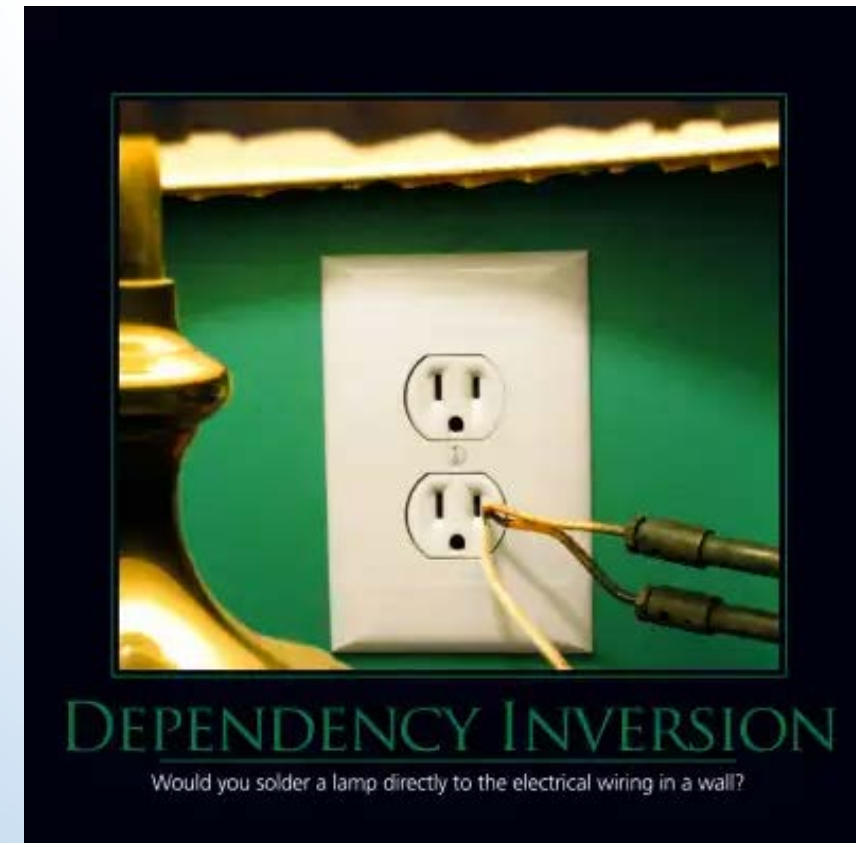
Trabaja de dia...
Hace una pausa para almorzar...
Trabaja de noche...
Hace una pausa para cenar...
Trabaja todo el dia sin pausa...

SOLID: DIP (Dependency Inversion Principle)

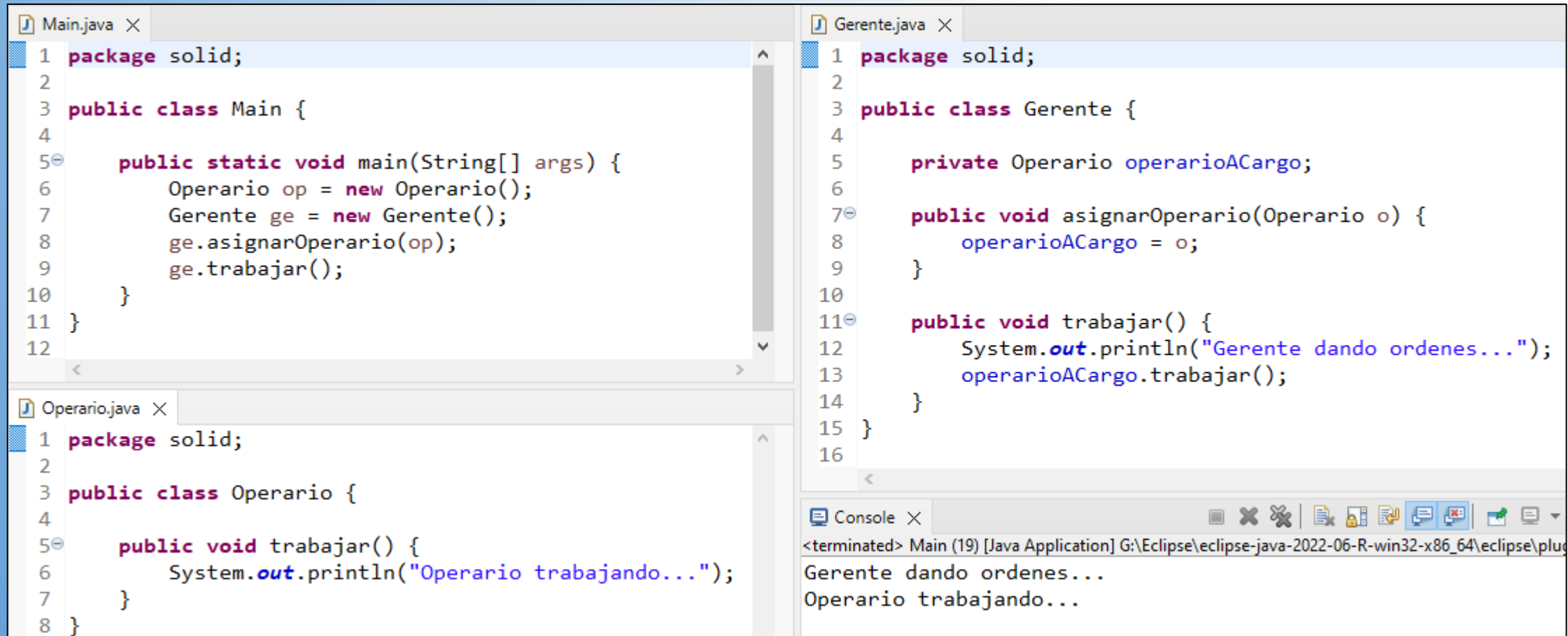
Principio de inversión de la dependencia

Principio de diseño orientado a objetos que propone que:

- Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
- Las abstracciones no deberían depender de detalles (implementaciones concretas). Las detalles deberían depender de abstracciones.



Ejemplo donde se viola el principio DIP:



```

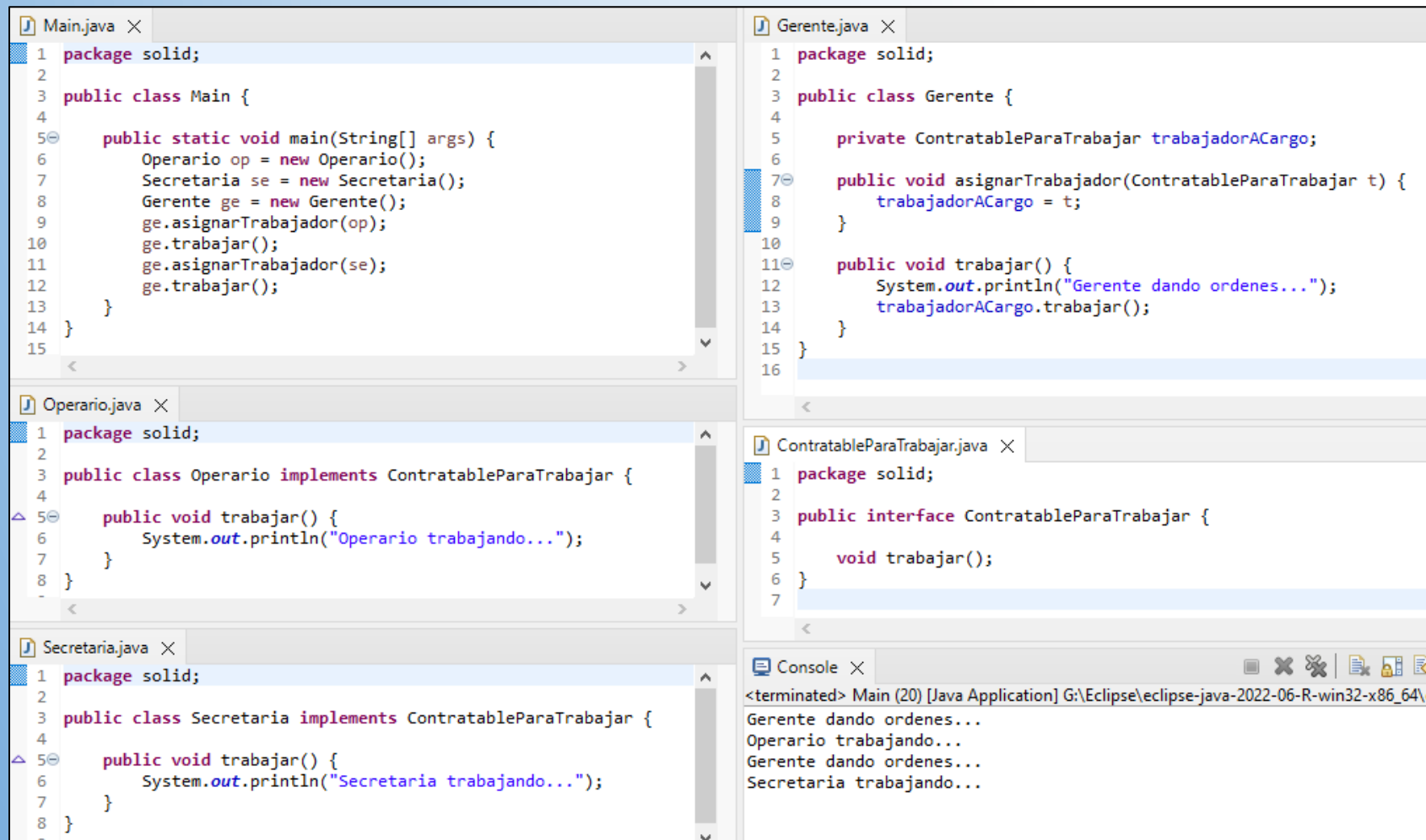
Main.java
1 package solid;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         Operario op = new Operario();
7         Gerente ge = new Gerente();
8         ge.asignarOperario(op);
9         ge.trabajar();
10    }
11 }
12

Operario.java
1 package solid;
2
3 public class Operario {
4
5     public void trabajar() {
6         System.out.println("Operario trabajando...");
7     }
8 }

Gerente.java
1 package solid;
2
3 public class Gerente {
4
5     private Operario operarioACargo;
6
7     public void asignarOperario(Operario o) {
8         operarioACargo = o;
9     }
10
11    public void trabajar() {
12        System.out.println("Gerente dando ordenes...");
13        operarioACargo.trabajar();
14    }
15 }
16

Console
<terminated> Main (19) [Java Application] G:\Eclipse\eclipse-java-2022-06-R-win32-x86_64\eclipse\plug
Gerente dando ordenes...
Operario trabajando...
```

Ejemplo donde se respeta el principio DIP:



```
1 package solid;
2
3 public class Main {
4
5     public static void main(String[] args) {
6         Operario op = new Operario();
7         Secretaria se = new Secretaria();
8         Gerente ge = new Gerente();
9         ge.asignarTrabajador(op);
10        ge.trabajar();
11        ge.asignarTrabajador(se);
12        ge.trabajar();
13    }
14 }
15
```

```
1 package solid;
2
3 public class Gerente {
4
5     private ContratableParaTrabajar trabajadorACargo;
6
7     public void asignarTrabajador(ContratableParaTrabajar t) {
8         trabajadorACargo = t;
9     }
10
11    public void trabajar() {
12        System.out.println("Gerente dando ordenes...");
13        trabajadorACargo.trabajar();
14    }
15 }
16
```

```
1 package solid;
2
3 public class Operario implements ContratableParaTrabajar {
4
5     public void trabajar() {
6         System.out.println("Operario trabajando...");
7     }
8 }
9
```

```
1 package solid;
2
3 public class Secretaria implements ContratableParaTrabajar {
4
5     public void trabajar() {
6         System.out.println("Secretaria trabajando...");
7     }
8 }
9
```

```
1 package solid;
2
3 public interface ContratableParaTrabajar {
4
5     void trabajar();
6 }
7
```

Console X

```
<terminated> Main (20) [Java Application] G:\Eclipse\eclipse-java-2022-06-R-win32-x86_64\
Gerente dando ordenes...
Operario trabajando...
Gerente dando ordenes...
Secretaria trabajando...
```

***FIRST* (Fast, Independent, Repeatable, Self-checking, Timely)**
Rápido, independiente, repetible, autovalidado, oportuno

Principio de diseño para pruebas que es el acrónimo de las cinco características que deben tener nuestros *tests* unitarios para ser considerados **pruebas con calidad**.

