

# SOLID VS CUPID

---

- Efrén García Valencia UO277189
- José Rodríguez Rico UO276922
- Daniel Machado Sánchez UO276257



# PRINCIPIOS SOLID



# ORÍGENES

---

- Introducidos por Robert C. Martin
- Fueron reforzados por Michael Feathers, que introdujo el acrónimo SOLID
- Objetivo: código mantenible, entendible y flexible

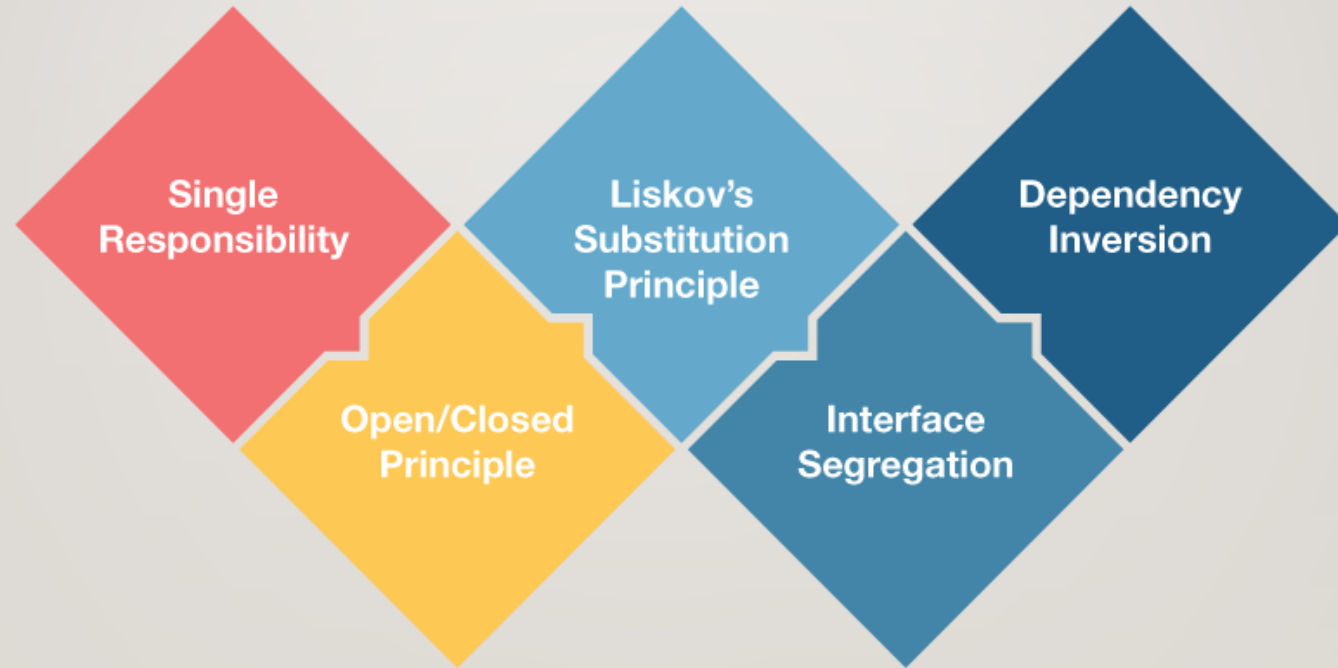


*Robert C. Martin*



*Michael Feathers*

# S.O.L.I.D.



# SINGLE RESPONSIBILITY

---

- Nos dice que una clase sólo debe tener una única responsabilidad, es decir, un único motivo para el cambio.

```
public class Book {  
  
    private String name;  
    private String author;  
    private String text;  
  
    //constructor, getters and setters  
  
    // methods that directly relate to the book properties  
    public String replaceWordInText(String word){  
        return text.replaceAll(word, text);  
    }  
  
    public boolean isWordInText(String word){  
        return text.contains(word);  
    }  
}
```

```
public class Book {  
    //...  
  
    void printTextToConsole(){  
        // our code for formatting and printing the text  
    }  
}
```

```
public class BookPrinter {  
  
    // methods for outputting text  
    void printTextToConsole(String text){  
        //our code for formatting and printing the text  
    }  
  
    void printTextToAnotherMedium(String text){  
        // code for writing to any other location..  
    }  
}
```

# OPEN/CLOSED

---

- Las clases deberían estar abiertas para la extensión, pero cerradas para la modificación.

```
public class Guitar {  
  
    private String make;  
    private String model;  
    private int volume;  
  
    //Constructors, getters & setters  
}
```

```
public class SuperCoolGuitarWithFlames extends Guitar {  
  
    private String flameColor;  
  
    //constructor, getters + setters  
}
```

# LISKOV SUBSTITUTION

---

- Los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento de un programa.

```
public interface Car {  
  
    void turnOnEngine();  
    void accelerate();  
}
```

```
public class ElectricCar implements Car {  
  
    public void turnOnEngine() {  
        throw new AssertionError("I don't have an engine!");  
    }  
  
    public void accelerate() {  
        //this acceleration is crazy!  
    }  
}
```

```
public class MotorCar implements Car {  
  
    private Engine engine;  
  
    //Constructors, getters + setters  
  
    public void turnOnEngine() {  
        //turn on the engine!  
        engine.on();  
    }  
  
    public void accelerate() {  
        //move forward!  
        engine.powerOn(1000);  
    }  
}
```

# INTERFACE SEGREGATION

---

- Es mejor tener muchas interfaces específicas para cada cliente que una sola interfaz de propósito general.

```
public interface BearKeeper {  
    void washTheBear();  
    void feedTheBear();  
    void petTheBear();  
}
```

```
public interface BearCleaner {  
    void washTheBear();  
}  
  
public interface BearFeeder {  
    void feedTheBear();  
}  
  
public interface BearPetter {  
    void petTheBear();  
}
```



# DEPENDENCY INVERSION

---

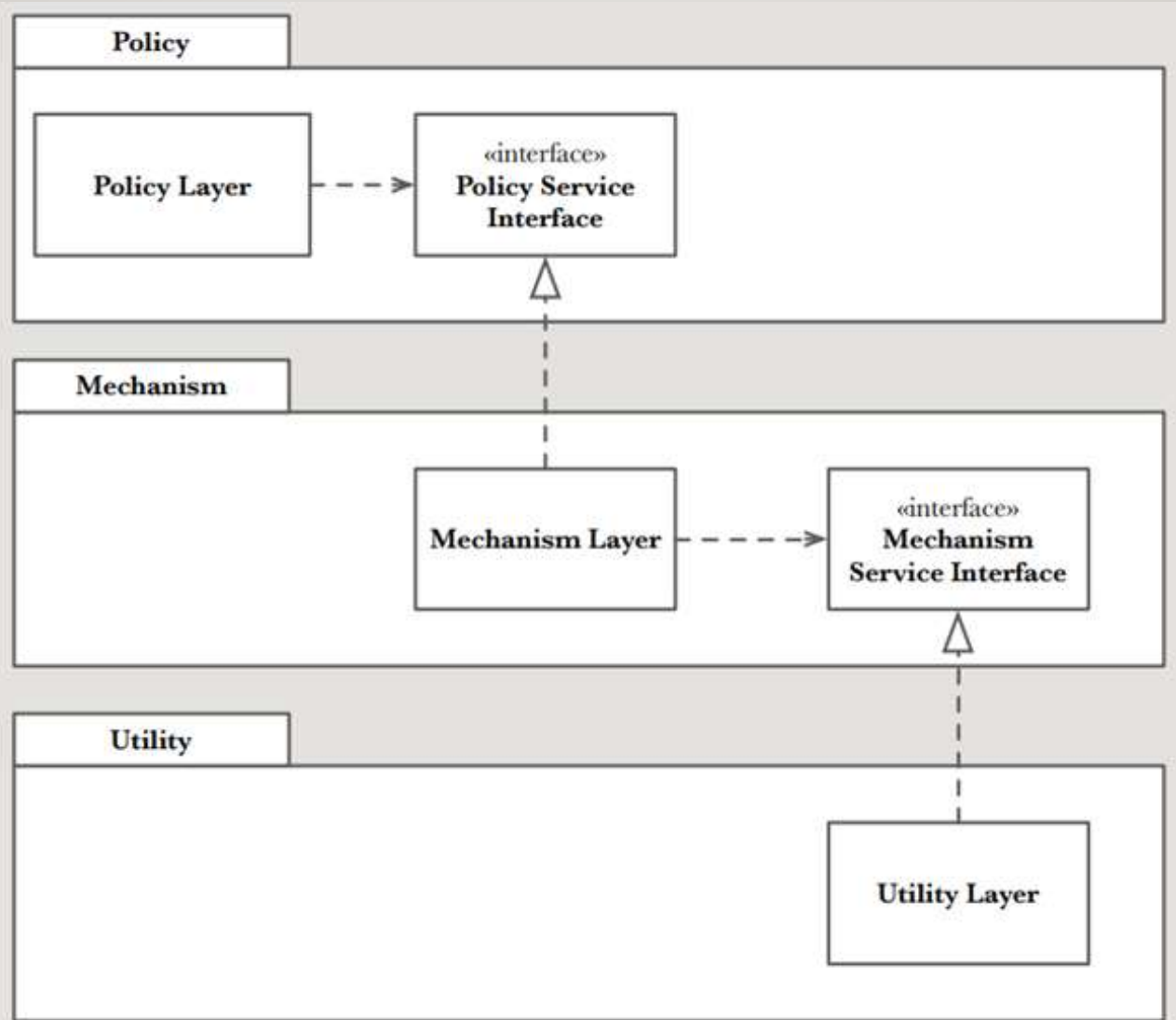
- Los módulos de alto nivel no deben depender de los de bajo nivel; ambos deben depender de abstracciones.
- Las abstracciones no deben depender de los detalles, sino éstos de las abstracciones.


```
public class Windows98Machine {  
  
    private final StandardKeyboard keyboard;  
    private final Monitor monitor;  
  
    public Windows98Machine() {  
        monitor = new Monitor();  
        keyboard = new StandardKeyboard();  
    }  
  
}
```

```
public class Windows98Machine{  
  
    private final Keyboard keyboard;  
    private final Monitor monitor;  
  
    public Windows98Machine(Keyboard keyboard, Monitor monitor) {  
        this.keyboard = keyboard;  
        this.monitor = monitor;  
    }  
  
}
```

```
public interface Keyboard { }
```

```
public class StandardKeyboard implements Keyboard { }
```





# PROBLEMAS PRINCIPIOS SOLID



# PRINCIPIOS VS PROPIEDADES

---

Información obtenida de Daniel Terhorst-North.

- ❖ Los principios se pueden cumplir o no cumplir.
- ❖ Las propiedades son cualidades del código a seguir.



# SR: POINTLESSLY VAGUE

---

¿Cómo es posible predecir lo que va a cambiar?

- Idea ambigua y confusa
- Cambiar únicamente por algo que "tengas en la cabeza"

# OP: CRUFT ACCRETION

---

La idea de este principio se ha quedado anticuada.

- Antiguamente el código era como ladrillos de edificio
- Hoy en día es maleable como la arcilla

# LSP: SIMPLE AND SMALL

---

¿De cuántas formas se pueden combinar los subtipos y las subclases?

Lo más importante es construir tipos simples y fáciles de razonar



# IS: STABLE DOOR

---

¿Estoy aplicando un principio o un patrón de diseño?


- ❖ Un principio es un buen consejo en cualquier contexto
- ❖ Un patrón es una estrategia para un contexto dado.

# DI:WRONG GOAL

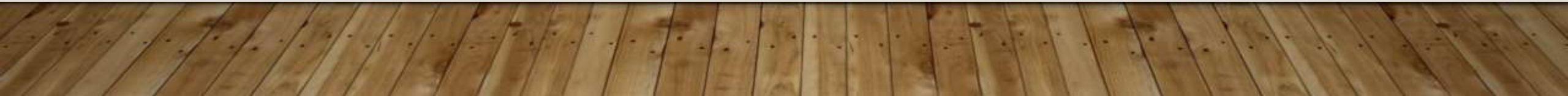
---

Intentar cumplir este principio a causado grandes pérdidas de dinero

- Centrarse en el uso en lugar de en la reutilización



# PROPIEDADES CUPID



# ENFOQUE

---

Principios SOLID no están equivocados pero son demasiado vagos

“Si pudiera dar principios para el desarrollo de software moderno en estos días, ¿cuáles elegiría?”

Colección de propiedades centradas en la persona que desarrolla el código.

# CUPID: COMPONIBLE

---

- Código fácil de usar se puede reutilizar
- Cuanto menos hay que aprender, menos puede ir mal, menos conflictos
- Fácil de descubrir, fácil de evaluar
- Dependencias mínimas



# CUPID: FILOSOFIA UNIX

---

- Un buen código garantiza que un programa/clase/función solo realice una tarea
- Qué hace el código y no cómo cambia

UNIX

# CUPID: PREDECIBLE

---

- Sin sorpresas
- Determinista → misma cosa siempre con características bien entendidas
- Observable → estado interno puede derivar de las salidas

# CUPID: IDIOMÁTICO

---

- Tratar con buen código debe sentirse natural
- Modismos del contexto
- Ecosistema del lenguaje





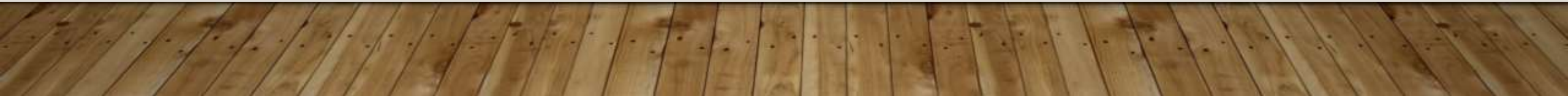
# CUPID: BASADO EN DOMINIO

---

- Lenguaje de dominio y estructura para buen código
- Estructura refleja la solución, no el framework
- Límites del dominio como límites de módulo y unidades de implementación
- Términos de dominio en lugar de conceptos técnicos



**CUPID** no es un reemplazo de  
**SOLID**, sino un enriquecimiento



**GRACIAS POR  
VUESTRA ATENCIÓN**

