

MADS
S9: Diseño SOLID
(Bloque 2 - XP)

El tweet de la semana

The Big Branch Theory is a technology podcast. The screenshot shows the homepage with the title 'THE BIG BRANCH THEORY' and a description of the latest episode: '13 de junio: CUPID, Un amor poco SOLIDO: En este episodio Alfredo Casado y Juan Vega debaten sobre la alternativa a SOLID presentada por Dan North. CUPID -- for joyful coding: <https://dannorth.net/2022/02/10/cupid-for-joyful-coding/> Referencias - Charla de Alfr...'. Below this, there's a section for 'Episodios' with two episodes listed: '13 DE JUNIO' and '20 DE ABRIL'.

A tweet from 'The Big Branch Theory' (@BigBranchTheory) dated August 12, 2021, at 4:59 PM via TweetDeck. The tweet reads: 'Para nuestro último episodio de la temporada continuamos con nuestra serie de SOLID, por lo que este episodio hablaremos de Open Closed. Contamos con la participación de @mangelviera @AlfredoCasado @trikitrok @fran_reyes ¡Esperamos que lo disfruten! thebigbranchtheorypodcast.github.io/post/open-clos...'.

**La única aplicación que no
cambia es la que ha sido un
fracaso y nadie usa.**

Objetivo de un buen diseño

- Poder cambiar fácilmente la aplicación:
- Reducir las dependencias
- Aumentar la cohesión

Dependencias

```
import java.util.logging.Level;
import java.util.logging.Logger;

public class Foo {
    public void method() {
        Logger logger = new Logger();
        logger.log(Level.WARNING, "Log message");
    }
}
```

Cohesión

Cohesion is a measure of the strength of association of the elements inside a module. A highly cohesive module is a collection of statements and data items that should be treated as a whole because they are so closely related. Any attempt to divide them up would only result in increased coupling and decreased readability.

Tom de Marco (1978) *Structured Analysis and System Specification*

Síntomas de un mal diseño

- Rigidez
- Fragilidad
- Inmovilidad
- Viscosidad

2000

Principios SOLID

www.objectmentor.com

1

Design Principles and Design Patterns



Robert C. Martin
www.objectmentor.com

What is software architecture? The answer is multilayered. At the highest level, there are the architecture patterns that define the overall shape and structure of software applications¹. Down a level is the architecture that is specifically related to the pur-

Principios SOLID

- S: Single responsibility
- O: Open-closed
- L: Liskov substitution
- I: Interface segregation
- D: Dependency inversion

Single Responsibility

Un módulo o clase debe tener una única responsabilidad. Martin va más allá y explica en qué consiste el concepto de responsabilidad indicando que una clase debe tener cambiar por una única razón. Si existen más de una posibilidad de cambio de una clase, ya no tiene una única responsabilidad.

No cumple el principio

The screenshot shows a Java API documentation page. At the top, there is a navigation bar with links: OVERVIEW, PACKAGE (which is highlighted in orange), CLASS, USE, TREE, DEPRECATED, INDEX, and HELP. To the right of the navigation bar, it says "Java™ Platform Standard Ed. 8". Below the navigation bar, there are links for PREV PACKAGE, NEXT PACKAGE, FRAMES, NO FRAMES, and ALL CLASSES. The main content area has a dark blue header with the text "Package java.util". Below the header, there is a descriptive paragraph: "Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array)." The rest of the page is white space.

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

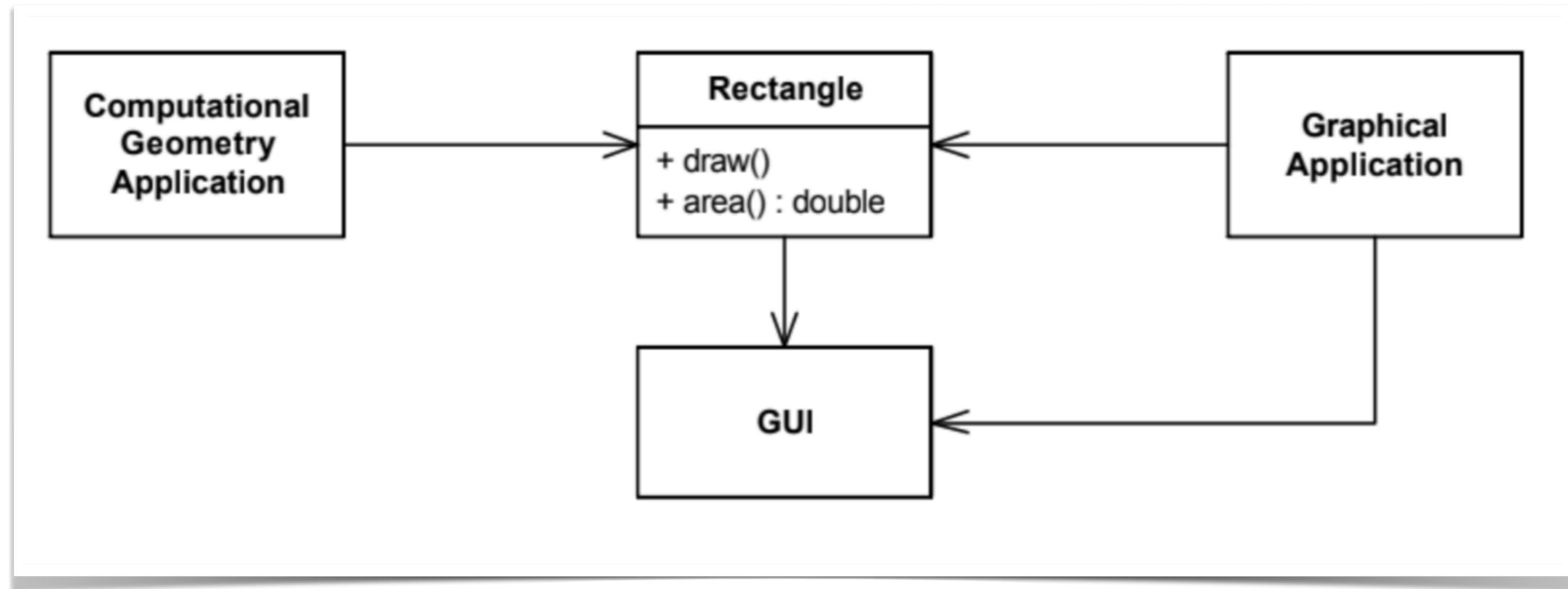
Java™ Platform Standard Ed. 8

PREV PACKAGE NEXT PACKAGE FRAMES NO FRAMES ALL CLASSES

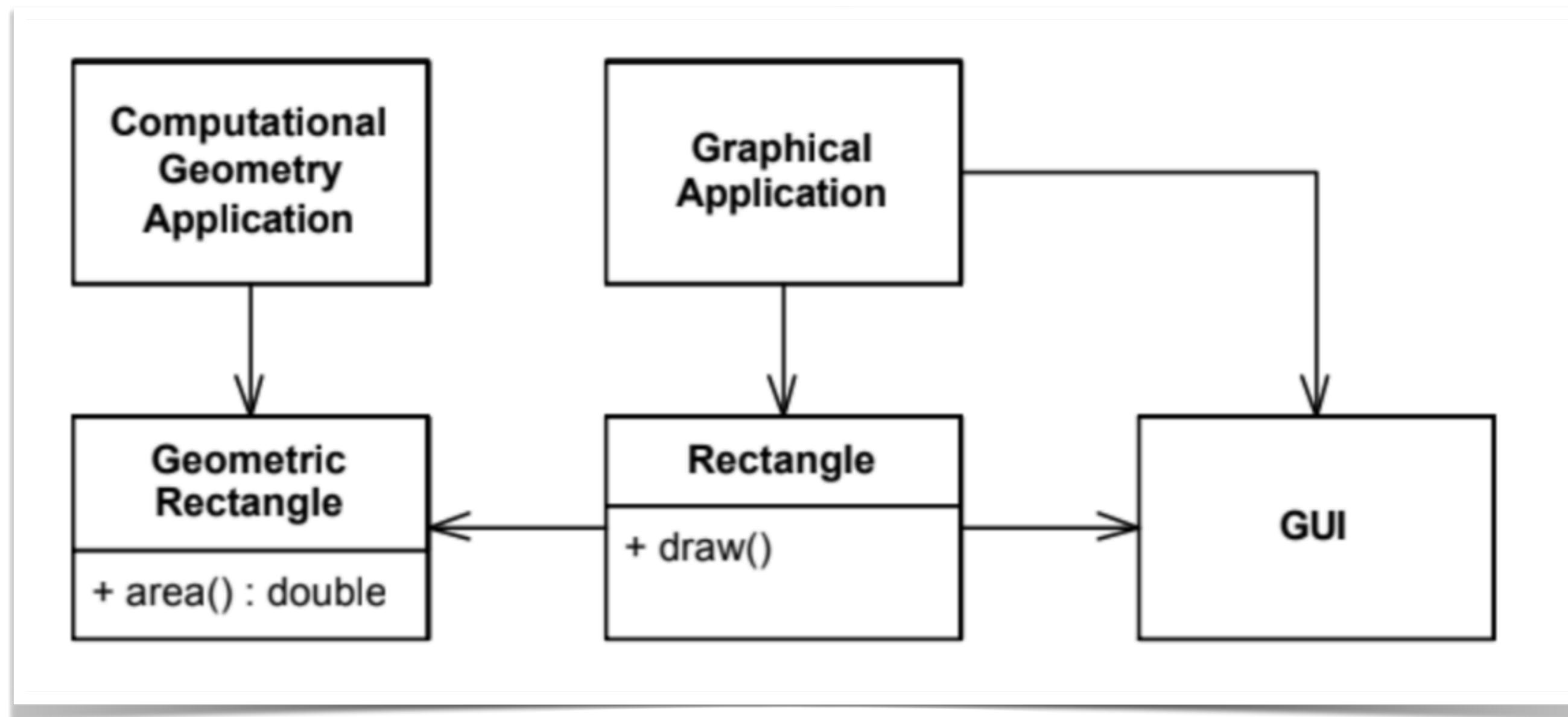
Package java.util

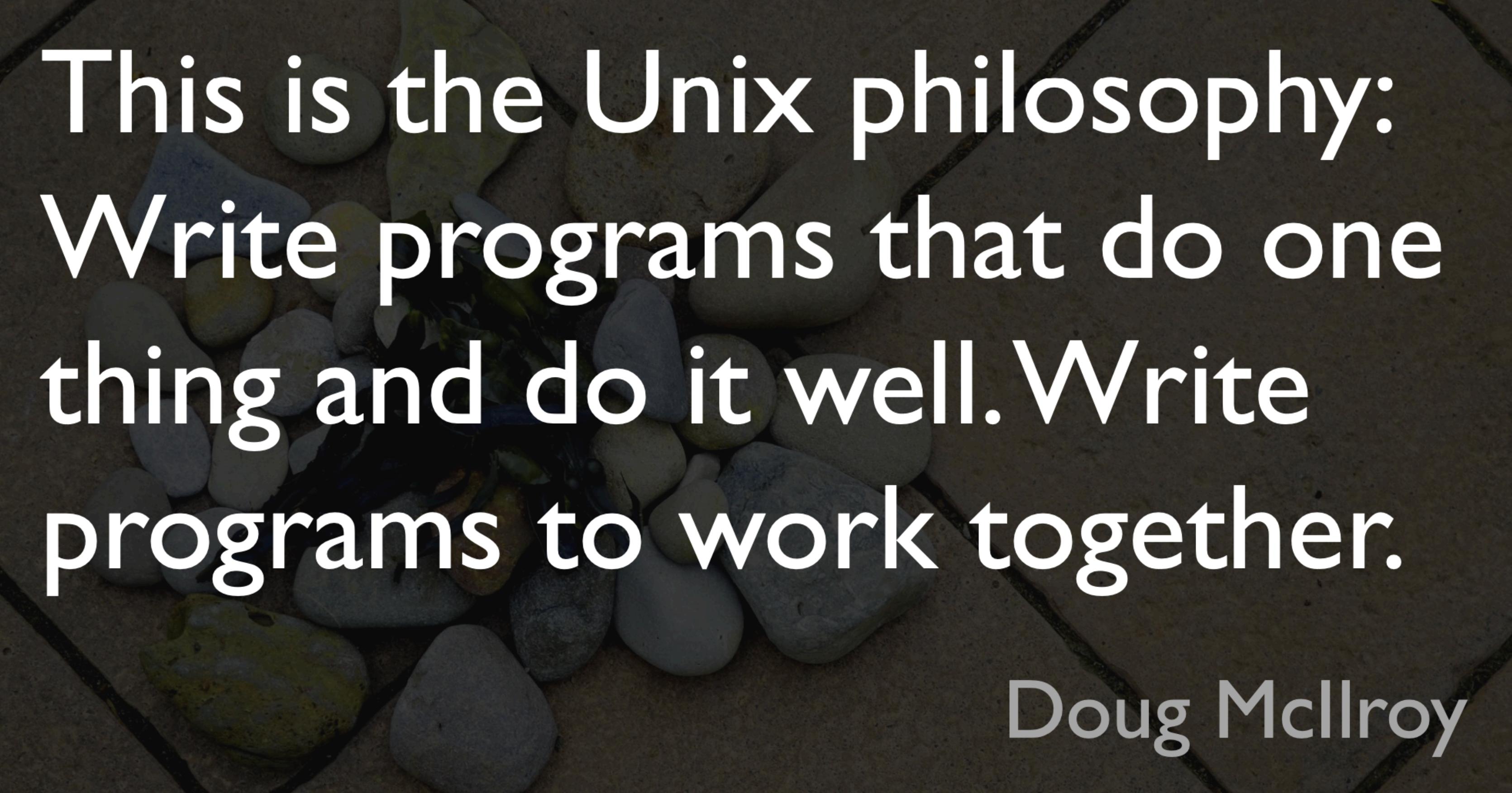
Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

No cumple el principio



Sí lo cumple





This is the Unix philosophy:
Write programs that do one
thing and do it well. Write
programs to work together.

Doug McIlroy



Domingo Gallardo
@domingogallardo

Cualquiera que use filtros de UNIX está usando programación funcional:

```
cat datos.txt | sort | uniq | wc -l
```

Cada filtro recibe unas entradas y las transforma en unas salidas, que recibe el siguiente filtro.

Es la misma idea que se usa en la composición de funciones.

[Translate Tweet](#)

1:22 PM · Feb 2, 2020 · Twitter Web App



Open-Closed

- Se dice que un módulo está abierto si es posible extenderlo. Por ejemplo, debería ser posible añadir campos a las estructuras de datos que contiene o nuevos elementos al conjunto de funciones que realiza.
- Se dice que un módulo está cerrado si es posible usarlo desde otros módulos [de forma estable, sin que cambie en el futuro]. Esto supone que el módulo ha sido bien definido y tiene una interfaz estable [no va cambiar].

Open-Closed

A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.

Bertrand Meyer (1988) *Object-oriented software construction*

Open-Closed

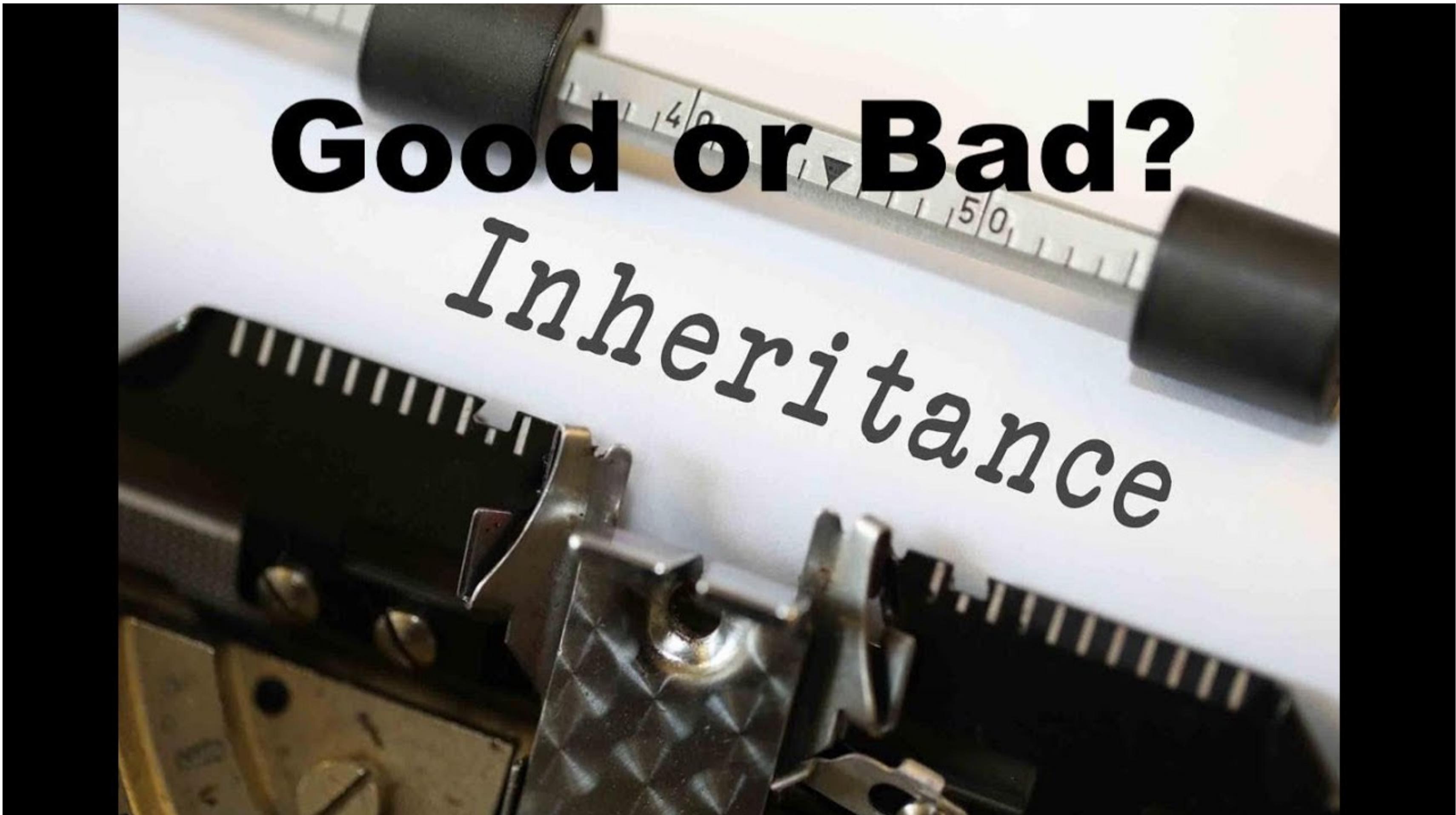
```
public class Guitar {  
  
    private String make;  
    private String model;  
    private int volume;  
  
    //Constructors, getters & setters  
}
```

Open-Closed

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

Good or Bad?

Inheritance



Practical Protocol-Oriented Programming



Nat

Developer Tools

Protocol-Oriented Programming in Swift

Session 408

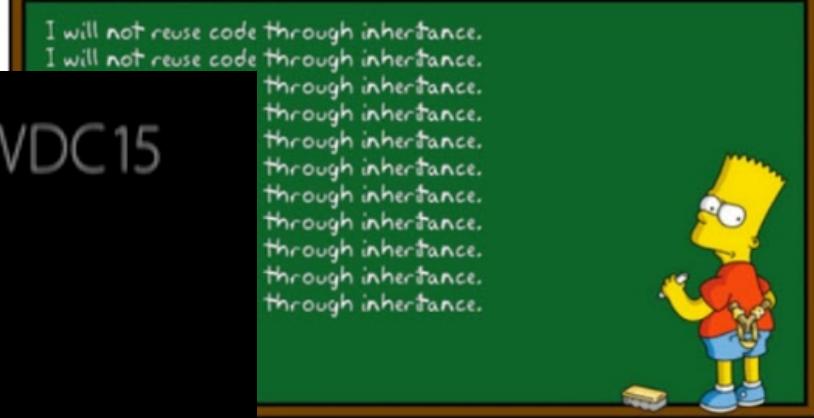
Dave Abrahams Professor of Blowing-Your-Mind

Favour composition over inheritance

Inheritance may be a fundamental feature of OOP, but it is typically over used. The problem with inheritance is that it is not as flexible as it looks. Most things you are trying to achieve with inheritance can also be achieved with composition.

The downside of inheritance is that if any errors are made (i.e. in the base

#WWDC15



1

Mal uso de la herencia

```
public class ListaSencilla {  
    private List datos = new ArrayList();  
  
    public void insertar(String elemento) {  
        datos.add(elemento);  
    }  
  
    public void insertarVarios(Collection elementos) {  
        for (String elemento : elementos)  
            this.insertar(elemento);  
    }  
}
```

2

```
public class ListaAuditable extends ListaSencilla {  
    private Auditor auditor;  
  
    public ListaAuditable(Auditor auditor) {  
        super();  
        this.auditor = auditor;  
    }  
  
    @Override  
    public void insertar(String elemento) {  
        super.insertar(elemento);  
        this.auditor.elementoInsertado(elemento);  
    }  
  
    @Override  
    public void insertarVarios(Collection elementos) {  
        super.insertarVarios(elementos);  
        for (String elemento : elementos)  
            this.auditor.elementoInsertado(elemento);  
    }  
}
```

3

```
public class ListaAuditable extends ListaSencilla {  
    private Auditor auditor;  
  
    public ListaAuditable(Auditor auditor) {  
        super();  
        this.auditor = auditor;  
    }  
  
    @Override  
    public void insertar(String elemento) {  
        super.insertar(elemento);  
        this.auditor.elementoInsertado(elemento);  
    }  
}
```

4

```
public class ListaSencilla implements Lista {  
    private List datos = new ArrayList();  
  
    @Override  
    public void insertar(String elemento) {  
        datos.add(elemento);  
    }  
  
    @Override  
    public void insertarVarios(Collection elementos) {  
        datos.addAll(elementos);  
    }  
}
```

5

```
public class ListaSencilla {  
    private List datos = new ArrayList();  
  
    public void insertar(String elemento) {  
        datos.add(elemento);  
    }  
  
    public void insertarVarios(Collection elementos) {  
        for (String elemento : elementos)  
            this.insertar(elemento);  
    }  
}
```

Doble moraleja:

- Un método de la clase padre no debe basar su implementación en otro método que pueda ser sobreescrita por una clase hija. Si es así, hay que marcar el segundo método como `final` para evitar esa sobrescritura.
- Un método de una clase hija no debe basarse en detalles de implementación de la clase padre. La clase padre puede cambiar de implementación en cualquier momento.

Liskov substitution

A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra.

What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when we use o_1 or o_2 , then S is a subtype of T .

Barbara Liskov (1987) *Data Abstraction and Hierarchy*



El principio nos está indicando otra vez que tenemos que tener cuidado a la hora de usar la herencia. Debemos cumplir siempre el criterio "IS-A" a la hora de definir la clase derivada. Es el típico ejemplo de los animales. Un Gato y un Caballo cumplen "IS-A" con Mamífero , por lo que en cualquier parte en donde usemos un objeto mamífero podremos usar un gato o un caballo.

C#

Copy

```
class ThreeDPoint : TwoDPoint
{
    public readonly int z;

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        this.z = z;
    }

    public override bool Equals(System.Object obj)
    {
        // If parameter cannot be cast to ThreeDPoint return false:
        ThreeDPoint p = obj as ThreeDPoint;
        if ((object)p == null)
        {
            return false;
        }

        // Return true if the fields match:
        return base.Equals(obj) && z == p.z;
    }

    public bool Equals(ThreeDPoint p)
    {
        // Return true if the fields match:
        return base.Equals((TwoDPoint)p) && z == p.z;
    }

    public override int GetHashCode()
```



```
public class Rectangle {  
    double height;  
    double width;  
  
    public Rectangle(double height, double width) {  
        this.height = height;  
        this.width = width;  
    }  
  
    public void setHeight(double height) {  
        this.height = height;  
    }  
  
    public void setWidth(double width) {  
        this.width = width;  
    }  
}
```

```
public class Square extends Rectangle {  
  
    public Square(double side) {  
        super(side, side);  
    }  
  
    @Override  
    public void setWidth(double width) {  
        super.setWidth(width);  
        super.setHeight(width);  
    }  
  
    @Override  
    public void setHeight(double height) {  
        super.setHeight(height);  
        super.setWidth(height);  
    }  
  
    public void setSide(double side) {  
        this.setHeight(side);  
    }  
}
```



Interfaz Segregation

El principio *Interface segregation* establece que ningún cliente debería depender de métodos que no usa. El principio propone dividir interfaces que sean muy grandes en otras más pequeñas y específicas, de forma que los clientes sólo tengan que saber de los métodos que sean de su interés. Estas interfaces especializadas se suelen denominar *Role Interfaces*.

```
public interface LineIO {  
    String read();  
    void write(String lineToWrite);  
}
```

```
public interface LineReader {  
    String read();  
}  
  
public interface LineWriter {  
    void write(String lineToWrite);  
}
```

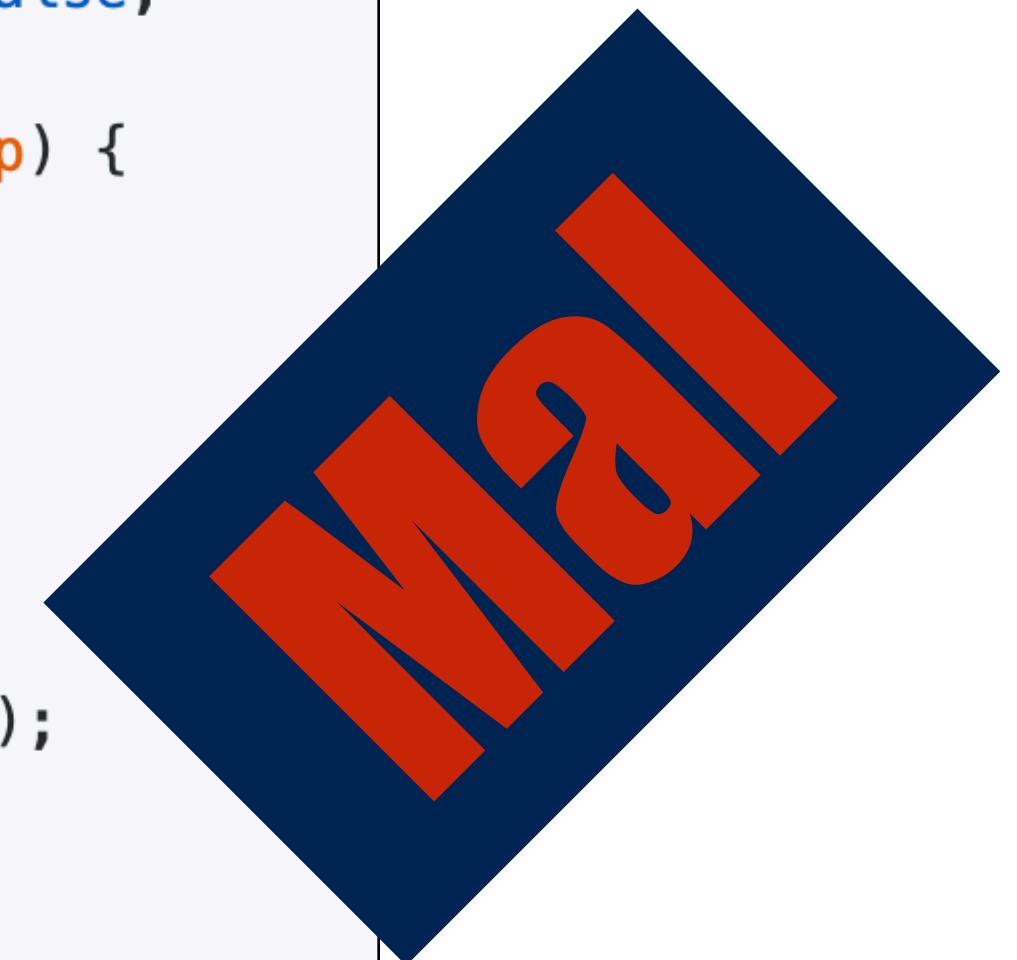
Dependency Inversion

En concreto, el principio se suele definir con las siguientes dos reglas:

1. Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deben depender de abstracciones.
2. Las abstracciones no deberían depender de detalles. Los detalles deberían depender de las abstracciones.

```
public class Lamp {  
  
    public void switchOn() {  
        // Encendemos la lámpara  
    }  
  
    public void switchOff() {  
        // Apagamos la lámpara  
    }  
}
```

```
public class Button {  
  
    private Lamp lamp;  
    private Boolean on = false;  
  
    public Button(Lamp lamp) {  
        this.lamp = lamp;  
    }  
  
    public void turnOn() {  
        if (!on) {  
            on = true;  
            lamp.switchOn();  
        }  
    }  
  
    public void turnOff() {  
        if (on) {  
            on = false;  
            lamp.switchOff();  
        }  
    }  
}
```



```
public interface Switchable {  
    public void switchOn();  
    public void switchOff();  
}  
  
public class Lamp implements Switchable {  
    @Override  
    public void switchOn() {  
        // Encendemos la lámpara  
    }  
  
    @Override  
    public void switchOff() {  
        // Apagamos la lámpara  
    }  
}
```

```
public class TvSet implements Switchable {  
    @Override  
    public void switchOn() {  
        // Encendemos la TV  
    }  
  
    @Override  
    public void switchOff() {  
        // Apagamos la TV  
    }  
}
```

```
public class Button {  
  
    private Switchable device;  
    private Boolean on = false;  
  
    public Button(Switchable device) {  
        this.device = device;  
    }  
  
    public void turnOn() {  
        if (!on) {  
            on = true;  
            device.switchOn();  
        }  
    }  
  
    public void turnOff() {  
        if (on) {  
            on = false;  
            device.switchOff();  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    Lamp lamp = new Lamp();  
    TvSet tv = new TvSet();  
    Button lampButton = new Button(lamp);  
    Button tvButton = new Button(tv);  
    lampButton.turnOn();  
    tvButton.turnOn();  
}
```

```
public interface ILogger {  
    public void log(Level level, String message);  
}  
  
public class StandardLogger implements ILogger {  
    public void log(Level level, String message) {  
        // Implementación  
    }  
}  
  
public class FileLogger implements ILogger {  
    public void log(Level level, String message) {  
        // Implementación  
    }  
}  
  
public LoggerFactory {  
    static public ILogger standardLogger() {  
        return new StandardLogger();  
    }  
    static public ILogger fileLogger() {  
        return new FileLogger();  
    }  
}
```

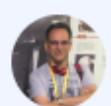
```
public class Foo {  
    private ILogger logger;  
  
    public Foo(ILogger logger) {  
        this.logger = logger;  
    }  
  
    public void method() {  
        logger.log(Level.WARNING, "Log message");  
    }  
    ...  
}  
  
public static void main(String[] args) {  
    Foo foo = new Foo(LoggerFactory.standardLogger());  
    foo.method();  
}
```

Ejemplo de código

The screenshot shows a GitHub repository page for the user 'domingogallardo' with the repository name 'solid'. The 'Code' tab is selected. At the top right, there is an 'Unwatch' button with a count of 1. Below the tabs, there are buttons for 'Go to file', 'Add file', and a green 'Code' button. A message indicates that the 'solid' branch is 5 commits ahead and 2 commits behind the 'main' branch. There is a 'Pull request' and a 'Compare' button. The main content area displays a list of recent commits:

Author	Commit Message	Date	Commits
domingogallardo	Separada la interfaz IPost en interfaces ...	yesterday	6 commits
	src	Separada la interfaz IPost en interfaces más especí...	yesterday
	.gitignore	Versión 1 Blog	5 days ago
	README.md	Versión 1 Blog	5 days ago

Below the commit list, there is a preview of the 'README.md' file, which contains the word 'solid'.

[Code](#)[Issues](#)[Pull requests](#)[Actions](#)[Projects](#)[Wiki](#)[Security](#)[master](#)[apuntes-mads / apuntes / solid / solid.md](#)**domingogallardo** Ejemplo solid

Latest commit 31c

1 contributor

1423 lines (1155 sloc) | 44.1 KB

Diseño de software SOLID

Razones para un buen diseño

Hemos aprendido en otras sesiones de la asignatura que una de las características fundamentales del diseño es la flexibilidad. Los clientes no han sabido lo que quieren o no han sabido expresarlo. Nosotros no hemos conocido sus necesidades, lo hemos ido aprendiendo conforme hemos ido entregándoles versiones de la aplicación. La aplicación es perfecta y ha sido un éxito también cambiará, porque todo el mundo querrá más cosas.