

---

---

# Programación orientada a aspectos

—

David Gantiva  
Diego Duarte  
Alejandro Santamaria

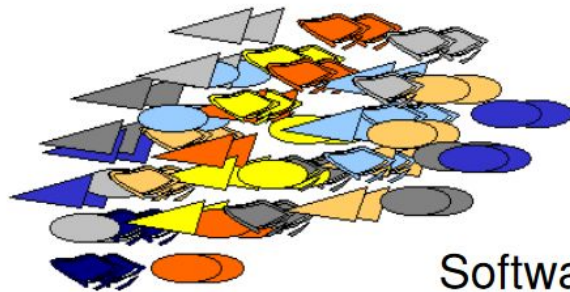
---

---

# Introducción

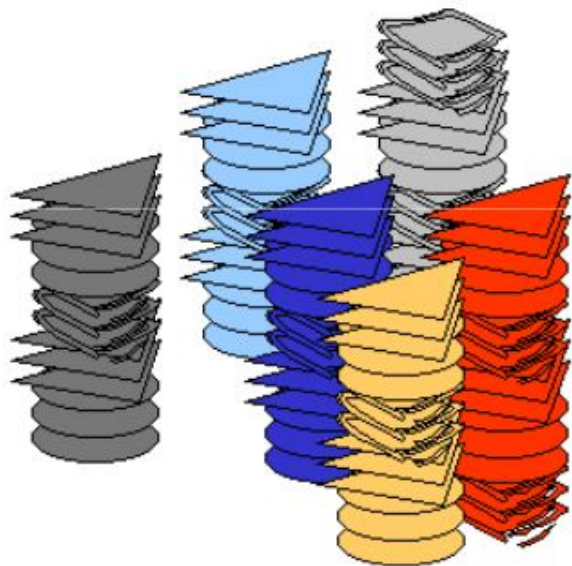
Código Spaghetti: Código donde no existía separación de conceptos.

Datos y funcionalidades mezcladas sin una línea divisoria.



Software= Datos(formas) +  
Funciones(colores)

# Introducción



Programación Funcional: Identificación de funciones que definen el dominio del problema.

Siguen existiendo datos compartidos y esparcidos por todo el código

Software= Datos(formas) +  
Funciones(colores)

# Introducción

Programación orientada a Objetos:

Se ajusta mejor a problemas del dominio real, facilitando la integración de nuevos datos.

Funciones esparcidas por todo el código.

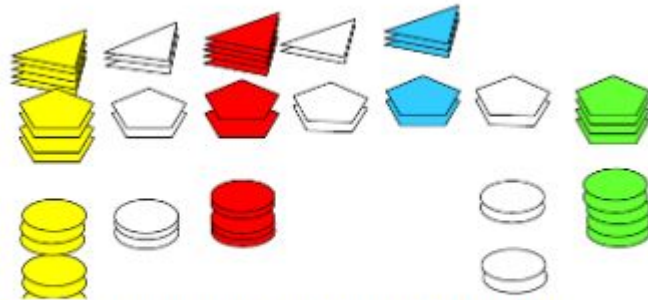


Software= Datos(formas) +  
Funciones(colores)

# Programación Orientada a aspectos

Separación de componentes y *aspectos*. De manera que sea posible abstraerlos y componerlos para formar todo el sistema.

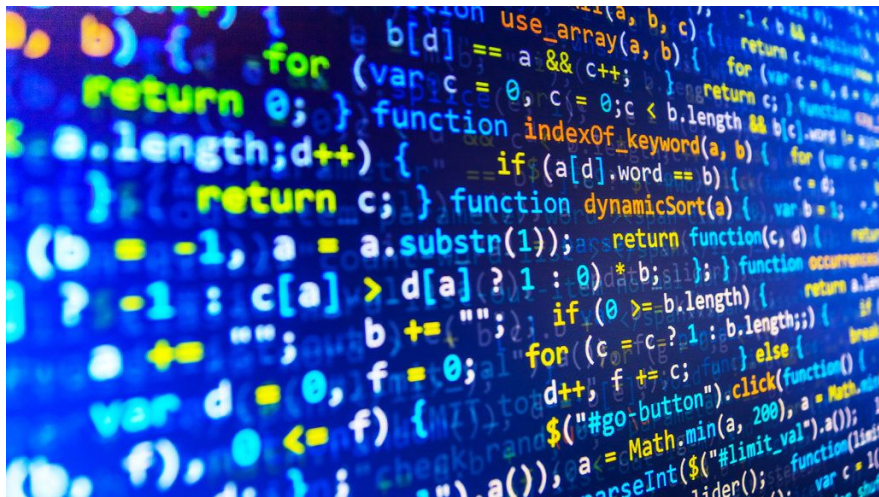
- Componente: Propiedad que se puede encapsular(método, objeto API)
- Aspecto: En los lenguajes tradicionales no se puede encapsular.



**Descomposición en Aspectos**

Software= Datos(formas) +  
Funciones(colores)

# ¿Es la POA un paradigma de programación?

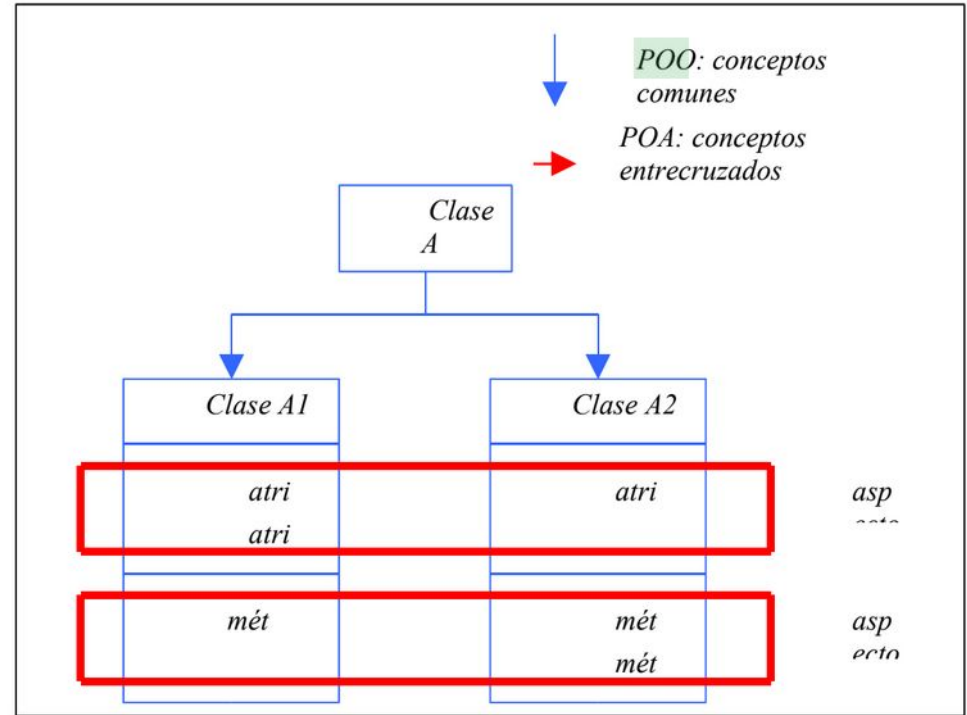


Como paradigma denominamos **todo** aquel modelo, patrón o ejemplo que debe seguirse en determinada **situación**. Por lo tanto, podemos decir, que un **paradigma de programación** es un modelo para desarrollar software.

# POA Y POO

La Programación Orientada a Aspectos **NO** es un reemplazo de la Programación Orientada a Objetos.

La POA es un patron de programacion que complementa la POO.



# Historia

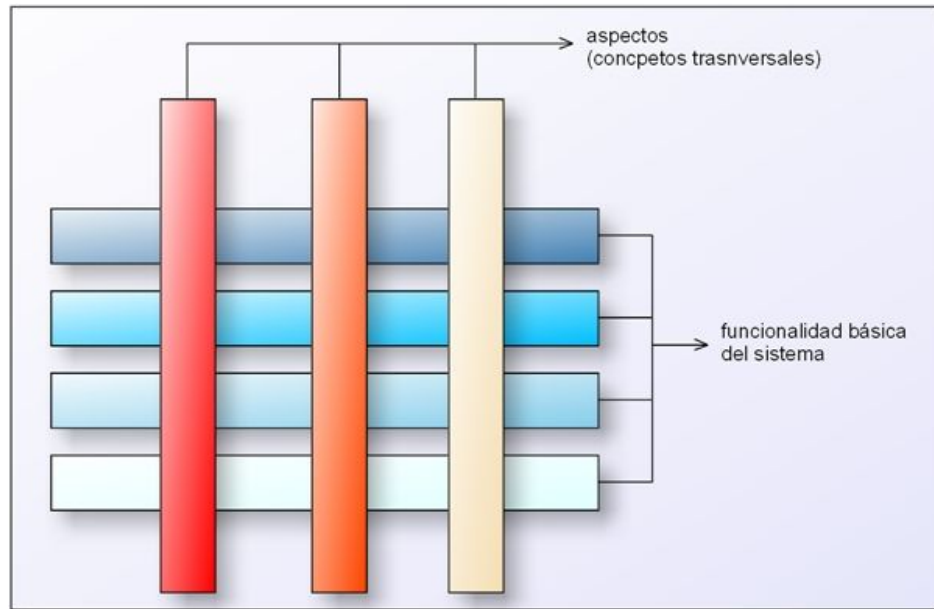


- 1991: Equipo Demeter: programación adaptativa.
- 1995: Introducción del término *aspecto*
- 1997: *Aspect Oriented Programming*: Equipo Demeter(Gregor Kiczales) junto a Cristina Lopes y J. Lieberherr.



# Filosofía

La programación orientada a aspectos busca principalmente tratar las incumbencias transversales de nuestros programas como módulos separados (aspectos) para lograr una correcta separación de las responsabilidades.



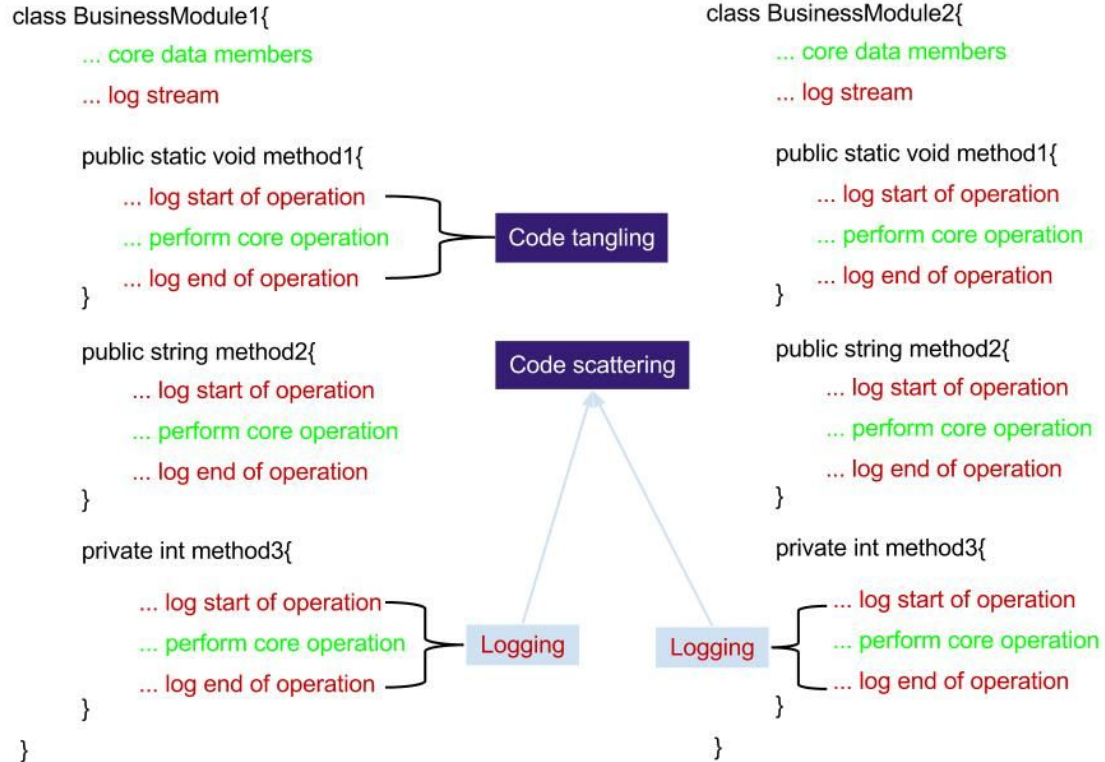
# Incumbencias Transversales

Conceptos diseminados por el código que atraviesan partes del sistema no relacionados en el modelo.



# Necesidades a cubrir

- Código Mezclado (Code Tangling): En un mismo módulo de un sistema de software puede tratarse simultáneamente más de un requerimiento.
- Código Diseminado (Code Scattering): Como los requerimientos están esparcidos sobre varios módulos, la implementación resultante también queda diseminada sobre esos módulos.



# Aspecto

*Unidad modular del programa que aparece en otras unidades funcionales del programa (G. Kiczales).*

En otras palabras: Entidad en la que se encapsulará una funcionalidad común, es decir, una incumbencia transversal modularizada.

**Ejemplos:** patrones de acceso a memoria, sincronización de procesos recurrentes, manejo de errores, registro de actividades, etc.

# Anatomía Aspecto (AspectJ)

```
▪  
aspect MyCrosscuttingConcern{  
    //Attributes  
    //Methods  
  
    //Poincuts  
    //Advices  
}
```

# Joinpoint

Un Joinpoint o punto de unión, es el punto específico de la aplicación, como la ejecución del método, el manejo de excepciones, el cambio de los valores de la variable del objeto, etc.

# PointCut

Los PointCut o punto de corte, son expresiones que coinciden con los puntos de unión para determinar si el Advice debe ser ejecutado o no. Pointcut usa diferentes tipos de expresiones que coinciden con los puntos de unión.

**¿Dónde se ejecutará el aspecto?**

# PointCut y JoinPoint

```
pointcut mi_pointcut():  
    call(void MiClase.metodoX(int))           ||  
    call(void MiClase.metodoY(int))           ||  
    call(void MiOtraClase.metodoXY(int,int)) ||  
    call(void TuClase.metodoZ(MiClase))       ||  
    call(void TuClase.metodoW(MiClase));
```

# Advice

Los Advice son acciones tomadas para un punto de unión en particular. En términos de programación, son métodos que se ejecutan cuando se alcanza un determinado punto de unión con punto de corte coincidente en la aplicación.

```
before() : mi_pointcut() {  
    // cuerpo adicional  
}
```

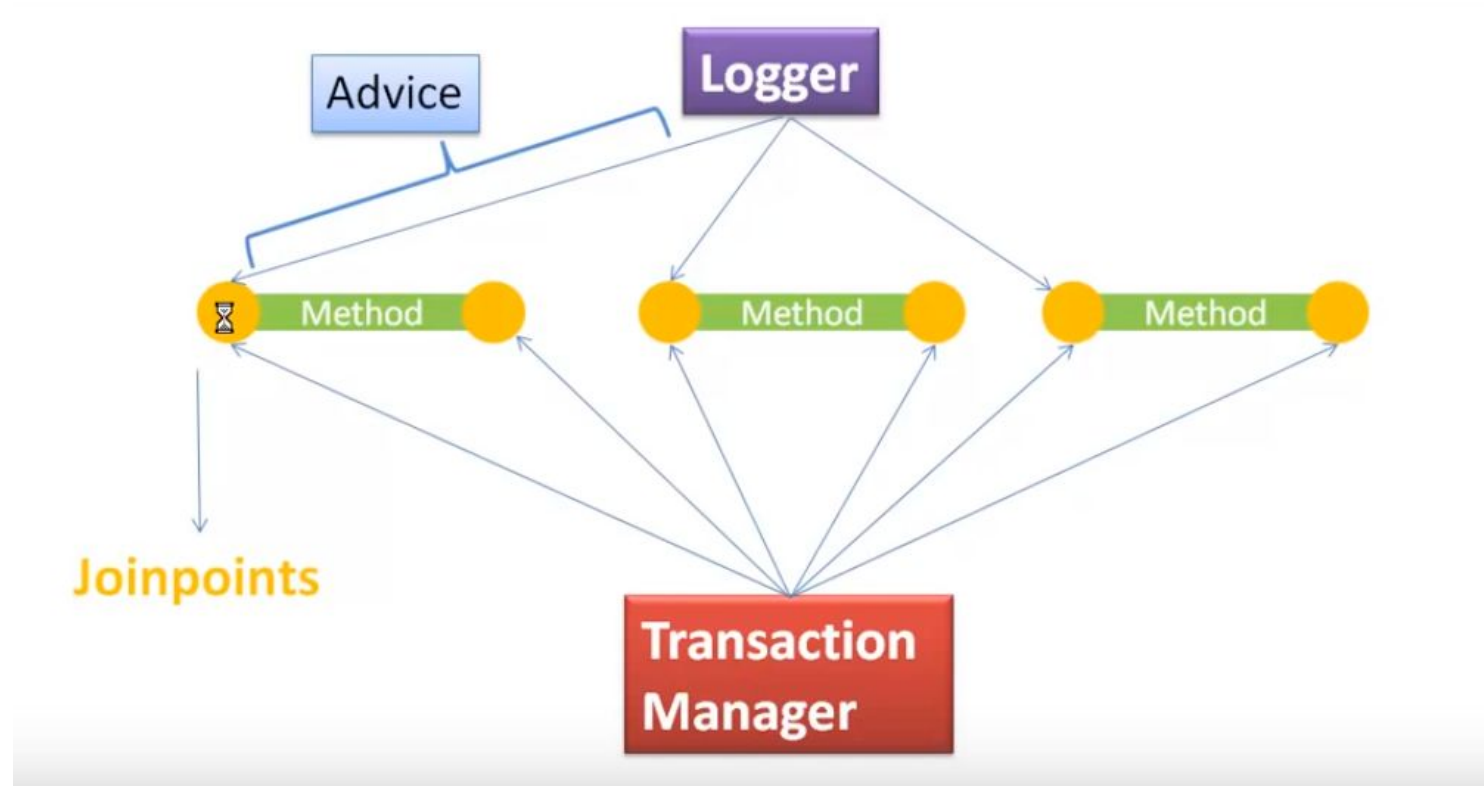
```
after() : mi_pointcut() {  
    // cuerpo adicional  
}
```



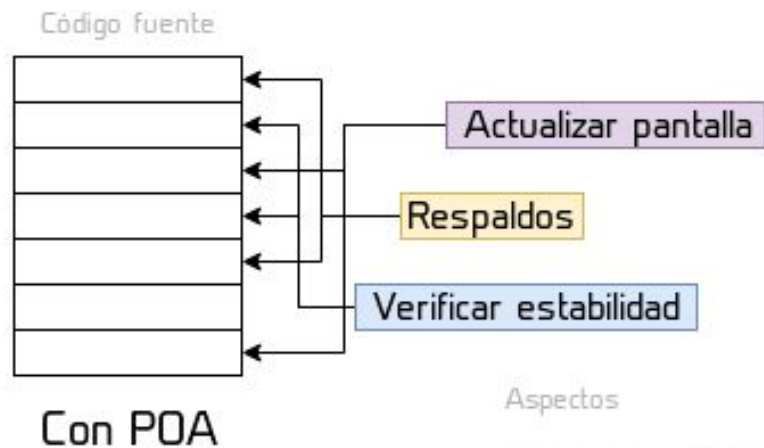
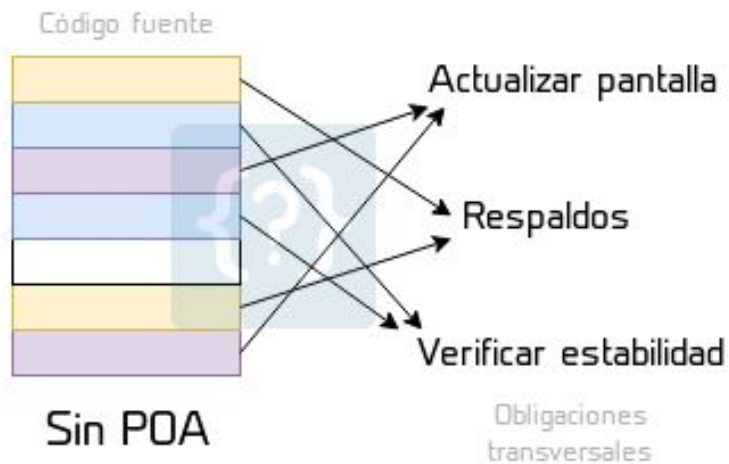
# Advice types

- Before Advice: estos Advice se ejecutan antes de la ejecución de los métodos de los JoinPoints.
- After(finally) Advice: se ejecuta después de que el método de JoinPoint termina de ejecutarse, ya sea normal o lanzando una excepción.
- After Returning Advice: Se ejecuta si el JoinPoint se ejecuta de forma normal.
- After Throwing Advice: Se ejecuta solo si el JoinPoint arroja una excepción.
- Around advice: Este tipo de Advice se ejecutan antes y después de la ejecución de los JoinPoints.

# Funcionamiento



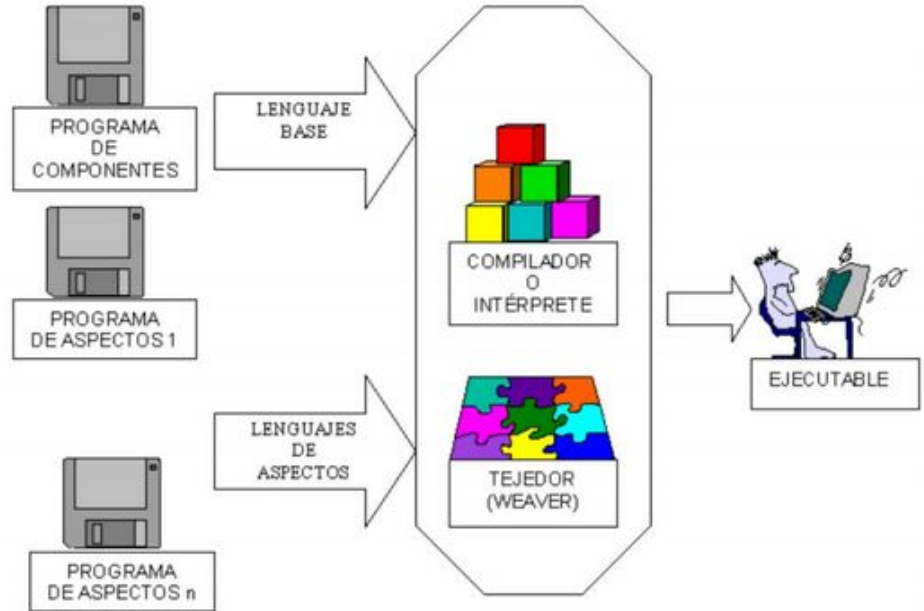
## Ejemplo de funcionamiento de la programación orientada a aspectos (POA)



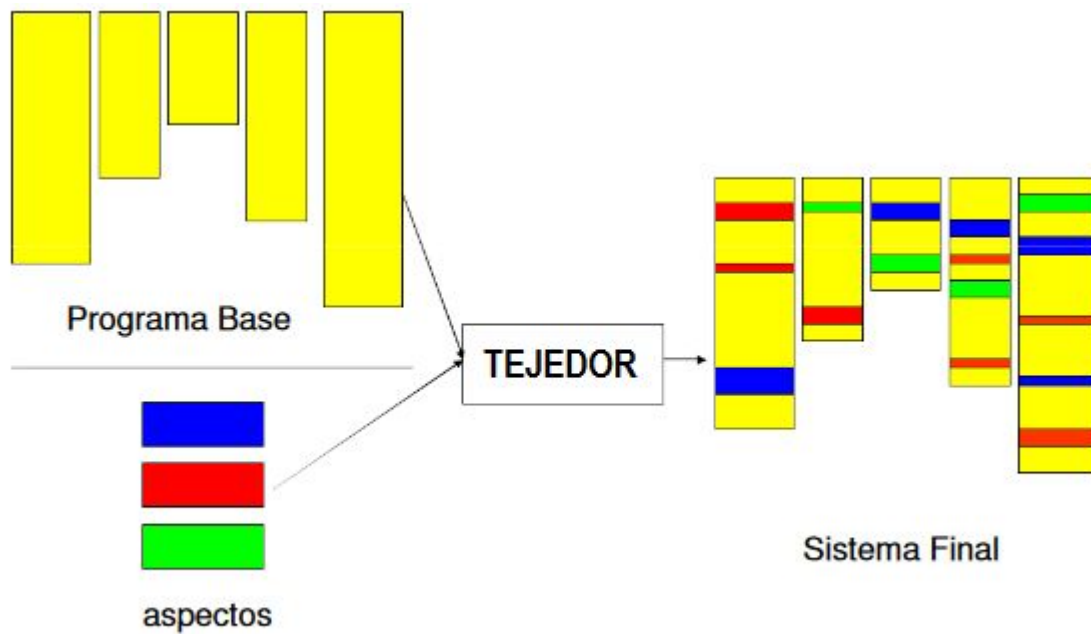
# Aspect Weaver(tejedor de aspectos)

Utilidad de metaprogramación para lenguajes orientados a aspectos.

Toma información de clases y aspectos sin procesar y crea nuevas clases con el código de aspecto entretejido apropiadamente en las clases.



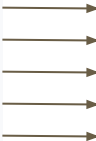
# Tejedor



# Weaving ejemplo

```
aspect Logger {
    pointcut method() : execution(* *(..));
    before() : method() {
        System.out.println("Entering " +
            thisJoinPoint.getSignature().toString());
    }
    after() : method() {
        System.out.println("Leaving " +
            thisJoinPoint.getSignature().toString());
    }
}

public class Foo {
    public void bar() {
        System.out.println("Executing Foo.bar()");
    }
    public void baz() {
        System.out.println("Executing Foo.baz()");
    }
}
```



```
public class Foo {
    public void bar() {
        System.out.println("Entering Foo.bar()");
        System.out.println("Executing Foo.bar()");
        System.out.println("Leaving Foo.bar()");
    }
    public void baz() {
        System.out.println("Entering Foo.baz()");
        System.out.println("Executing Foo.baz()");
        System.out.println("Leaving Foo.baz()");
    }
}
```

# Aspecto ejemplo | AspectJ

```
public class Test {  
  
    public void Say(String name) {  
        System.out.println(name);  
    }  
    public static void main(String[] args) {  
        Test test = new Test();  
  
        test.Say("Hello");  
  
        System.out.println("She say - ");  
  
        test.Say("bye");  
    }  
}
```

**Salida del programa:**

```
public aspect Pensando {  
  
    pointcut speak() : call(public void Say(String));  
  
    before() : speak() {  
        System.out.println("Before Say - Think");  
    }  
  
    after() : speak() {  
        System.out.println("After Say - Smile");  
    }  
}
```

Resultado:  
Before Say - Think  
Hello  
After Say - Smile  
She say -  
Before Say - Think  
bye  
After Say - Smile

---

# Ejemplo II

```
class Point {
    int _x = 0;
    int _y = 0;

    void set (int x, int y) {
        _x = x; _y = y;
    }

    void setX (int x) { _x = x; }

    void setY (int y) { _y = y; }

    int getX() { return _x; }

    int getY() { return _y; }
}
```

```
aspect ShowAccesses {
    static before Point.set,
                Point.setX,
                Point.setY {
        System.out.println("W");
    }
}
```

```
class Point {
    int _x = 0;
    int _y = 0;

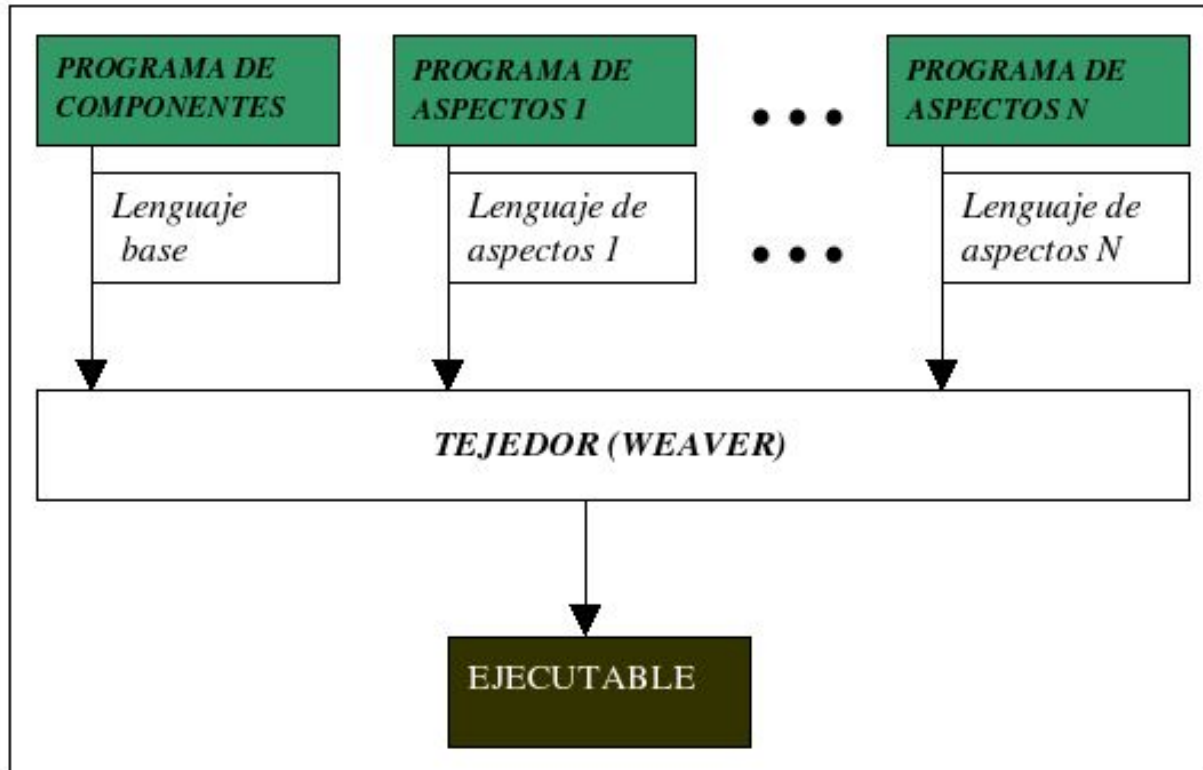
    void set (int x, int y) {
        System.out.println("W");
        _x = x; _y = y;
    }

    void setX (int x) {
        System.out.println("W");
        _x = x;
    }

    void setY (int y) {
        System.out.println("W");
        _y = y;
    }
    .
    .
}
```



# Estructura de la POA



# Lenguajes orientados a aspectos

Los LOA (Lenguajes Orientados a Aspectos) : lenguajes que permiten separar la definición de la funcionalidad pura de la definición de los diferentes aspectos.

- Cada aspecto debe ser claramente identificable.
- Los aspectos deben ser fácilmente intercambiables.
- Los aspectos no deben interferir entre ellos.
- Los aspectos no deben interferir con los mecanismos usados para definir y evolucionar la funcionalidad, como la herencia.
- Cada aspecto debe auto-contenerse.

# Lenguajes orientados a aspectos (Propósito General)

- JPAL
- D
- GO!
- ASPECT(PERL)
- ASPECTC++
- ASPECTJ
- SPRING



J-PAL



spring



# GO!

GO! es un moderno framework orientado a aspectos en PHP con con excelentes funciones para el nuevo nivel de desarrollo de software. El framework permite resolver los problemas de cross-cutting que se presentan en el tradicional código orientado a objetos de PHP proveyendo una alta eficiencia para el código existente

# SPRING

Este framework es implementado en java puro con las anotaciones de java @Aspect o basado en esquema con un XML.



```
package com.xyz.someapp;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemArchitecture {

    @Pointcut("execution(* com.xyz.someapp.service.*.*(..))")
    public void businessService() {}

}
```

```
<aop:config>
  <aop:advisor
    pointcut="com.xyz.someapp.SystemArchitecture.businessService()"
    advice-ref="tx-advice"/>
</aop:config>

<tx:advice id="tx-advice">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

# ASPECTS

AspectS, un lenguaje de aspectos de propósito general, utiliza el modelo de lenguaje de AspectJ y ayuda a descubrir la relación que hay entre los aspectos y los ambientes dinámicos. Soporta programación en un metanivel, manejando el fenómeno de Código Mezclado a través de módulos de aspectos relacionados. Está implementado en Squeak sin cambiar la sintaxis, ni la máquina virtual.



# Aspect (Perl)

Perl es un acrónimo de Practical Extracting and Reporting Language. Es práctico para extraer información de archivos de texto y generar informes a partir del contenido de los ficheros.



```
package My::Aspect;

my $switch = 1;

before {
    print "Calling Foo::bar\n";
} call 'Foo::bar' & true { $switch };

sub enable {
    $switch = 1;
}

sub disable {
    $switch = 0;
}

1;
```

# Python

Python no necesita de ningún plug-in a librería para hacer POA. ya que esto lo puede lograr mediante los decorators.

Existen algunos proyectos como Aspyct, sin embargo este fue abandonado.

```
# Create our decorator
def simple_decorator(func):
    # This is were your neurons should start heating up...
    def wrapper():
        '''Prints "before" and "after" around `func`'''
        print("before")
        func()
        print("after")

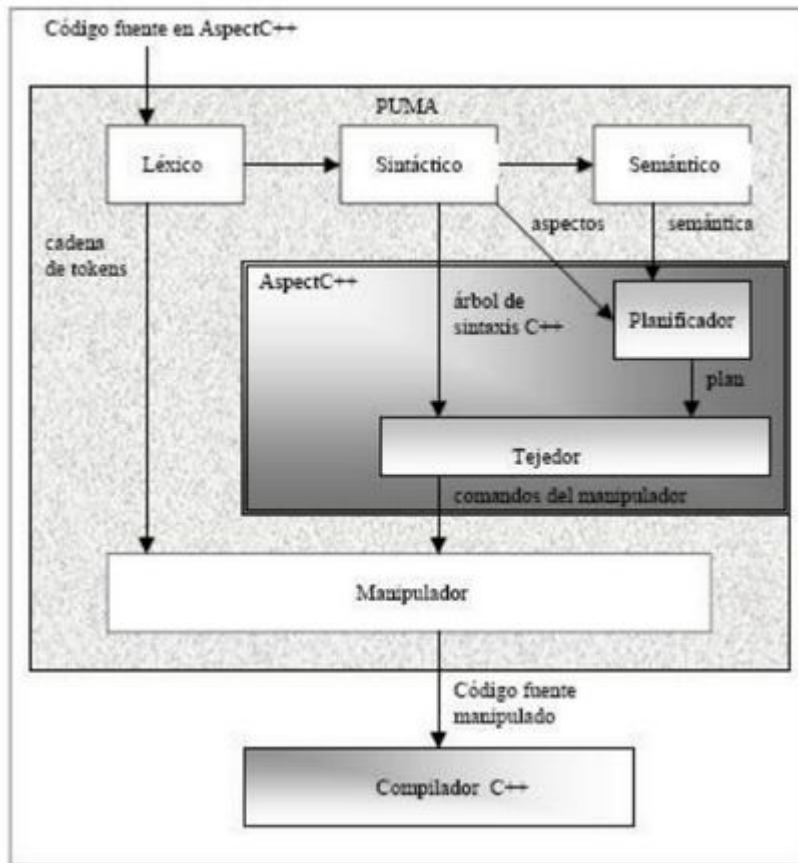
    return wrapper

# Now decorate a function with the '@' sign
@simple_decorator
def test():
    print("inside")
```

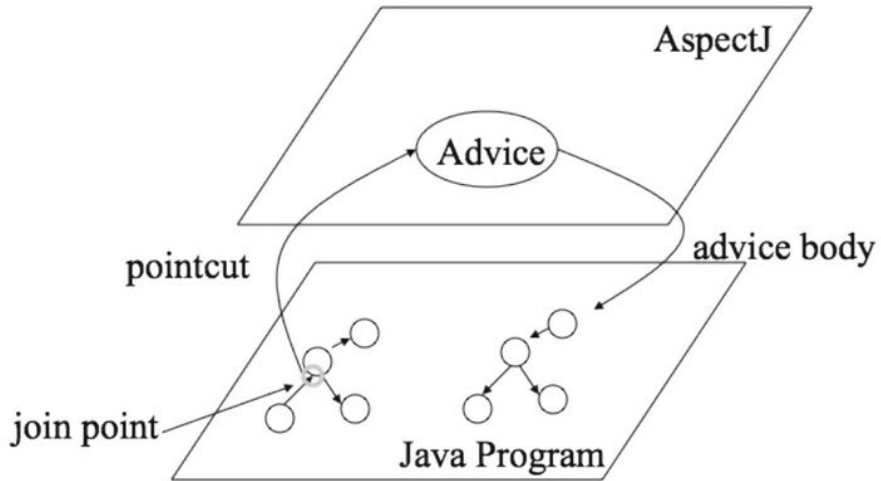


# ASPECTC++

AspectC++ es una extensión orientada a aspectos de los lenguajes C y C++. Tiene un compilador source-to-source, el cual traduce el código fuente de AspectC++ en código compatible con C++.



# ASPECTJ



Es un lenguaje de programación orientado por aspectos construido como una extensión del lenguaje Java creado en Xerox PARC. Un compilador de AspectJ hace llegar la noción de aspecto hacia el código de máquina virtual implementando así una noción de relación.

# Lenguajes Dominio específico.

- COOL(COOrdination Language): trata los aspectos de sincronismo entre hilos concurrentes. (Java)
- RIDL (Remote Interaction and Data transfers aspect Language)
- MALAJ (Multi Aspect LAnguage for Java)
- KALA: Modelos transaccionales avanzados
- DIE: Un lenguaje de aspectos de dominio específico para eventos de IDE
- HYPERJ
- AspectG(ANTLR)
- AspectMatlab

# APLICACIONES

# The a-kernel Project

El objetivo del proyecto a-kernel es determinar si la programación orientada a aspectos se puede utilizar para mejorar la modularidad del sistema operativo y, por lo tanto, reducir la complejidad y la fragilidad asociadas con la implementación del sistema.

# FACET

El objetivo del proyecto FACET es investigar el desarrollo de middleware personalizable utilizando métodos de programación orientados a aspectos.

Se espera que el uso de aspectos en middleware tendrá los siguientes beneficios:

- Mejor modularización de características utilizando aspectos
- Reducción del código de middleware mediante la activación selectiva de funciones

# FACET

**Framework for Aspect Composition for an Event channel**

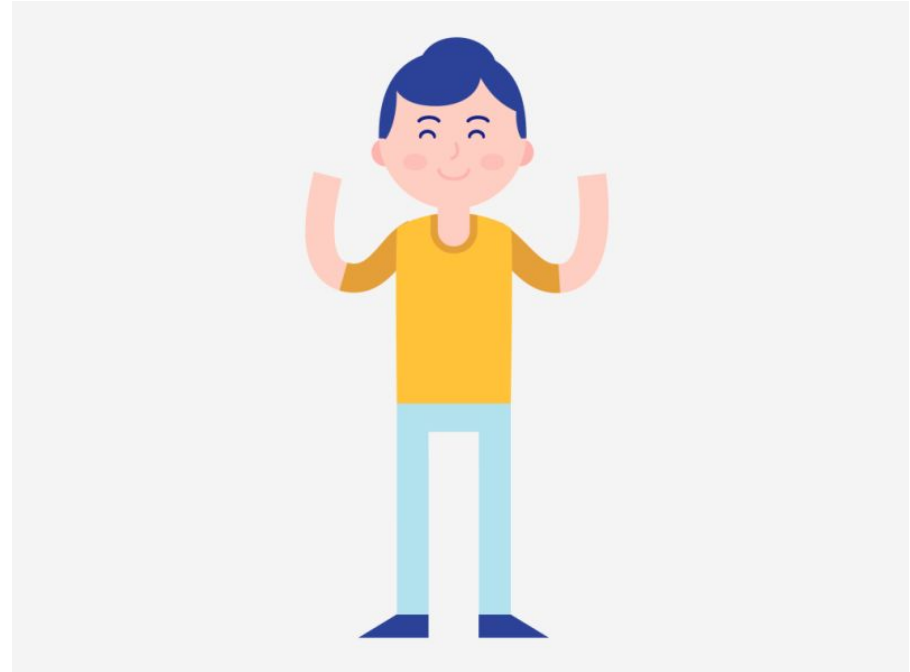
# ¿Cuándo usarlo?

- Manejo de transacciones
- Sincronización
- Manejo de Memoria
- Control de Acceso o Seguridad
- Logging
- Manejo de Excepciones



# Ventajas

- Código fuente más limpio.
- Mayor evolucionabilidad.
- Mayor reusabilidad.
- Facilidad de modificar el código.
- Puede mezclarse con cualquier otro paradigma de programación.





# Desventajas

- Posibles choques entre aspectos.
- Posibles choques entre el código de aspectos y los mecanismos del lenguaje.
- Problemas de herencia.



# Bibliografía

- <https://www.journaldev.com/2583/spring-aop-example-tutorial-aspect-advice-pointcut-joinpoint-annotations>
- [https://en.wikipedia.org/wiki/Aspect-oriented\\_software\\_development](https://en.wikipedia.org/wiki/Aspect-oriented_software_development)
- [https://en.wikipedia.org/wiki/Aspect-oriented\\_programming](https://en.wikipedia.org/wiki/Aspect-oriented_programming)
- <https://codingornot.com/que-es-la-programacion-orientada-a-aspectos-aop>
- <https://web.archive.org/web/20030722015207/http://aosd.net:80/technology/research.php>
- <https://ldc.usb.ve/~yudith/docencia/ci-4822/presentacionPOA.pdf>

# Bibliografía

- Visión General de la Programación Orientada a Aspectos. Antonia M. Reina, 2000. <http://www.lsi.us.es/docs/informes/aopv3.pdf>
- Aspect Oriented Programming w/ AspectJ. Cristina Lopes, Gregor Kiczales, 1998. <http://slideplayer.com/slide/8424141/>
- Programemos en AspectJ. Mario Lorente, Fernando Asteasuain, Bernardo Contreras, 2005.  
<https://drive.google.com/file/d/0Bx1Aq3vYhs9QODVhMDFiYjQtYmE4Zi00MmNiLTg3MmEtMGY0MjkzNDFiNWJk/view>