Programación orientada a aspectos

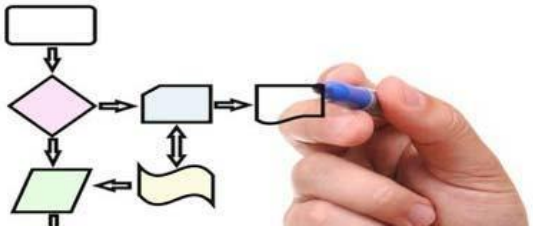
Dorian Abad Tovar Díaz
Gustavo Andrés Galvis Cifuentes
Diego Clemente Rojas Chingaté

Contenido

- Introducción
- Historia
- Filosofia del paradigma
- Fundamentos - Ejemplos
- Ventajas y desventajas
- Lenguajes de programación
- Aplicaciones

Introducción

- Podemos considerar que el progreso de la ingeniería de software se ha presentado en 3 etapas importantes a lo largo de la historia



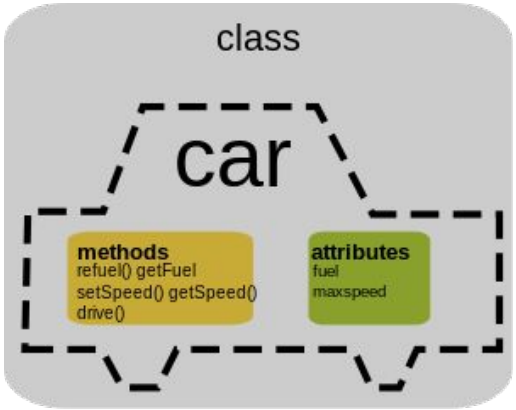
- Al inicio del desarrollo de los lenguajes de programación, no existía una separación de conceptos, datos ni funcionalidades clara, esto es conocido como la etapa del código spaghetti

Introducción

- Nace la descomposición funcional, que pone en práctica el principio de ‘divide y vencerás’, implementando las llamadas funciones, que separan piezas de código que realizan tareas específicas y que puedan reutilizarse

- Las funciones quedan algunas veces poco claras debido al uso de datos compartidos y estos quedan esparcidos por todo el código

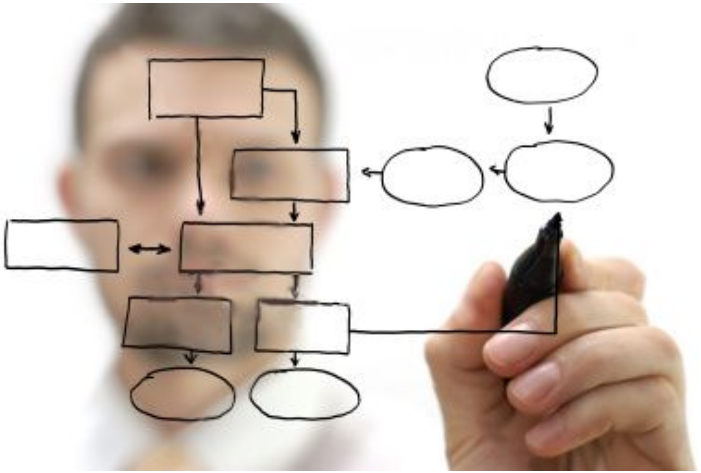
Introducción



- La programación orientada a objetos (POO) dio un gran paso en el desarrollo de la ingeniería de software, haciendo uso del principio de descomposición, ya que el modelo de objetos se ajusta mejor a la abstracción necesaria en los problemas de dominio real
- También presenta que las funciones quedan esparcidas por todo el código y algunas veces, para integrar nuevas funciones hay que modificar varios objetos

Introducción

- Estas metodologías tienen el inconveniente que no consideran un buen tratamiento de aspectos como la gestión de memoria, coordinación, distribución, restricciones de tiempo real, sincronización, gestión de seguridad, etc.



Un poco de historia

- La POA fue introducida por Gregor Kiczales, aunque el grupo Demeter había utilizado ideas orientadas a aspectos anteriormente

- Demeter trabajaba en la llamada programación adaptativa, que puede considerarse como antecesora de la POA, alrededor de 1991

Un poco de historia

- En 1995, Demeter publicó la primera definición de aspecto y gracias a la colaboración de Cristina Lopes y Karl J. Lieberherr con Gregor Kiczales introdujeron el término de programación orientada a aspectos

- Principalmente se buscaba separar conceptos y minimizar las dependencias entre ellos, buscaba, en pocas palabras, que cada cosa esté en su sitio, que cada decisión se tome en un lugar concreto

Problematica

- Las técnicas tradicionales no soportan de una manera adecuada la separación de las propiedades de aspectos distintos a la funcionalidad básica.
- Se tiende a implementar los requerimientos usando metodologías de una sola dimensión.
- Una dimensión es adecuada para la funcionalidad base, mas no para los demás requerimientos, por lo que éstos quedarán a lo largo de la dimensión base.

Inconvenientes

- Código Mezclado (Code Tangling): En un mismo módulo de un sistema de software pueden simultáneamente convivir 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.

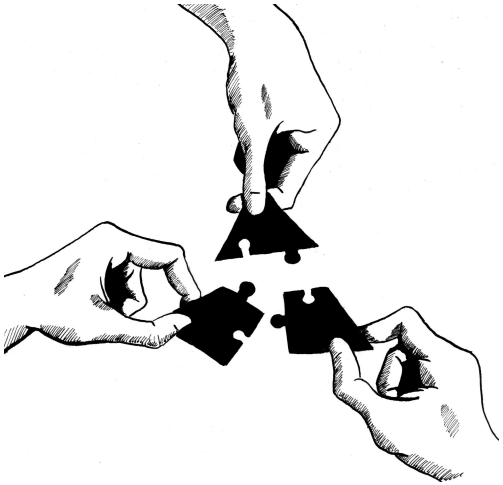
Consecuencias

- **Baja correspondencia:** Implementar varios requerimientos simultáneamente oscurece la relación entre un concepto y su implementación, dificultando la modularidad de estos aspectos.
- **Menor productividad:** Distrae al desarrollador del concepto principal, por concentrarse también en los conceptos periféricos, disminuyendo la productividad.
- **Menor reuso:** Al estar controlando estos requerimientos en cada módulo, resulta en un código poco reusable.

Consecuencias

- **Baja calidad de código:** El código mezclado produce un código propenso a errores. Además, al tener como objetivo demasiados conceptos al mismo tiempo se corre el riesgo de que algunos de ellos sean subestimados.
- **Evolución más dificultosa:** Resulta necesario revisar cada módulo en donde haya sido implementado un concepto cuando se requiera realizar un cambio. Esta tarea se vuelve compleja debido a la insuficiente modularización.

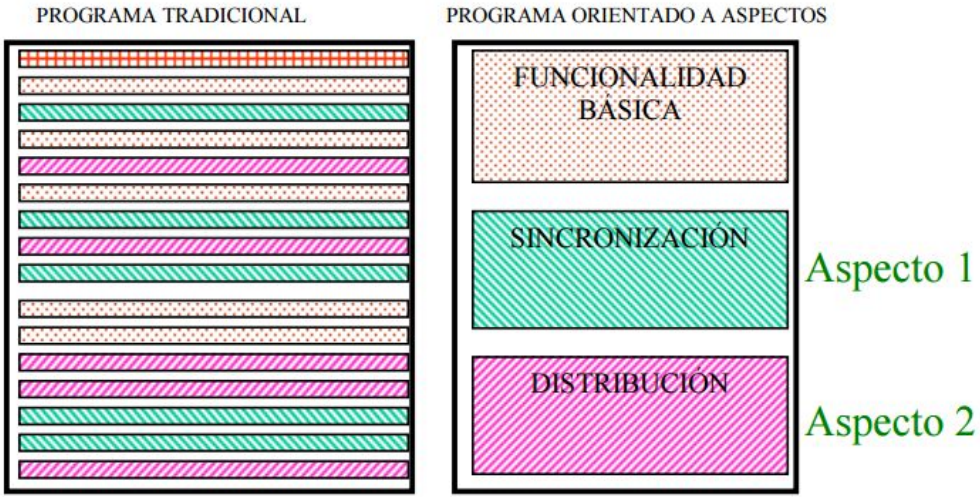
Filosofia del paradigma



- La programación orientada a aspectos (POA) es una metodología de programación que aspira a soportar la separación de competencias para los aspectos antes mencionados
- Permite a los programadores escribir, ver y editar un aspecto diseminado por todo el sistema como una entidad por separado, de una manera inteligente, eficiente e intuitiva.

Filosofia del paradigma

- Intenta separar los componentes y aspectos unos de otros, proporcionando mecanismos que hagan posible abstraerlos y componerlos para formar todo un sistema.



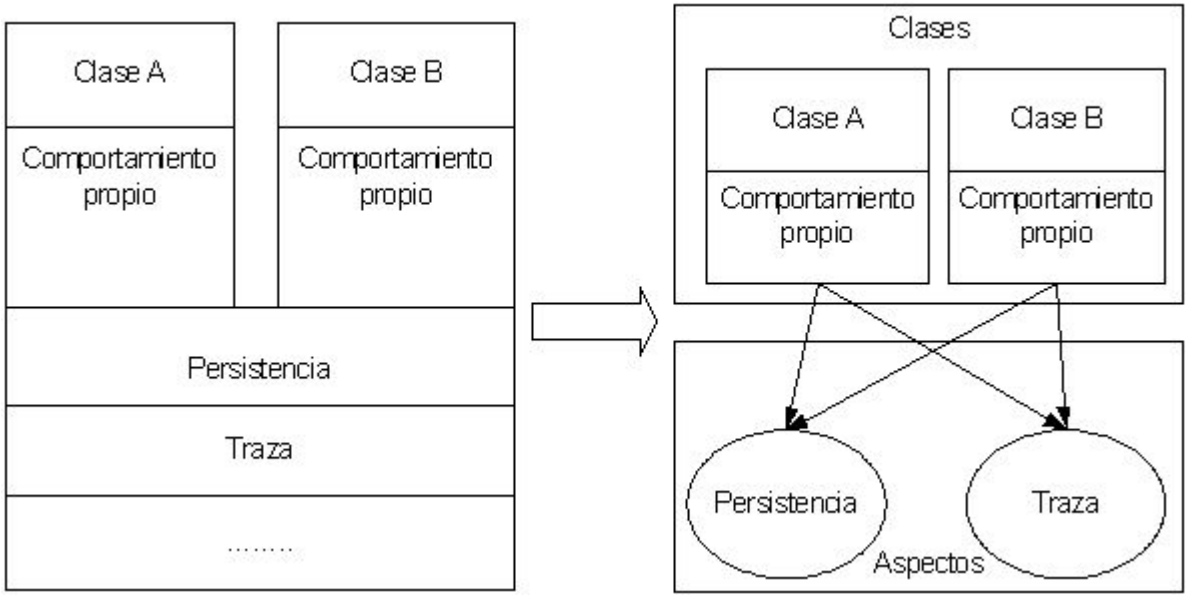
Estructura de un programa POA

Se realiza un programa formando un sistema a partir de un conjunto de aspectos y un modelo de objetos.

- En el modelo de objetos se implementa la funcionalidad básica.
- El conjunto de aspectos implementa las características de rendimiento, seguridad, gestión de errores y otras no relacionadas con la funcionalidad esencial.

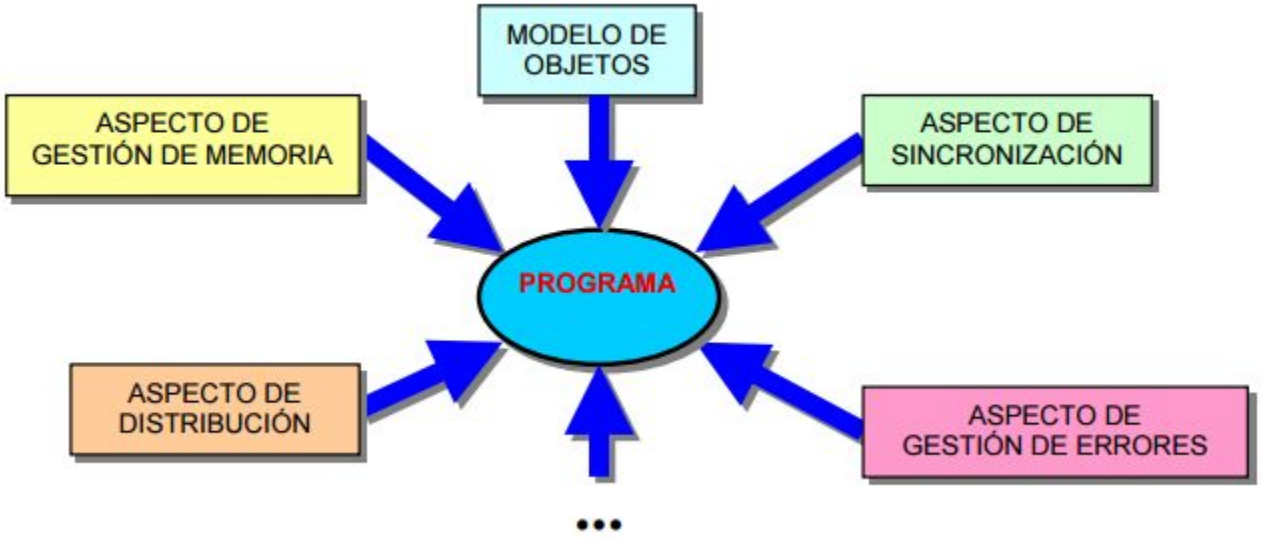
Estructura de un programa POA

Cada aspecto se centraliza en un punto, y las clases que necesiten este requerimiento pueden hacer uso de él.



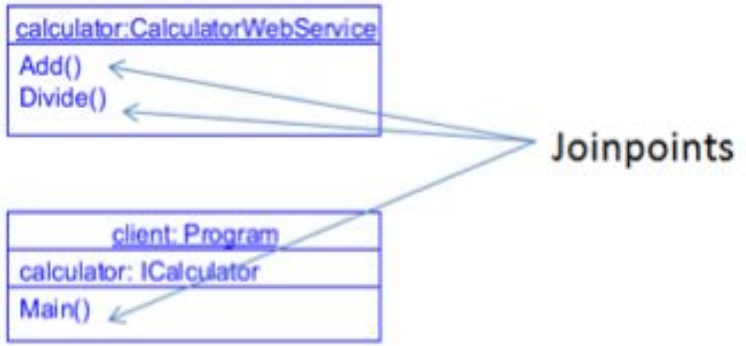
Conceptos básicos

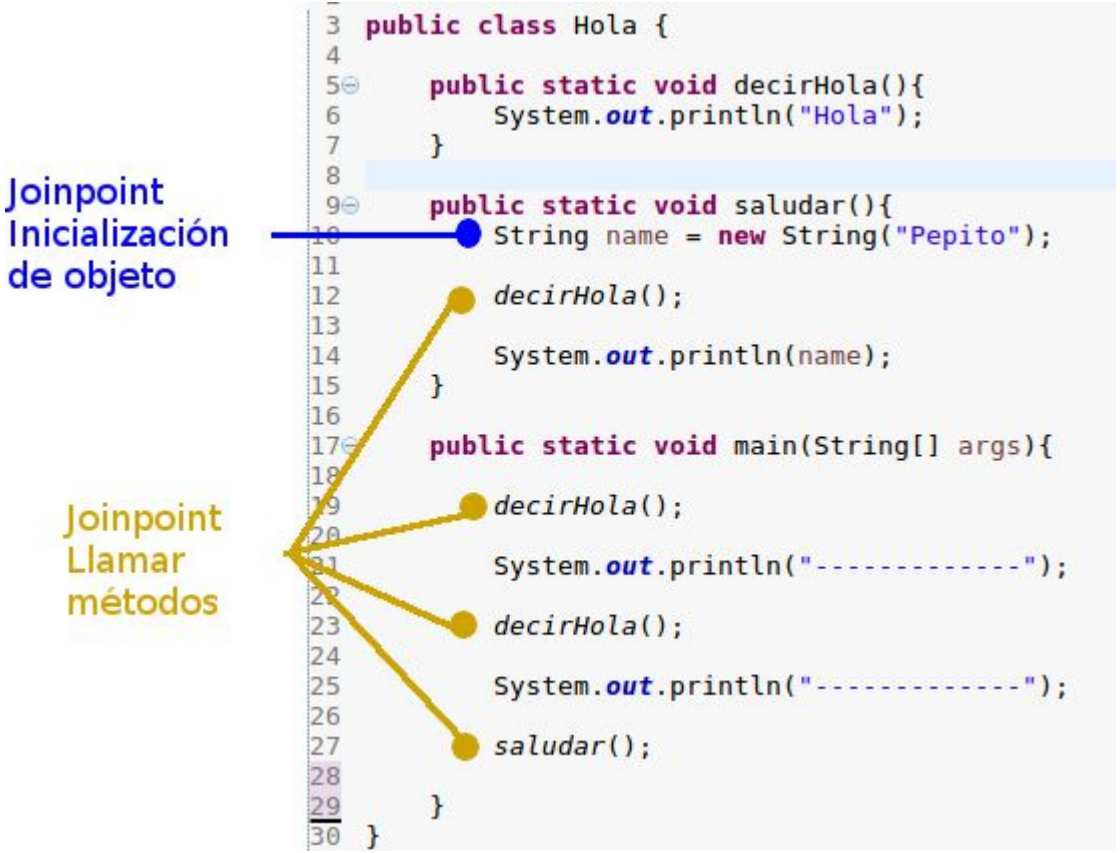
Aspecto (*Aspect*): es una unidad modular que se disemina por la estructura de otras unidades funcionales.



Conceptos básicos

Punto de enlace (*Join point*) Son una clase especial de interfaces que conectan los aspectos con los módulos que describen a los componentes, son los lugares del código en los que se puede aumentar la funcionalidad con comportamientos adicionales.




Joinpoint
Inicialización
de objeto
Joinpoint
Llamar
métodos
3 public class Hola {
4
5 public static void decirHola(){
6 System.out.println("Hola");
7 }
8
9 public static void saludar(){
10 String name = new String("Pepito");
11
12 decirHola();
13
14 System.out.println(name);
15 }
16
17 public static void main(String[] args){
18
19 decirHola();
20
21 System.out.println("-------------");
22
23 decirHola();
24
25 System.out.println("-------------");
26
27 saludar();
28
29 }
30 }

Conceptos básicos

Los conceptos no son totalmente independientes entre ellos, y hay una relación entre los componentes y los aspectos, por lo tanto, el código de los componentes y de estas nuevas unidades de programación tienen que interactuar de alguna manera

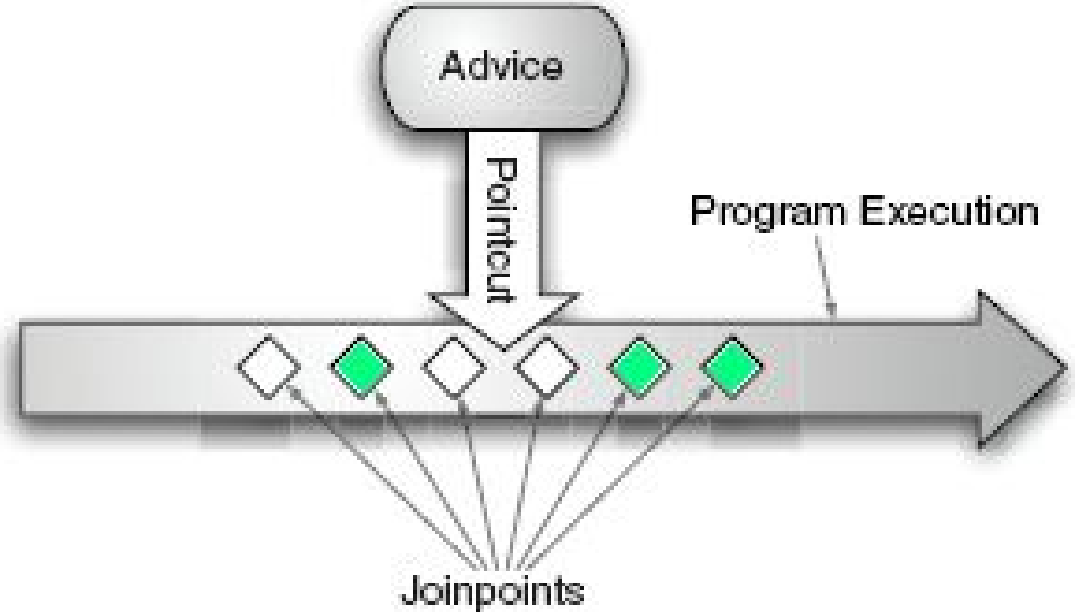
Puntos de Corte (*Pointcut*): capturan colecciones de eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, invocaciones de constructores, y señalización y gestión de excepciones.

Un corte está formado por una parte izquierda(nombre y contexto del corte) y una parte derecha(define los eventos del corte), separadas ambas por dos puntos.

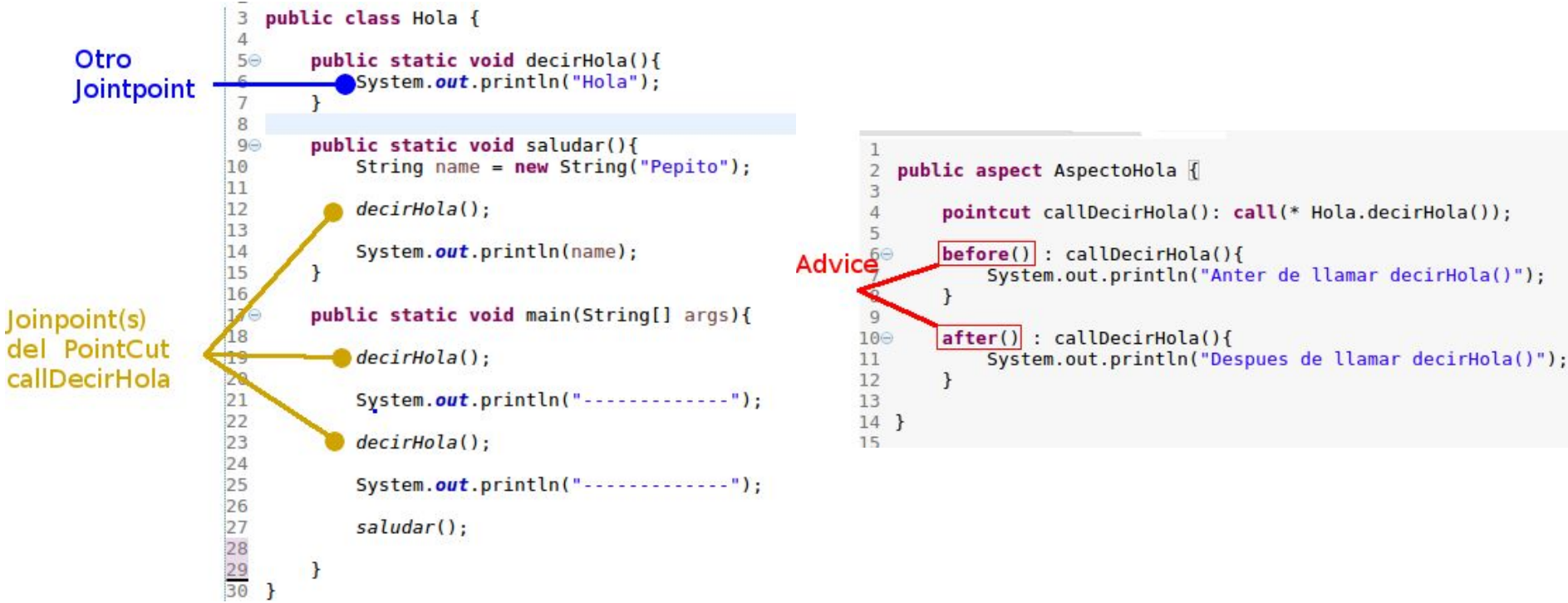
Conceptos básicos

Consejo (*Advice*) es la implementación del aspecto, es decir, contiene el código que implementa la nueva funcionalidad. Se insertan en la aplicación en los puntos de cruce.

- Before: Justo antes de las acciones asociadas con los eventos del corte.
- After: Justo después de las acciones asociadas con los eventos del corte.
- Around: Una parte antes y una parte después de la ejecución del punto de corte. Se debe especificar en qué momentos hacer uso del consejo.
- Catch: Durante la ejecución se captura una excepción del tipo definido en la propia cláusula catch.
- Finally: Justo después de la ejecución de las acciones, incluso aunque se haya producido una excepción durante la misma.


Advice
Pointcut
Program Execution
Joinpoints

Conceptos básicos



Conceptos básicos

Introducción (*Introduction*) permite añadir elementos a clases ya existentes. Entre estos elementos podemos añadir:

- Un nuevo método a la clase.
- Un nuevo constructor.
- Un atributo.
- Varios de los elementos anteriores a la vez.
- Varios de los elementos anteriores en varias clases.

Conceptos básicos

Destinatario (Target)

Captura todos los puntos de enlace donde el objeto objetivo, es una instancia de una clase que coincide con el patrón de clase o con la clase asociada.

Resultante (Proxy)

Corresponde a un objeto creado después de aplicar una notificación al objeto destinatario.

Conceptos básicos

Tejido (*Weaving*) es el proceso de aplicar aspectos a los objetos destinatarios para crear los nuevos objetos resultantes en los especificados puntos de cruce. Este proceso puede ocurrir a lo largo del ciclo de vida del objeto destinatario:

- Aspectos en tiempo de compilación: Los aspectos se tejen al compilar la clase destinataria.
- Aspectos en tiempo de carga: los aspectos se implementan cuando el objeto destinatario es cargado.
- Aspectos en tiempo de ejecución: Los aspectos se tejen en algún momento durante la ejecución del programa.

Entrelazado estático

- Implica modificar el código fuente de una clase insertando sentencias en estos puntos de enlace, es decir, el código del aspecto se introduce en el de la clase
- La principal ventaja de este entrelazado es que evita que el nivel de abstracción que se introduce con los aspectos derive en un impacto negativo en el rendimiento, aunque es bastante difícil identificar los aspecto en el código una vez esté tejido

Entrelazado dinámico

- Un requisito para que se pueda realizar un entrelazado dinámico es que los aspectos existan de forma explícita tanto en tiempo de compilación como en tiempo de ejecución.
- Dada una interfaz de reflexión, el tejedor es capaz de añadir, adaptar y borrar aspectos de forma dinámica durante la ejecución
- El principal inconveniente es el rendimiento y que se utiliza más memoria con la generación de todas las subclases necesarias

Requerimientos

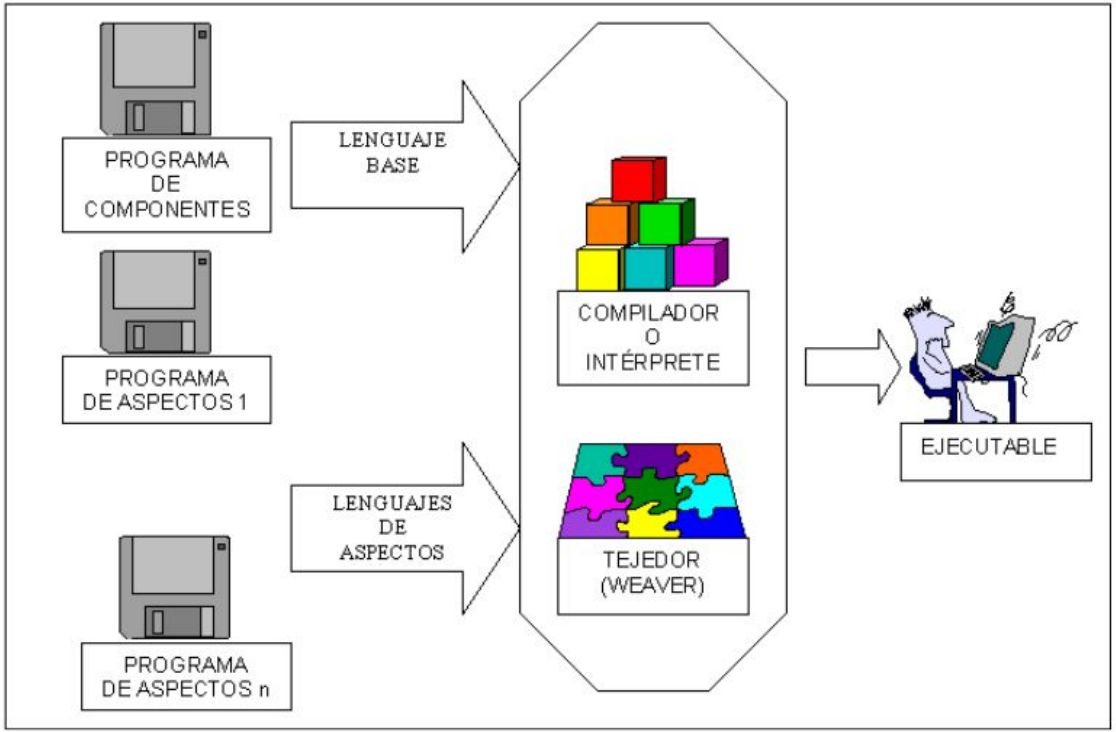


Figura 6. Estructura de una implementación en los lenguajes de aspectos.

Fundamentos de la POA: Puntos de enlace

- Cuando algo sucede, algo ocurrirá

```
class Point {
    private int x, y;

    Point(int x, int y) { this.x = x; this.y = y; }

    void setX(int x) { this.x = x; }
    void setY(int y) { this.y = y; }

    int getX() { return x; }
    int getY() { return y; }
}
```

Fundamentos de la POA: Puntos de enlace

- Por ejemplo, el siguiente pointcut:

```
pointcut setter(): target(Point) &&
                   (call(void setX(int)) ||
                    call(void setY(int)));
```

- Escoge cada llamada a setX y a setY cuando es llamado desde una instancia de Point

Fundamentos de la POA: Puntos de enlace

- Este ejemplo de Pointcut muestra el funcionamiento de un evento que se dispara al recibir una excepción de entrada/salida

```
pointcut ioHandler(): within(MyClass) && handler(IOException);
```

Fundamentos de la POA: Puntos de enlace

- Cuando un método particular se ejecuta
  - execution(void Point.setX(int))
- Cuando se invoca un método
  - call(void Point.setX(int))
- Cuando una excepción es recibida
  - handler(ArrayOutOfBoundsException)
- Cuando el objeto en ejecución es de tipo *SomeType*
  - this(SomeType)
- Cuando el objeto objetivo es de tipo *SomeType*
  - target(SomeType)

Fundamentos de la POA: Consejos

- Un ejemplo de un Pointcut junto con un Consejo que se ejecuta antes del evento descrito

```
pointcut setter(Point p1, int newval): target(p1) && args(newval)
                                       (call(void setX(int) ||
                                        call(void setY(int)));

before(Point p1, int newval): setter(p1, newval) {
    System.out.println("About to set something in " + p1 +
                       " to the new value " + newval);
}
```

Fundamentos de la POA: Consejos

- Ejemplo de un Consejo que se ejecuta sin necesidad de un Pointcut descrito previamente

```
before(Point p1, int newval): target(p1) && args(newval)
                              (call(void setX(int)) ||
                               call(void setY(int))) {
    System.out.println("About to set something in " + p1 +
                       " to the new value " + newval);
}
```

Lenguajes de programación.

En los lenguajes de programación orientados a aspectos se distinguen principalmente 2 enfoques, los lenguajes orientados a aspectos de dominio específico y de propósito general.

Los LOA de dominio específico son diseñados para soportar algún tipo particular de aspectos como por ejemplo concurrencia o sincronización. Algunos de estos necesitan imponer restricciones en el lenguaje base.

Los LOA de propósitos generales son diseñados para soportar cualquier tipo de aspectos. En este tipo de lenguajes no hay restricciones sobre el lenguaje base.

Algunos LOA y sus caracteristicas destacables.

COOL (Coordination language):

Es un lenguaje de dominio específico desarrollado por Xerox. Su principal función es tratar aspectos como sincronismo entre hilos concurrentes. Su lenguaje base es Java pero con una versión modificada. Por ejemplo en este lenguaje se restringe el lenguaje base eliminando los métodos “wait”, “notify”, “notifyAll” y la palabra clave “synchronized” para evitar inconsistencias.

Algunos LOA y sus caracteristicas destacables.

RIDL (Remote Interaction and Data transfers aspect Language):

Es un lenguaje de dominio específico que maneja la transferencia de datos entre diferentes espacios de ejecución.

Un programa en RIDL consiste en un conjunto de módulos llamados “Portales”. Un portal es el encargado de manejar la interacción remota y la transferencia de datos de la clase asociada a este.

Algunos LOA y sus caracteristicas destacables.

MALAJ (Multi Aspect LAnguage for Java):

Es un lenguaje de dominio específico enfocado procesos de sincronización y reubicación.

Al igual que COOL también elimina los métodos “wait”, “notify”, “notifyAll” y la palabra clave “synchronized” para evitar inconsistencias.

Algunos LOA y sus caracteristicas destacables.

AspectJ:

Es un lenguaje de propósito general que extiende Java con una nueva clase de módulos que implementan los aspectos. En AspectJ un aspecto es una clase tal como las clases usadas normalmente en Java, pero con la particularidad que pueden contener unos constructores de corte que no existen en Java.

Estos cortes de AspectJ capturan los eventos en la ejecución de un programa. Estos eventos pueden ser invocaciones de métodos, de constructores y excepciones entre otros. Los cortes no definen acciones sino que describen eventos.

Algunos LOA y sus caracteristicas destacables.

AspectC:

Es un lenguaje de propósito general que extiende C. Es parecido a AspectJ pero sin soporte a programación orientada a objetos.

El código de aspectos interactúa con la funcionalidad básica en los límites de una llamada a una función y puede ejecutarse antes, durante o después de dicha llamada.

mundo.mc

```
int main() {
    printf("mundo");
}
```

hola.acc

```
before(): execution(int main()){
    printf("Hola ");
}

after(): execution(int main()){
    printf("! \n");
}
```

```
>> acc hola.acc mundo.mc
```

Se generan los archivo hola.c y mundo.c

```
>> gcc hola.c mundo.c
```

Algunos LOA y sus caracteristicas destacables.

AspectC++:

Es un lenguaje de propósito general que extiende C++ para soportar el manejo de aspectos.

Al igual que en AspectC los aspectos interactúan con las funcionalidades del programa que en este caso incluye clases. Un aspecto es muy parecido a una clase de C++ pero incluye otras funciones adicionales que definen los aspectos.

hola.h

```
void hola(){
  std::cout<<"Hola "<<std::endl;
}
```

main.cc

```
#include "hola.h"

int main(){
  hola();
  return 0;
}
```

mundo.ah

```
aspect World {

  advice execution("void hola()") : after() {

    std::cout << "Mundo" << std::endl;
  }
};
```

LOA	Lenguaje Base	Tejido	Propósito	Características Salientes
Cool	Java	Estático	Específico	Describe la sincronización de hilos concurrentes. Visibilidad limitada del aspecto
RIDL	Cualquier lenguaje orientado a objetos	Estático	Específico	Modulariza la interacción remota. Visibilidad limitada del aspecto
MALAJ	Java	Dinámico	Especifico	Modulariza los aspectos de sincronización y relocación Su objetivo es eliminar los conflictos entre POA y POO.
AspectJ	Java	Dinámico - Estático	General	Los aspectos son extensiones del concepto de clase.
AspectC	C	Estatico	General	Usado en la implementación orientada a aspectos de sistemas operativos
AspectC++	C++	Estatico	General	Los aspectos son extensiones del concepto de clase.

Ejemplo: Cola circular

```
public class ColaCircular
 {
 private Object[] array;
 private int ptrCola = 0, ptrCabeza = 0;
 private int eltosRellenos = 0;

    public ColaCircular (int capacidad)
     {
          array = new Object [capacidad];
     }

    public void Insertar (Object o)
     {
          array[ptrCola] = o;
          ptrCola = (ptrCola + 1) % array.length;
          eltosRellenos++;
     }

    public Object Extraer ()
     {
      Object obj = array[ptrCabeza];

          array[ptrCabeza] = null;
          ptrCabeza = (ptrCabeza + 1) % array.length;
          eltosRellenos--;

      return obj;
     }
}
```

Ejemplo: Cola circular con sincronización

```
public synchronized void Insertar (Object o) {
   while (eltosRellenos == array.length){
        try {
           wait ();
        }catch (InterruptedException e) {}
   }
   array[ptrCola] = o;
   ptrCola = (ptrCola + 1) % array.length;
   eltosRellenos++;

   notifyAll();
 }
```

```
public synchronized Object Extraer () {
   while (eltosRellenos == 0){
        try {
           wait ();
        }catch (InterruptedException e) {}
   }
   Object obj = array[ptrCabeza];

   array[ptrCabeza] = null;
   ptrCabeza = (ptrCabeza + 1) % array.length;
   eltosRellenos--;

   notifyAll();


   return  obj;
   }
```

Ejemplo: Cola circular con sincronización (COOL)

```
coordinator ColaCircular
 {
     selfex Insertar, Extraer;
     mutex  {Insertar, Extraer};
     cond lleno = false, vacio = true;
     Insertar : requires !lleno;
        on_exit {
           vacio = false;
          if (eltosRellenos == array.length)
               lleno = true;
          }
     Extraer: requires !vacio;
          on_exit {
               lleno = false;
          if (eltosRellenos == 0) vacio = true;
 }
```

Selfex: Mientras se ejecuta un hilo, si aparece otro con la misma petición debe esperar.

Mutex: Mientras un hilo ejecuta uno de los métodos del conjunto, los demás que requieran usar alguno de estos métodos deben esperar.

Ejemplo: Cola circular con sincronización (AspectJ)

```
aspect ColaCirSincro{
 private int eltosRellenos = 0;

 pointcut insertar(ColaCircular c):
   instanceof (c) && receptions(void Insertar(Object));
 pointcut extraer(ColaCircular c):
   instanceof (c) && receptions (Object Extraer());

 before(ColaCircular c):insertar(c) {
         antesInsertar(c);}

 protected synchronized void antesInsertar
                                    (ColaCircular c){
   while (eltosRellenos == c.getCapacidad()) {
     try { wait(); } catch (InterruptedException ex) {};
   }
}
```

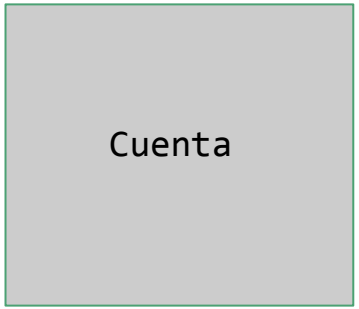
```
after(ColaCircular c):insertar(c) { despuesInsertar();}
protected synchronized void despuesInsertar (){
     eltosRellenos++;
     notifyAll();
}

before(ColaCircular c):extraer(c) {antesExtraer();}
protected synchronized void antesExtraer (){
   while (eltosRellenos == 0) {
     try { wait(); } catch (InterruptedException ex) {};
   }

after(ColaCircular c):extraer(c) {
                despuesExtraer();}

protected synchronized void despuesExtraer (){
     eltosRellenos--;
     notifyAll();
}
```

Ejemplo: Registro de movimientos en una cuenta




```java
public void hacerDeposito(Double cantidad) {
 if(cantidad <= 0) {
     System.out.println("No hay dinero suficiente para hacer el depósito");
 }
 else {
     Date hora = new Date();
     SimpleDateFormat formatoDeFecha = new SimpleDateFormat("yyyy.MM.dd G 'at' HH:mm:ss");
     //Realizamos el depósito
     this.saldo += cantidad;
     //Registramos el movmiento
     System.out.println("Movimiento realizado a las " + formatoDeFecha.format(hora)) ;
 }
}
```

```java
public void hacerExtraccion(Double cantidad) {

 if(this.saldo < cantidad) {
     System.out.println("No hay fondos suficientes para la extracción.");
 }
 else {
     Date hora = new Date();
     SimpleDateFormat formatoDeFecha = new SimpleDateFormat("yyyy.MM.dd G 'at' HH:mm:ss");
     //Extraemos la cantidad del saldode la cuenta.
     this.saldo -= cantidad;
     //Registramos el movmiento
     System.out.println("Movimiento realizado a las " + formatoDeFecha.format(hora)) ;
 }
}
```

Ejemplo: Registro de movimientos en una cuenta

```java
public void hacerTransferencia(Double cantidad, Cuenta cuentaDestino)
{
 if(this.saldo < cantidad) {
     System.out.println("No hay fondos suficientes para la transferencia.");
 }
 else {
     Date hora = new Date();
     SimpleDateFormat formatoDeFecha = new SimpleDateFormat("yyyy.MM.dd G 'at' HH:mm:ss");
     cuentaDestino.hacerDeposito(cantidad);
     this.saldo -= cantidad;
     //Registramos el movmiento
     System.out.println("Movimiento realizado a las " + formatoDeFecha.format(hora)) ;
 }
}
```



```java
public aspect RegistroDeMovimientos {

   after():execution (public * hacer*(..)) {
       Date hora = new Date();
       SimpleDateFormat formatoDeFecha = new SimpleDateFormat("yyyy.MM.dd G 'at' HH:mm:ss");
       System.out.println("Movimiento realizado a las " + formatoDeFecha.format(hora)) ;
   }
}
```

Ejemplo: Registro de movimientos en una cuenta

```
public void hacerDeposito(Double cantidad) {
 if(cantidad <= 0) {
     System.out.println("No hay dinero suficiente para hacer el depósito");
 } else {
     this.saldo += cantidad;
 }
}

public void hacerTransferencia(Double cantidad, Cuenta cuentaDestino) {
 if(this.saldo < cantidad) {
     System.out.println("No hay fondos suficientes para la transferencia.");
 } else {
     cuentaDestino.hacerDeposito(cantidad);
     this.saldo -= cantidad;
 }
}

public void hacerExtraccion(Double cantidad) {
 if(this.saldo < cantidad) {
     System.out.println("No hay fondos suficientes para la extracción.");
 } else {
     this.saldo -= cantidad;
 }
}
```

Ventajas de la POA

- Un código menos enmarañado, más natural y más reducido
- Una mayor facilidad para razonar sobre las materias, ya que están separadas y tienen una dependencia mínima
- Más facilidad para depurar y hacer modificaciones en el código
- Realizar una modificación grande de una materia tenga un impacto mínimo en las otras
- Código más reusable y que se pueda acoplar y desacoplar cuando sea necesario

Desventajas de la POA

- Posibles choques entre el código funcional y el código de aspectos, que nacen de la necesidad de violar el encapsulamiento para implementar los aspectos
- Posibles choques entre los aspectos, El ejemplo clásico es tener dos aspectos que trabajan perfectamente separados, pero juntos presentan un comportamiento anormal
- Posibles choques entre el código de aspectos y los mecanismos del lenguaje

Aplicaciones

- WebSphere Application Server (WAS):

  Es una plataforma que actúa como un servidor de aplicaciones diseñado para configurar, operar e integrar aplicaciones empresariales. WebSphere se distribuye en distintas ediciones donde cada edición soporta diferentes funcionalidades y usan AspectJ internamente para aislar las funcionalidades asociadas a cada edición.

Aplicaciones

- JBoss Application Server (JBoss AS):

  Es un servidor de aplicaciones Java EE. Es un software libre y de código abierto, como está basado en java se puede usar en cualquier sistema operativo donde haya una máquina virtual de java. El núcleo de JBoss está integrado con programacion orientada a aspectos donde se usa principalmente para desplegar servicios tales como seguridad y administración de transacciones.

Aplicaciones

- Oracle TopLink:

  Es un framework para almacenar objetos Java en una base de datos relacional (Object-Relational mapping) o convertir estos objetos en formato XML. TopLink logra altos niveles de persistencia y transparencia usando Spring AOP (Componente del framework Spring usado para aplicar AOP).

Bibliografia

- https://www.lsi.us.es/docs/informes/aopv3.pdf
- http://www.angelfire.com/ri2/aspectos/Tesis/tesis.pdf
- http://es.slideshare.net/wfranck/programacin-orientada-a-aspectos-poa
- https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_aspectos
- https://sites.google.com/a/gapp.msrg.utoronto.ca/aspectc/examples#e1
- https://apt-browse.org/browse/debian/wheezy/main/i386/aspectc++/1:1.1+svn20120529-2/file/usr/share/doc/aspectc++/examples/helloworld
- http://repositorio.utc.edu.ec/bitstream/27000/1843/1/T-UTC-1334.pdf