Programación Orientada a Aspectos

Andrés Giovanny Aldana Wilches
Juan Diego Preciado Mahecha

Contenido

- Introducción
- ¿Qué es la POA?
- Filosofia
- Reseña histórica
- Conceptos Claves
- Lenguajes de programación orientados a aspectos

Contenido

- ¿Cuando usarlo?
- Ejemplos
- Aplicaciones de este paradigma
- Ventajas y Desventajas
- Breve comparación entre POA y POO
- Referencias

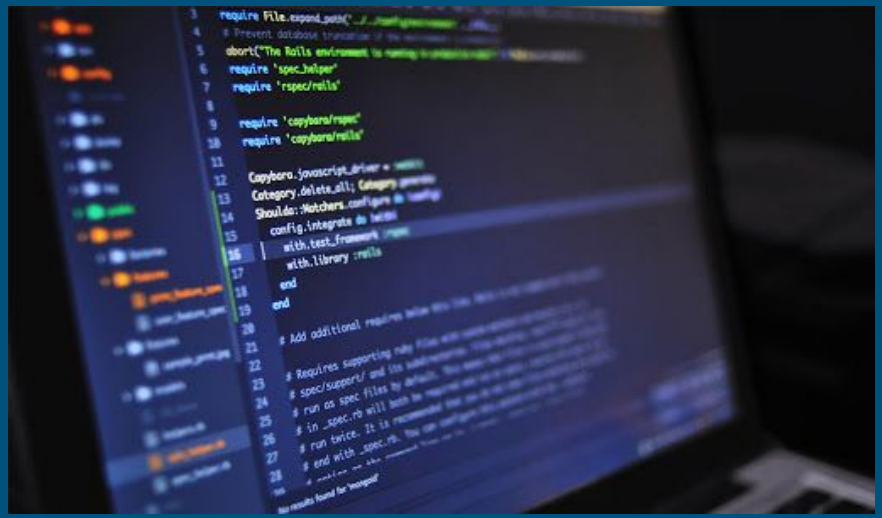
Introducción

Generalmente, el desarrollo de una aplicación involucra varias tareas.

Principales (detalladas) o “servicios comunes” (no detalladas).

Ej. (comunes) Logging, accesos a bases de datos, seguridad

Introducción



Es habitual que servicios comunes deban ser realizados en forma similar pero independiente en diversas partes del código. Se consideran como Incumbencia (concern en inglés).

Entendiendo que el código que la implementa debería incumbir solamente esa tarea.

Introducción

Es deseable la separación de incumbencias en cualquier Desarrollo.

Es una tarea difícil en lenguajes de programación típicos.

Teniendo como consecuencia que las tareas de los servicios comunes queden dispersas dentro del código de las incumbencias principales, incumbencias transversales (crosscutting concerns, en inglés)

Introducción

Separando las incumbencias, se disminuye a complejidad y se gana claridad, adaptabilidad, mantenibilidad, extensibilidad y reusabilidad.

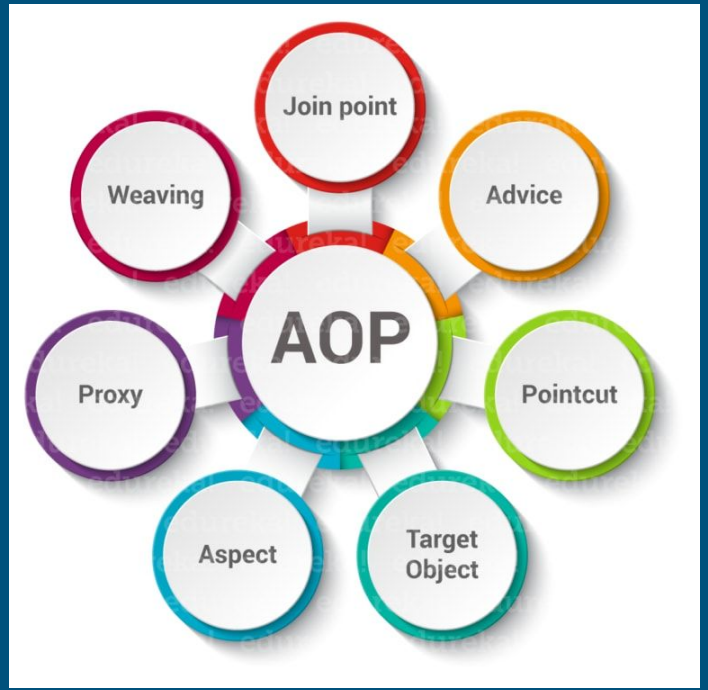
La POA busca resolver este problema de manera eficiente, clara y sistemática.

¿Qué es la POA?

La programación orientada a Aspectos es un paradigma de programación.

Permite una adecuada modularización de las aplicaciones y posibilida una mejor separación de responsabilidades.

Gracias a la POA se pueden encapsular los diferentes conceptos que componen una aplicación en entidades bien definidas, eliminando dependencias entre módulos.

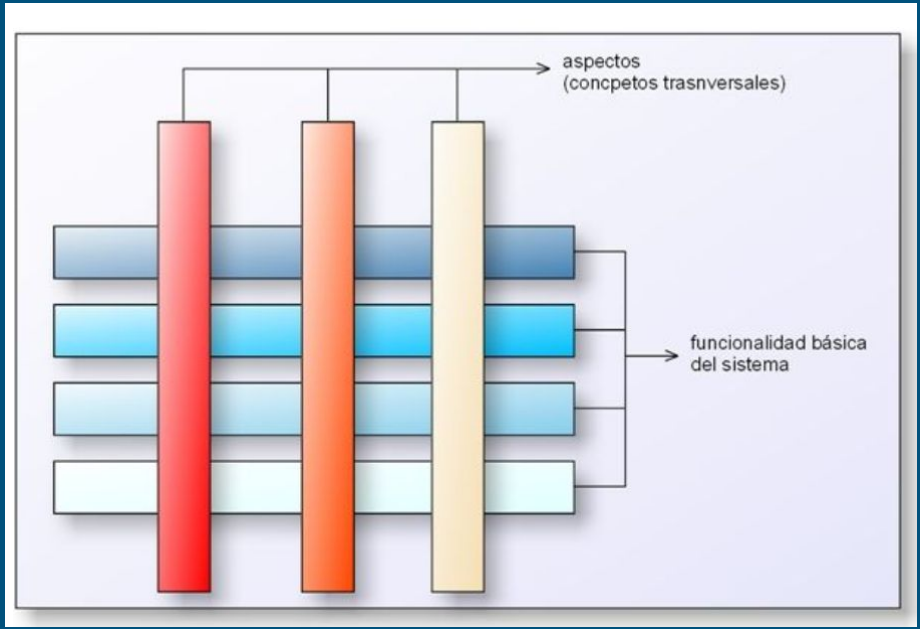


¿Qué es la POA?

De esta forma consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables.

El término POA es usado para referirse a varias tecnologías relacionadas como métodos adaptativos, filtros de composición, programación orientada a sujetos o separación multidimensional de competencias.

Filosofia



Busca tratar las incumbencias transversales de nuestros programas como módulos separados (aspectos) para lograr una correcta separación de las responsabilidades.

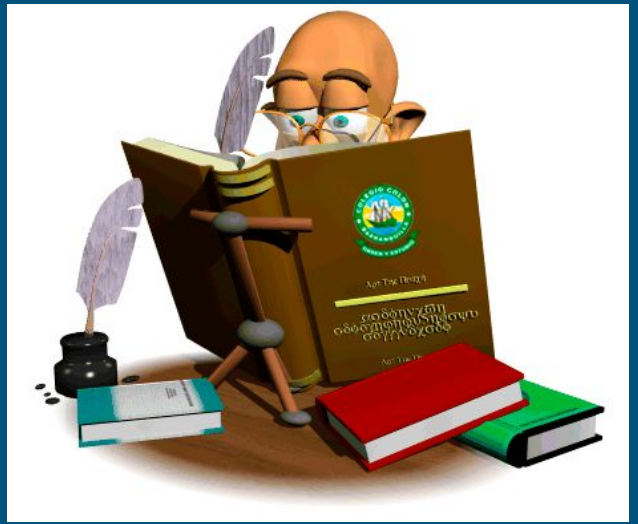
Reseña Histórica

- 1970, se introduce el término “separación de incumbencias” por Edsger W. Dijkstra. Resolución de un problema dado involucra varios aspectos o incumbencias, identificados y analizados en forma independiente.
- Principio de década de 1990, Programación adaptativa, concepto predecesor de la POA, introducido por el grupo Demeter, Karl Lieberherr.

Reseña Histórica

- 1997, los primeros conceptos de POA son introducidos por Gregor Kiezales y su grupo

- Actualmente se tienen varias implementaciones de lenguajes orientados a aspectos



Conceptos Claves

- **ASPECTO (Aspect):** Es una funcionalidad transversal (cross-cutting) que se va a implementar de forma modular y separada del resto del sistema. Ej. Logging
- **Punto de Cruce o de Unión (Join Point):** Es un punto de ejecución dentro del sistema donde un aspecto puede ser conectado, como llamada a un método, lanzamiento de una excepción o modificación de un campo.

Conceptos Claves

- **Consejo (Advice):** Implementación del aspecto. Contiene el código que implementa la nueva funcionalidad. Se insertan en los puntos de corte.
- **Punto de Corte (Pointcut):** define consejos que se aplicarán cada punto de cruce. Se especifica mediante expresiones regulares o mediante patrones de nombres (clases, métodos o campos) e incluso dinámicamente en tiempo de ejecución el valor de ciertos parámetros.

Conceptos Claves

- **Introducción (Introduction):** Permite añadir métodos o atributos a clases ya existentes.
- **Destinatario (Target):** clase aconsejada, la clase que es objeto de un consejo. Sin POA esta clase debería contener su lógica, además de la lógica del aspecto.
- **Resultante (Proxy):** es el objeto creado después de aplicar el consejo al objeto destinatario. El resto de la aplicación únicamente tendrá que soportar al objeto destinatario (pre-AOP) y no al resultante (post-AOP).

Conceptos Claves

- **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, Aspectos en tiempo de Carga, Aspectos en tiempo de ejecución.

Lenguajes de programación orientados a Aspectos

COOL

LOA de dominio específico enfocado en tratar los aspectos de sincronismo entre hilos concurrentes

RIDL

LOA de dominio específico que maneja la transferencia de datos entre diferentes espacios de ejecución

MALAJ

LOA de dominio específico enfocado en la sincronización y reubicación.

AspectC

LOA de propósito general que extiende de C. Bastante similar a AspectJ pero sin el soporte de de POO.

AspectC++

LOA de propósito general que entiende de C++. Los aspectos en este lenguaje son sintácticamente similares a una clase

AspectJ

LOA de propósito general que extiende de Java. Los aspectos cortan clases, interfaces y otros aspectos.

¿Cuando usarlo?

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

Ejemplos - POA orientada a seguridad

El objetivo de los autores era hacer validaciones al método malloc e implementar un método seguro de generación de números aleatorios

```
aspect secure_random {
  int secure_rand(void) {
    /**
     * Secure call to random defined here. Any other accompanying
     * functions can be defined in the same scope as this function.
     */
    return 0;
  }

  void support_function(void) {
  }

  funcCall<int rand(void)> {
    replace {
      secure_rand();
    }
  }
}

aspect malloc_check {
  funcCall<void * malloc(size_t x)> {
    after {
      if(__RETVAL__ == NULL) {
        printf("malloc(%i) failed in function %s(). exiting...\n",
               x, __FUNCTION__);
        exit(-1);
      }
    }
  }
}
```

Ejemplos - POA orientada a seguridad

```
int init_msg(char *[], int);

int main(void) {
  char * msg[4];
  int i;

  init_msg(msg, 4);
  i = rand() % 4;
  printf("%s",msg[i]);
  return 0;
}


int init_msg(char * arg[], int size) {
  int i;
  for(i=0 ; i<size ; ++i) {
    arg[i] = (char *) malloc(20);
    sprintf(arg[i], "Message Number %d\n", i);
  }
  return 0;
}
```

Así el cambio de su comportamiento será a causa de las implementaciones en los aspectos.

Ejemplos - Cálculo de PI usando hilos

El objetivo es calcular el valor de pi utilizando hilos, donde cada hilo se encargará de calcular un término de acuerdo con el siguiente método:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \frac{\pi}{4}.$$

Aquí, los aspectos serán los encargados de garantizar la sincronización entre los hilos.

Ejemplos - Cálculo de PI usando hilos

Cada hilo selecciona un término y calcula su valor para aproximar pi, luego agrega su aproximación al resultado global.

```java
@Override
public void run() {
    int currentTerm = store.getCurrentIteration();

    while (currentTerm < store.getTargetIterations()) {
        double factor = currentTerm * 2 + 1;
        if (currentTerm % 2 != 0) {
            factor *= -1;
        }

        store.setCurrentPiValue(1 / factor);
        currentTerm = store.getCurrentIteration();
    }
    this.finished = true;
```

Ejemplos - Cálculo de PI usando hilos

Utilizando semáforos, el aspecto los bloquea antes de ejecutar los métodos y los libera después de ejecutarlos

```
public aspect Synchronizer {

    private final Semaphore currentTermSemaphore = new Semaphore( permits: 1,  fair: true);
    private final Semaphore calculatedPiValueSemaphore = new Semaphore( permits: 1,  fair: true);

    pointcut currentIterationCall():
            call(int PiCalculator.getCurrentIteration());

    before(): currentIterationCall() {
        try {
            currentTermSemaphore.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    after(): currentIterationCall() {
        currentTermSemaphore.release();
    }

    pointcut calculatedPiValueCall():
            call(void PiCalculator.setCurrentPiValue(double));

    before(): calculatedPiValueCall() {
        try {
            calculatedPiValueSemaphore.acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    after(): calculatedPiValueCall() {
        calculatedPiValueSemaphore.release();
    }
}
```

Ejemplos - Banco

El objetivo es permitir operaciones de depósitos, retiros y transferencias entre cuentas.

```java
public double deposit(double amount) {
    this.balance += amount;
    return this.balance;
}

public double withdraw(double amount) {
    this.balance -= amount;
    return this.balance;
}

public double transfer(Account receiver, double amount) {
    receiver.deposit(amount);
    this.balance -= amount;
    return this.balance;
}
```

Ejemplos - Banco

El primer aspecto está encargado de realizar logs después de la ejecución de cada operación

```
public aspect Logger {
    static PrintStream logStream = System.out;

    public static void log(String message) {
        Date now = new Date();
        logStream.println(now.toString() + " " + message);
    }

    pointcut deposit(Account acc, double value):
            call(double Account.deposit(double)) && target(acc) && args(value);

    after(Account acc, double value): deposit(acc, value) {
        StringBuilder builder = new StringBuilder();
        builder.append("Deposit to account: ").append(acc.getId());
        builder.append(", amount: ").append(value);

        log(builder.toString());
    }

    pointcut withdraw(Account acc, double value):
            call(double Account.withdraw(double)) && target(acc) && args(value);

    after(Account acc, double value): withdraw(acc, value) {
        StringBuilder builder = new StringBuilder();
        builder.append("Withdraw from account: ").append(acc.getId());
        builder.append(", amount: ").append(value);

        log(builder.toString());
    }
```

Ejemplos - Banco

El segundo aspecto está encargado de realizar validaciones antes de ejecutar los operaciones.

```
public aspect Validator {
    pointcut InputAction(double amount):
            call(double Account.deposit(double)) && args(amount);

    before(double amount): InputAction(amount) {
        if (amount < 0) {
            throw new NegativeAmountException();
        }
    }

    pointcut OutputAction(Account acc, double amount):
            (
                call(double Account.withdraw(double)) ||
                call(double Account.transfer(Account, double))
            ) && target(acc) && args(amount);

    before(Account acc, double amount): OutputAction(acc, amount) {
        if (acc.getBalance() - amount < 0) {
            throw new NotEnoughFounds(acc.getBalance(), amount);
        }
    }
}
```

Aplicaciones de este paradigma

- **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

Aplicaciones de este paradigma

- **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

Ventajas

- Implementación modularizada
- Mayor evolucionabilidad
- Reusabilidad
- Dividir y conquistar
- N-Dimensiones
- Superar problemas causados por código mezclado y código diseminado
- Hace enfasis en el principio del mínimo acoplamiento y máxima cohesión

Desventajas

- Posibles choques entre aspectos
- Posibles choques entre el código de aspectos y los mecanismos del lenguaje
- Posible choque entre código funcional y el código de aspectos.

Breve comparación entre POA y POO

POO: modela los conceptos comunes de un sistema.

POA: Trata los conceptos entrecruzados como elementos de primera clase.

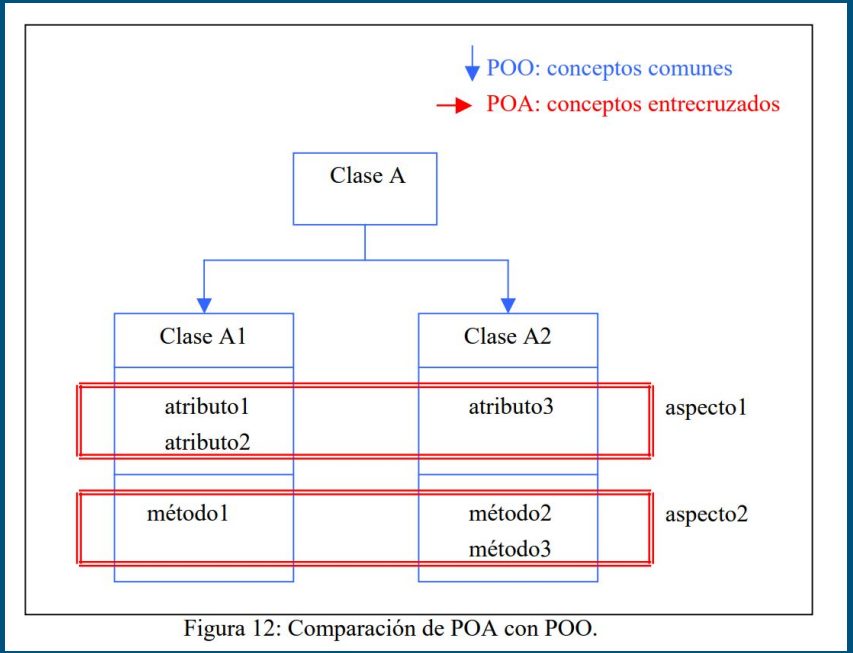


Figura 12: Comparación de POA con POO.

Referencias

[1] A. Fernando y C. Bernardo. Programación orientada a aspectos. Análisis del paradigma. [Online] Available: https://www.angelfire.com/ri2/aspectos/Tesis/tesis.pdf [Accessed: 18-Ene-2021]

[2] Programación orientada a aspectos. [Online]. Available: http://ferestrepoca.github.io/paradigmas-de-programacion/poa/poa_teoria/index.html [Accessed: 18-Enero-2021]

[3] J. José. Programación Orientada a Aspectos. [Online]. Available: http://iie.fing.edu.uy/~josej/docs/Programacion%20Orientada%20Aspectos%20-%20Jose%20Joskowicz.pdf [Accessed: 18-Enero-2021]

[4] R. Antonia. Visión General de la Programación Orientada a Aspectos. [Online]. Available: http://www.lsi.us.es/docs/informes/aopv3.pdf [Accessed: 19-Enero-2021]

[5] Programación orientada a Aspectos. [Online]. Available: https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_aspectos [Accessed: 18-Enero-2021]

[6] Programación Orientada a Aspectos [Online]. Available: https://sites.google.com/site/programacionhm/conceptos/aop [Accessed: 19-Enero-2021]