**Programación orientada a aspectos**

La **Programación Orientada a Aspectos** o POA (en [inglés](https://es.wikipedia.org/wiki/Idioma_ingl%C3%A9s): ***aspect-oriented programming***) es un [paradigma de programación](https://es.wikipedia.org/wiki/Paradigma_de_programaci%C3%B3n) relativamente reciente cuya intención es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de responsabilidades (Obligación o correspondencia de hacer algo).

Gracias a la POA se pueden encapsular los diferentes conceptos que componen una aplicación en entidades bien definidas, eliminando las dependencias entre cada uno de los módulos. De esta forma se consigue razonar mejor sobre los conceptos, se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reusables. Varias tecnologías con nombres diferentes se encaminan a la consecución de los mismos objetivos y así, el término POA es usado para referirse a varias tecnologías relacionadas como los [métodos adaptativos](https://es.wikipedia.org/w/index.php?title=M%C3%A9todo_adaptativo&action=edit&redlink=1), los [filtros de composición](https://es.wikipedia.org/w/index.php?title=Filtro_de_composici%C3%B3n&action=edit&redlink=1), la [programación orientada a sujetos](https://es.wikipedia.org/w/index.php?title=Programaci%C3%B3n_orientada_a_sujetos&action=edit&redlink=1) o la [separación multidimensional de competencias](https://es.wikipedia.org/w/index.php?title=Separaci%C3%B3n_multidimensional_de_competencias&action=edit&redlink=1).

* **Aspecto** (en inglés *Aspect*) es una funcionalidad transversal (*cross-cutting*) que se va a implementar de forma modular y separada del resto del sistema. El ejemplo más común y simple de un aspecto es el logging (registro de sucesos) dentro del sistema, ya que necesariamente afecta a todas las partes del sistema que generan un suceso.
* **Punto de Cruce o de Unión** (en inglés *Join point*) es un punto de ejecución dentro del sistema donde un aspecto puede ser conectado, como una llamada a un método, el lanzamiento de una excepción o la modificación de un campo. El código del aspecto será insertado en el flujo de ejecución de la aplicación para añadir su funcionalidad.
* **Consejo** (en inglés *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.
* **Puntos de Corte** (en inglés *Pointcut*) define los Consejos que se aplicarán a cada Punto de Cruce. Se especifica mediante Expresiones Regulares o mediante patrones de nombres (de clases, métodos o campos), e incluso dinámicamente en tiempo de ejecución según el valor de ciertos parámetros.
* **Introducción** (en inglés *Introduction*) permite añadir métodos o atributos a clases ya existentes. Un ejemplo en el que resultaría útil es la creación de un Consejo de Auditoría que mantenga la fecha de la última modificación de un objeto, mediante una variable y un método setUltimaModificacion(fecha), que podrían ser introducidos en todas las clases (o sólo en algunas) para proporcionarles esta nueva funcionalidad.
* **Destinatario** (en inglés *Target*) es la clase aconsejada, la clase que es objeto de un consejo. Sin AOP, esta clase debería contener su lógica, además de la lógica del aspecto.
* **Resultante** (en inglés *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 Objeto Resultante (post-AOP).
* **Tejido** (en inglés *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, que necesita un compilador especial.
  + Aspectos en Tiempo de Carga, los Aspectos se implementan cuando el Objeto Destinatario es cargado. Requiere un ClassLoader especial.
  + Aspectos en Tiempo de Ejecución.

Muchas veces nos encontramos, a la hora de [programar](https://es.wikipedia.org/wiki/Programaci%C3%B3n), con problemas que no podemos resolver de una manera adecuada con las técnicas habituales usadas en la [programación imperativa](https://es.wikipedia.org/wiki/Programaci%C3%B3n_imperativa) o en la [programación orientada a objetos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos). Con éstas, nos vemos forzados a tomar decisiones de diseño que repercuten de manera importante en el desarrollo de la aplicación y que nos alejan con frecuencia de otras posibilidades.

A menudo, hace falta escribir líneas de código que están distribuidas por toda o gran parte de la aplicación, para definir la lógica de cierta propiedad o comportamiento del sistema, con las consecuentes dificultades de mantenimiento y desarrollo. En [inglés](https://es.wikipedia.org/wiki/Idioma_ingl%C3%A9s) este problema se conoce como [*scattered code*](https://es.wikipedia.org/w/index.php?title=Scattered_code&action=edit&redlink=1), que podríamos traducir como [código disperso](https://es.wikipedia.org/w/index.php?title=C%C3%B3digo_disperso&action=edit&redlink=1). Otro problema que puede aparecer, es que un mismo módulo implemente múltiples comportamientos o aspectos del sistema de forma simultánea. En inglés este problema se conoce como [*tangled code*](https://es.wikipedia.org/w/index.php?title=Tangled_code&action=edit&redlink=1), que podríamos traducir como [código enmarañado](https://es.wikipedia.org/w/index.php?title=C%C3%B3digo_enmara%C3%B1ado&action=edit&redlink=1). El hecho es que hay ciertas decisiones de diseño que son difíciles de capturar, debido a que determinados problemas no se pueden encapsular claramente de igual forma que los que habitualmente se resuelven con funciones u objetos.