Aspect oriented programming (AOP)

In computing, aspect-oriented programming **(AOP)** is a patented[1] programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does so by adding additional behavior to existing code (an advice) without modifying the code itself, instead separately specifying which code is modified via a “pointcut” specification, such as “log all function calls when the function’s name begins with ‘set'”. This allows behaviors that are not central to the business logic (such as logging) to be added to a program without cluttering the code core to the functionality. AOP forms a basis for aspect-oriented software development.

AOP includes programming methods and tools that support the modularization of concerns at the level of the source code, while “aspect-oriented software development” refers to a whole engineering discipline.

Aspect-oriented programming entails breaking down program logic into distinct parts (so-called concerns, cohesive areas of functionality). Nearly all programming paradigms support some level of grouping and encapsulation of concerns into separate, independent entities by providing abstractions (e.g., functions, procedures, modules, classes, methods) that can be used for implementing, abstracting and composing these concerns. Some concerns “cut across” multiple abstractions in a program, and defy these forms of implementation. These concerns are called cross-cutting concerns.

Logging exemplifies a crosscutting concern because a logging strategy necessarily affects every logged part of the system. Logging thereby crosscuts all logged classes and methods.

All AOP implementations have some crosscutting expressions that encapsulate each concern in one place. The difference between implementations lies in the power, safety, and usability of the constructs provided. For example, interceptors that specify the methods to intercept express a limited form of crosscutting, without much support for type-safety or debugging. AspectJ has a number of such expressions and encapsulates them in a special class, an aspect. For example, an aspect can alter the behavior of the base code (the non-aspect part of a program) by applying advice (additional behavior) at various join points (points in a program) specified in a quantification or query called a pointcut (that detects whether a given join point matches). An aspect can also make binary-compatible structural changes to other classes, like adding members or parents.

```
void transfer(Account fromAcc, Account toAcc, int amount, User
user,
    Logger logger) throws Exception {
  logger.info("Transferring money…");

  if (!isUserAuthorised(user, fromAcc)) {
    logger.info("User has no permission.");
    throw new UnauthorisedUserException();
  }

  if (fromAcc.getBalance() < amount) {
    logger.info("Insufficient funds.");
    throw new InsufficientFundsException();
  }

  fromAcc.withdraw(amount);
  toAcc.deposit(amount);

  database.commitChanges();  // Atomic operation.

  logger.info("Transaction successful.");
}
```