Aspect-Oriented Programming

Separation of Concerns

- Breaking down a program into distinct parts that overlap in functionality as little as possible.
- All programming methodologies support some separation and of concerns, e.g. procedures, packages, classes, and methods.
- Other examples include MVC architecture, CSS style sheets with HTML, etc.

Cross-cutting Concerns

- Secondary requirements that are shared between program units.
  - For example: we may want to add logging to classes within the data-access layer and also to classes in the UI layer whenever a thread enters or exits a method. Even though the primary functionality of each class is very different, the code needed to perform the secondary functionality is often identical.
- Normal forms of encapsulation are not adequate to handle crosscutting concerns (e.g. security, logging) that cut across multiple modules in a program.

Cohesion

- When one concern (e.g. logging or security) is spread over a number of components (e.g., classes and methods) there is a loss of cohesion.
- Components end up tangled with multiple concerns (e.g., account processing, logging, and security). Changing one component entails understanding all the tangled concerns.
- Code to implement a single concern is scattered among many components, making it harder to understand and maintain.

Aspects

- Cross-cutting concerns do not get properly encapsulated in their own modules. This increases the system complexity and makes evolution considerably more difficult.
- AOP attempts to solve this problem by allowing the programmer to express cross-cutting concerns in separate stand-alone modules called aspects.

Advice

- The additional code that you want to apply to your existing model.
  - For example, a security module could include advice that performs a security check before accessing a bank account.
- Advice code is joined to specified points in the program called join points.

Join Point Models

- A JPM defines three things:
  - **Join points** that specify where in the existing code the advice can run. They should be stable across minor program changes.
  - **Point-cuts** that determine whether a given join point matches.
  - **Advice** that specifies the code to run at a join point. Advice can run before, after, or around join points.

AspectJ Join Points

- method or constructor call or execution
- initialization of a class or object
- field read and write access
- exception handlers, etc.
- **Not**: loops, super calls, throws clauses, blocks, etc.

AspectJ Pointcut Designators

- **Kinded PCDs** match a particular kind of join point (e.g., method execution) and take as input a Java-like signature:
  - execution(* set*(*))
    matches a method execution join point, if the method name starts with "set" and there is exactly one argument of any type.
- **Dynamic PCDs** check runtime types and bind variables, e.g:
  - this(Point)
    matches when the currently-executing object is an instance of class Point.
- **Scope PCDs** limit the scope of the join point, e.g:
  - within(com.company.*)
- Pointcuts can be composed and named for reuse.
  - pointcut set() : execution(* set*(*)) && this(Point) &&
        within(com.company.*);

AspectJ Advice

- Advice specifies the code to run before, after, or around join points that match a specified pointcut.
- Advice is invoked automatically by the AOP runtime when a join point matches the pointcut. For example:

```
after() : set() {
   Display.update();
}
```

- If a join point matches the set() pointcut, the code "Display.update()" will run after the join point completes.

Disadvantages of AOP

- Tool support is currently weak
- Mistakes in expressing crosscutting can lead to widespread program failure.
- Changes in the join points of a program, e.g. renaming or moving methods, that were not anticipated by the aspect writer may cause failures.
- Security can be broken by using AOP to inject additional code at the appropriate places.