Aspect Oriented Paradigm

Ming-Hwa Wang, Ph.D.
Department of Computer Engineering
Santa Clara University

Terminology

- *Concern*: functionality or requirement necessary in a system. In a typical system, a large number of concerns need to be addressed in order for the system to accomplish its goals.
- *Crosscutting*: when a requirement for the system is met by placing code into objects through the system – but the code doesn't directly relate to the functionality defined for those objects, e.g., logger.

Disadvantages of object-oriented programming (OOP)

- The mixing of concerns leads to a condition called *code scattering* (the code necessary to fulfill one concern is spread over the classes needed to fulfill another concern) or *code tangling* (using a single method or class to implement multiple concerns).
- Those cause the following problems: classes that are difficult to change, code that can't be reused, code that's impossible to trace.

Aspect-oriented programming (AOP):

- Two fundamental goals: allow for separation of concerns as appropriate for a host language, and provide a mechanism for the description of concerns that crosscut other components.
- Two languages (can be same or different languages):
  - The *component language*: code the requirements or concerns into units of code
  - The *aspect language*: code the secondary or support requirements/aspects
  - The compiler weaves the aspect code into the component code to create a functioning system, there are 2 types of weaving:
    - *Static weaving* (or compile-time weaving): the aspect code is converted to the primary language and inserted directly into the primary application code (e.g., AspectJ 1.0.x), drawback including the inability to dynamically change the aspect used against the primary code, and need to have the source code available for all aspects (so JAR can't be used)
    - *Dynamical weaving* (link-time - e.g., AspectJ 1.1.x, load-time, or run-time weaving): place hooks in the methods/constructor of the primary code, when the hooks are executed, a modified runtime system determines whether any aspects need to execute. It is more complicated than static weaving.
- Advantages of AOP: creating clean and well-encapsulated objects without extraneous functionality, less tangled code, shorter code, easier application maintenance and evolution, easier to debug/refactor/modify, more reusable, etc.

Major components of an AOP language:

- *Joint point*: a predictable point in the execution of an application, a well-defined location within the primary code where a concern will crosscut the application, it can be method calls, constructor invocation, exception handlers, etc.
- *Pointcut*: a structure designed to identify and select join points, and tells the aspect language when the joint point should match (only when it's part of the a method call), the match can be exact match, or wildcard match
- *Advice*: code to be executed when a join point is reached, it can execute at 3 different places when a join point is reached: before, around and after
- *Inter-type declarations*: mechanism to add new functionality by defining new attributes and methods to previously established classes
- *Aspect*: encapsulates join points, pointcuts, advice, and inter-type declarations

AOP Considered Harmful

- Obliviousness (i.e., not visible) of application
- Non-certainty of application – the actual matching state in general is not known
- Dynamic dispatch similar to goto statement (at least goto is visible) and Dijkstra considered goto is harmful