



## SS2018 – Model-driven Software Development in Technical Spaces

# Aspect-Oriented Programming (AOP)

Professor: Prof. Dr. Uwe Aßmann

Tutor: Dr.-Ing. Thomas Kühn

### Task 1 Aspect-Oriented Programming (AOP)

The goal of *aspect-oriented programming* (AOP) is to elucidate design decisions that cross-cut the system's basic functionality. According to Kiczales et al. [3] *aspect-oriented programming* (AOP) “*makes it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code.*”

- a) Describe the terms *scattering* and *tangling*. Why are both effects bad?

**Solution:**

- **Scattering:** Code handling one concern is split and spread throughout different modules/classes.
- **Tangling:** Code in one module/class handles multiple concerns. Both effects indicate unmanaged redundancy of code as well as bad modularity.

- b) What does *cross-cutting* mean wrt. AOP?

**Solution:** AOP acknowledges that complex systems inherently contain *cross-cutting concerns* that should be specified separately. In detail, the *Cross-cutting concerns* have a clear purpose and regular interaction points. Thus, AOP distinguishes *core* components and *aspect* components.

- c) Describe the terms *Pointcut*, *Join Point*, *Advice* and *Weaving*. Give examples for each.

**Solution:**

- **Pointcut:** Declares a set of logically related join points that must be matched to activate an advice, e.g., intercept all calls to `print` or `println`.
- **Join Point:** Specifies the points where an open core component can be extended.
- **Advice:** Denotes a code fragment with behavior that should become active, whenever a dynamic join point is encountered.
- **Weaving:** The technology for composing aspects and core components.

d) Where is the difference between *static* and *dynamic* join points?

**Solution:**

- **static join points:** are code positions, hooks, open for extension, e.g., class definitions, method definitions.
- **dynamic join points:** are some points in the execution trace of an application, open for extension, e.g., method entry, method exit.

e) Where is the difference between *static* and *dynamic* weaving?

**Solution:**

- **Static aspect weaving:** Performs the composition of the core components and aspects at compile-time or load-time. Advice code is statically bound to join points.
- **Dynamic aspect weaving:** Only prepares the join points at load-time, whereas the aspects are dynamically linked to the join points at runtime. Consequently, new core components and aspects can be dynamically loaded, woven and activated.

f) Describe the *component model*, *composition technique* and *composition language* of AspectJ.

**Solution:**

- **Component model:** core components and aspect components containing advices
- **Composition technique:** Aspect weaving, i.e., extending join points of core components with advices.
- **Composition language:** Pointcut description language

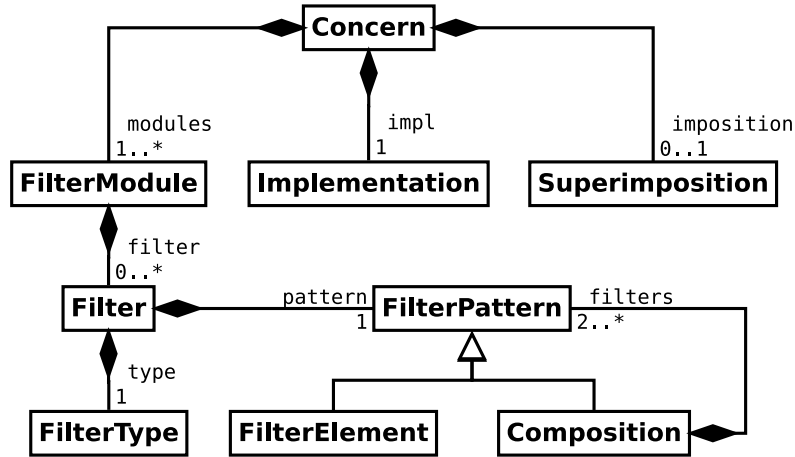


Figure 1: Metamodel of the Composition Filters.

## Task 2 Composition Filters

In contrast to AOP, one of the main goals of *Composition Filters* is to “*extend existing (object-oriented) programming models in a modular way, instead of replacing or adapting them*” [1].

- a) What are the basic concepts of *Composition Filters*? How are they related? Draw an analysis class diagram.

**Solution:** As shown in Figure 1 Concerns, filter modules, filters, filter types, filter patterns, filter elements, filter compositions, superimpositions.

- b) In what ways can *Filters* adapt messages before passing them one to the next *Filter*?

**Solution:**

Ingoing and outgoing messages can be

- *accepted* (passed to inner object),
- *dispatched* (passed to external objects),
- *delayed* until a condition becomes true,
- *discarded* invocation is rejected (with an exception),
- *substituted* replaced with another message, and
- *transformed* to the metaclass `Message`.

- c) Can the `java.util.stream` library of Java be considered a Composition Filter architecture?

**Solution:** Partially, as Java streams can be subject to multiple input and output filters by composing the respective method invocations. Moreover, functions, such as `parallel()`, permit adding a concern into an existing stream. Yet, both streams

and filters only emit streams. In contrast, Composition Filters permits filtering messages to and from object, regardless of its type. Thus, while Java streams support input filters, they limit output filters to always emit another stream.

- d) Describe the *component model*, *composition technique* and *composition language* of Composition Filters [1]?

**Solution:**

- **Component model:** Components and Filters with methods as ports.
- **Composition technique:** Static composition employing Multiple Inheritance or dynamic composition with decorators. Superimposition allows adaptation of filters and components.
- **Composition language:** Simple dispatch composition for input and output filters.

## Task 3 Programming with AspectJ

In this task, you will learn how aspect-oriented software is implemented. You are tasked to write simple aspects for a Java application using **AspectJ**, an aspect-oriented extension to Java. Luckily, there exists direct *Eclipse* support by means of the *AspectJ Development Tools* (AJDT) [2].

As our example we consider a tiny library that represents the structure of houses. The library offers an interface to construct houses. In the current version any combination of **ComplexHouseParts** is possible (e.g., a level can contain hallways, which can contain levels etc.). Furthermore, rooms can either be clean or not clean. They can be cleaned by calling the `setClean(boolean clean)` method. House parts can be entered and visited. Visiting a house part means entering the part itself and visiting all the subparts. However, as we want to use our library for any kind of domain, we want to exclude the consistency checking code from the base implementation. Rather than, we want to use customer-specific aspects, which are woven into the base implementation. Use **AspectJ** to implement the following features.

- a) Download and install **AspectJ** and the corresponding example.
  1. Install the *AspectJ Development Tools* (AJDT).<sup>1</sup>
  2. Download the source of the core project from the CBSE website.
  3. Edit the core project to solve the task (by adding aspects).
- b) Define an Aspect that logs (prints to the console) when the program starts and stops running.
- c) Define an Aspect, which restricts that levels cannot contain other levels.
- d) Define an Aspect, which marks a bathroom as not clean, when it is entered.
- e) Define an Aspect, which prints a warning when a room is entered which is not cleaned.
- f) Define an Aspect, which prohibits (e.g., throws an exception) any person entering a bathroom which is not cleaned.
- g) In the initialization Script (`buildSimpleHouse()`), a house with a private room is created. Define an aspects, which prohibits any other person that the owner of the house, to enter the private room.

Hand in your solution as archive `*.zip` before the next exercise.

**Solution:** A possible solution can be downloaded from the website.

---

<sup>1</sup>Choose the right update site wrt. your Eclipse version <http://www.eclipse.org/ajdt/downloads/index.php>

## Additional Information

- [AspectJ Development Tools](#)<sup>2</sup>
- [AJDT Documentation](#)<sup>3</sup>
- [Tutorials to getting started](#)<sup>4</sup>
- [More elaborate tutorial](#)<sup>5</sup>

## References

- [1] Lodewijk Bergmans and Mehmet Aksit. Principles and design rationale of composition filters. *Aspect-Oriented Software Development*, pages 63–95, 2004.
- [2] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse aspectj: aspect-oriented programming with aspectj and the eclipse aspectj development tools*. Addison-Wesley Professional, 2004.
- [3] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

---

<sup>2</sup><https://www.eclipse.org/ajdt/>

<sup>3</sup><https://eclipse.org/aspectj/doc/released/proguide/starting.html>

<sup>4</sup><https://eclipse.org/ajdt/demos/>

<sup>5</sup><https://o7planning.org/en/10257/java-aspect-oriented-programming-tutorial-with-aspectj>