**Week 2: Annotated Bibliography**

Nate Bachmeier

May 12th, 2019

TIM-8110: Programming Languages and Algorithms

Northcentral University

# Topic 1: Aspect Oriented Programming

## Kiczales, G; et. al. Aspect Oriented Programming (1997)

Modern software is written as a collection of objects that represent the various components of the system. This approach leads to modular designs that are loosely coupled and can be upgraded relatively easily.

However, there are ‘aspects’ of the system which are difficult to prevent tight coupling, an example might be logging. How can the logging framework be decoupled when nearly every method must call it? Similarly, challenges can be seen with object caching, unrolling loops, security assertions, and retry policy to name a few. To address these challenges AOP identifies these ‘cross-cutting concerns’ and attempts to centralize them.

Consider an image processing system that needs to apply several filters to a bitmap. Each filter must enumerate the pixels and perform some action.

If every filter runs sequentially then the program will require width\*height\*filters fetches. By encapsulating the fetching into a centralized dispatcher and broadcasting to the filters, then the program can be reduced to width\*height fetches.

This introduces its own set of challenges as our dispatch code can become too tightly coupled with the filter implementation. AOP addresses this by pipelining the system code either at compile or runtime.

The pipelining adds ‘joinpoint’ which are possible injection points throughout the code base. Examples could include before a method is called or after an exception is thrown. Next an ‘advice’ is represented as a callback behavior and bound to the joinpoint as a ‘pointcut’. This system provides a mechanism to push the complexity of ‘weaving’ functionality down to the compiler and away from the system engineer.

## Qu, L; Liu, D. Aspect Mining Using Method Call Trees (2007)

“Aspect mining tries to identify crosscutting concerns in legacy systems and thus supports the adaptation to an aspect-oriented design.” This is relevant anytime software needs to be promoted to new frameworks and technologies.

One of the challenges with legacy software is that it tends to be grey box, by which the details can be known but are expensive to extract. Previous efforts have tried to work around this by processing predefined workloads through the system and then taking snapshots of the application state at runtime.

However, this is not a complete solution as the workloads might not be representative of the entire system. The results can also be overfitted and misrepresent the priority to address certain results.

The authors mitigate this scenario by using static analysis instead of dynamic analysis. First, they extract the call graph from the application and label which methods call what other methods. As they traverse the graph they used a stack to build up the relationships of “A leads to B.” This is converted into a matrix and the summation of these state changes is provided.

Most of the matrix will have a low or zero valued score as most methods do not call most other methods. Where an aspect needs to exist, there will naturally be a higher score. For instance, the logging code is called from everywhere thus its methods will have a high score.

## Cojocar, G; et. al. Top-Down Aspect Mining Approach for Cross Cutting Concerns(2017)

## Mens, K; Kellen, A. Pitfalls in Aspect Mining (2008)

# Topic 2: Functional Programming

## Aliv, d; et.al. Comparative analysis of Functional and OO Programming (2016)

## Khanfor, A; Yang, Ye. Overview of Practical Impacts of Functional Programming (2017)