**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)

The authors acknowledge that generically finding aspects within a program is complex and can have lots of false positives. They propose using several specialized search algorithms to find specific types of aspects.

The most frequently referenced aspect is logging and to no surprise they choose logging as their proof of concept. The algorithm looks at class metadata and attempts to find static fields are named “logger” or “tracer.” References to these objects are then tracked through the codebase. The results are sorted based on the number of references to the implementation class.

The biggest challenge with this approach is that it is overly specialized and is unlikely to work in most scenarios. If the logger was called “OutFile” or was an instance field it would be missed.

These challenges are briefly touched upon and dismissed by the authors as being non-issues. This belief is driven by the assumption that their four examples are representative of the entire Java ecosystem.

According to Table III clearly this cannot be the case as all projects use the same Apache logging framework. Instead the author should have identified the top four logging frameworks for Java and examples which used them. Another challenge with their approach is that the correct logger was frequently not identified within the top 5 results. For instance, the logger for Spoon was the 15th proposed class in the list.

The paper concludes by stating that the specialized searching method is highly scalable. However, this cannot be true as each aspect minor is overly specialized and is unlikely to work across a series of related libraries. At best this is glorified text extraction at worse this is not a fully thought out solution.

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

“Most aspect mining techniques have not lived up to their expectations yet. In this paper [the authors] … provide a detailed accounting of root causes.” The authors continue to describe how aspect mining and aspect refactoring have attempted to automate their trade, however it is often immature.

The first challenge is identifying which crosscutting concerns should be migrated into aspects. Early aspect mining attempts to look at early versions of the code as these initial ideas are less likely to be well thought out, leading to poor design seeds. They also discuss special purpose code browsers and automated mining via seed points.

In the Top-Down Logging paper they could have used the mining seed approach to improve their efficiency. Often the developer knows which class implements the logger, and automated discovery is of minimal value. Instead the value-add is fully realized after the mining identifies the classes and their usages.

The authors state that aspect mining tends to be unreliable because (1) poor precision, (2) poor recall, (3) subjectivity, (4) scalability, (5) empirical validation, (6) comparability and composability, and (7) simple crosscutting concerns are not so simple.

This suggests that the entire field maybe immature and lacking the additional layers of abstraction. For instance, the poor precision and recall sounds like overly optimistic matching of examples. An alternative explanation might be imprecise definitions of the search criteria. No matter how much data is available, without proper search constraints the right answer will not be returned.

The authors conclude by describing methods to improve the quality of results. One solution is to move from simple pattern matching to semantic modeling. This is analogous to web crawling, where the advanced systems look for entities instead of simple words.

# Topic 2: Functional Programming

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

Functional programming languages implement lambda calculus logic constructs as a mechanism for describing transformation, reductions, and conversions.

By convention the input arguments are not modified while producing the output. This enables many different functions to run concurrently across the same data sets and not encounter collisions. By convention OO languages take the opposite approach where the state is passed around, and the exposed methods directly modify itself.

This is an overly generalized statement as immutable design is common practice in modern OO languages. Similarly, functional languages allow passing by reference which leads to shared state between functions.

The authors performed a deeper analysis of FP vs OO programming by implementing three standard algorithms in Haskell, C#, F#, and Java. The performance characteristics of the programs was measured. Based on the results Java was the fastest and Haskell had the fewest lines.

While they took accurate measurements the results and meaningless. No details are provided how the C# code was built nor if they used F# native libraries. None of these results could be identically reproduced by another peer. The test environments are also mixed with Linux for Haskell and Java versus Windows for F# and C#.

The challenge with naively comparing OO languages versus FP is that the differences are only syntactical. Consider C++ the gold standard of OO languages which passes around objects, but what are those objects? They are regular C structures with an extra pointer to a virtual call table. Then look at instance methods which are really functions that require the first parameter be the instance type. It’s all smoke and mirrors.

Instead the discussion should center around what paradigm is best suited for a given use-case. For instance, adding one to every value in an array can be more efficiently written as a function than a class with one static method. The optimizer is as likely to choose one as the other. This leads to the need to focus on readability when designing software.

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

What is the research outlook in FP paradigm? In which software engineering development practice FP paradigm had been studied? These research questions were addressed by reviewing 184 papers published between 1980 and 2016.

Pure functional languages lack common features such as state and exception handling. This can complicate the design and make those languages less feasible for general industry adoption. With less adoption it stands to reason there will be fewer experts and the maintainability costs of pure functional languages must increase.

Impure functional languages are more susceptible to security issues due to the possibility of state changes. Though any language that allows dynamic memory allocation and direct memory access will always be at risk.

An argument is made that FP is more reusable than OOP due to each compilation unit being smaller and more well defined. They make this claim based on a comparison of SML versus C++ where they find that SML uses more library functions in a contrived example.

This argument sounds more academic than a real-world challenge. It misses the point that the industry lives on frameworks not snippets. Reusability comes from the object model that represents a task.

Next an argument is made that FP languages are more difficult to test than an OO model. The research claims that this is due to race conditions and “off-by-one” calculations. It is not clear that the language is at fault. Tools and frameworks are being enhanced daily however most of it goes to OOP because that’s where the audience lives. For instance, Visual Studio has extensive support for static analysis of C#, some for C++, and a little for C. This makes sense as there is more C# than pure C written within their product.