

FOP and AOP: Benefits, Pitfalls and Potential for Interaction

Ian Adams

Graduate Student
Department of Computer Science
University of California, Santa Cruz
iadams@soe.ucsc.edu

Sigmon Myers

Graduate Student
Department of Computer Science
University of California, Santa Cruz
sig@soe.ucsc.edu

Keywords Aspect Oriented Programming,
Feature Oriented Programming

1. Introduction

We intended to find if in combining FOP and AOP methodologies we could devise a solution which reduces complexity and maintainability of a given software system. Additionally we worked to analyze FOP and AOP as individual methodologies and critiqued their strengths and weaknesses.

One of the issues in the feature-oriented programming (FOP) paradigm is the proliferation of very similar or identical features throughout the diagram and, consequently, the implementation. This leads to unnecessary code duplication and complicates both the modeling and implementation of the program. Our work combines aspects from aspect-oriented programming (AOP) into the stepwise refinement method of FOP to reduce both the code duplication problems and the overall complexity of a software systems design and implementation. To accomplish this we proposed a modified feature diagram to model a combined AOP and FOP program, and produced a small example as a proof of concept using FeatureIDE and AspectJ. Additionally we examine the individual strengths and weaknesses of FOP and AOP in the light of the aforementioned tools. The paper is structured as follows: a discussion of related work and introduction of AOP and FOP, details of our methodology, and finally our results and summary.

2. Related Work

Here we will present the current state of FOP and AOP research, paying particular respect to the AHEAD Tool Suite (2) and AspectJ (12). We will then discuss current research which focuses on a fusion of FOP and AOP.

2.1 Feature Oriented Programming

FOP is a relatively new programming paradigm whereby programs are modeled as sets of features that in aggregate represent the final product. FOP was designed with an eye towards software product lines (8), which allows for signif-

icant code reuse and the generation of arbitrary programs consisting of user-selected features. The intent is that this will increase code reuse and modularity, while reducing code bloat and proliferation of unnecessary program elements. A related programming and modeling paradigm is variability modeling. Variability modeling also is aimed at software product lines and code reuse (7).

For our research we used the AHEAD and FeatureIDE tool suites. These work by modifying a base program with incremental refinements that are called *features*. These features correspond directly to those in a feature diagram modeled during feature-oriented domain analysis, aka FODA (2). Currently, the creators of the AHEAD tool suite are working on algorithms to understand modifications occurring between feature diagrams (4). Other aspects of their research include computational design (5) and generation of test cases for software product lines (6).

2.2 Aspect Oriented Programming

AOP is a programming methodology which is used to address crosscutting concerns within a system. A typical example of a crosscut would be logging. This is because in object-oriented systems logging is often a repetitive task carried out in a variety of different classes. To deal with this, aspects can define a set of join-points, which are well-specified areas of code. When a join-point occurs, a pointcut is created to inject advice at the specified join-point. Advice is the actual implementation of a task such as logging or error handling in one centralized place. Aspects are therefore a powerful technique to inject advice into a code base without actually modifying any existing code.

The AOP research community is currently focusing on analyzing the effects of AOP in various forms. Design patterns, which are a method of expressing solutions to recurring problems, are being explored by Cacho et al. This group is working on utilizing aspects with design patterns to assess if AOP can help handle problems relating to pattern interactions (18). Kulesza et al. have worked on analyzing the use of aspects for maintenance tasks in a quantitative study,

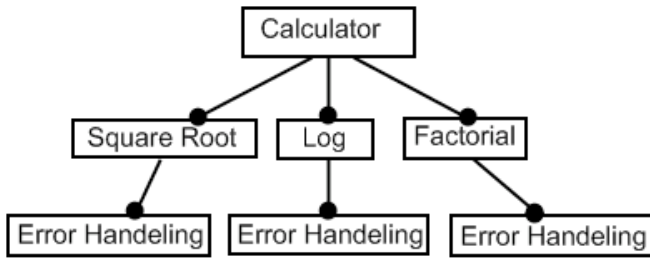


Figure 1. A feature diagram of a simple calculator. Note the repeated error handling feature.

determining that AOP systems lead to more stability and reusability (19). Work is also being done relating to the direct applicability of AOP and web services. This has been popular in efforts geared toward making code more manageable and reusable (20).

At the forefront of AOP implementations is AspectJ, where active development continues today. Currently, AspectJ has an open-source community and regularly addresses issues, bugs and other work items on a rolling three month schedule. This is a well-formed community and is leading to a more stable and refined system to work with as time progresses.

2.3 Feature and Aspect Research

There are currently a few different groups pursuing research relating to aspects and features. Lee et al. are working on a refined method for coupling feature oriented analysis with AOP (16). Batory et al. have recently discussed when the use of features and aspects is called for (17), whereas other members of the same research group are also discussing the usability of a system containing features and aspects (15). It seems relatively little research which explicitly relates to both AOP and FOP usability is being done outside the Product Line Architecture Research Group headed by Batory.

3. Methodology

For the purposes of examining the merits and downfalls of AOP and FOP we created a few small test programs designed to explore these programming methodologies and the existing tool support. This was somewhat tangential to our primary goal of combining FOP and AOP and thus is discussed only briefly in our evaluation section as the design and implementation of these programs is not particularly interesting or relevant to our main goal.

Our method to develop a software system including both features and aspects consisted of three steps. The first was modeling a combined feature and aspect program as a modified feature diagram, this will be described shortly. Second, we implemented the core program in FOP, utilizing the FeatureIDE tool set for Eclipse. The third and final step was

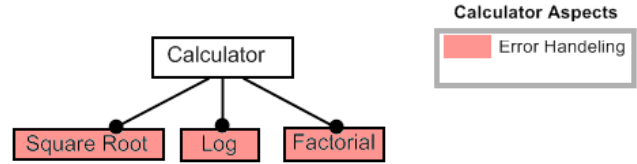


Figure 2. The calculator feature diagram with aspect coloring. The coloring reduces the size of the diagram and clearly marks what features will be influenced by an aspect.

the application of aspects (via AspectJ) to the generated java source code from FeatureIDE.

3.1 Feature and Aspect Modeling

One of the first problems to address when considering an FOP implementation is how to graphically represent the system in such a way that it is directly implementable from a completed feature diagram and minimizes redundancy of features. Figure 1 represents a feature diagram of a calculator which has three recurring features, *Error Handling*. The mathematical functions *Square Root*, *Log* and *Factorial* all require a value greater than zero to be taken as input. Therefore, to implement this error handling feature across all of these features, we introduce an aspect which obviates the need to hand code these recurring features multiple times. To represent the removal of this redundancy and the addition of an aspect in the feature model, we add a coloring mechanism which reduces clutter and improves readability as shown in Figure 2. The coloring mechanism adds a color (or it could easily be some small symbol) to the feature model. Features that have the same color all have some equivalent cross-cutting concern which is then detailed in a key adjacent to the diagram. Note how a small aspect can have a significant impact in reducing the model size. Figure 3 shows the necessary code to implement the error handling aspect, which impacts three features in our example. This new notation method reduces the visual complexity of the diagram, improves understandability when implementing features that would otherwise be repeated throughout the code base, and clearly demarcates where and what the aspects will affect.

```

pointcut errorHandler(double d) : call(* Calculator.calculate(double))
    && (target(Factorial)
    ||target(Log)
    ||target(SquareRoot))
    && args(d);

before() : errorHandler(double d)
{
    if(d<0)
        throw new IllegalArgumentException("Positive numbers only, please.");
}
  
```

Figure 3. Error handling code for an aspect.

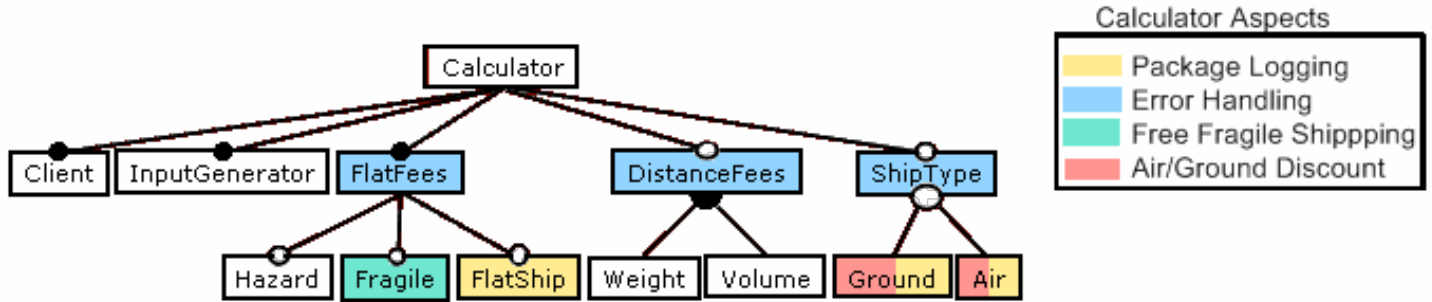


Figure 4. Colored feature diagram for the shipping calculator test program.

3.2 A Shipping Calculator Example

To illustrate the use of aspects and features in a software system, we have developed a generic command-line shipping calculator. The goal of this program implementation is to identify the benefits of incorporating aspects into a feature oriented system using currently available software. Due to this being a proof of concept, we aimed at creation of a basic functioning program, not an ideal solution for shipping calculators.

Our shipping calculator contains several features: a shipping method of air or ground, a distance calculator based on volume or weight of an item, and optional hazardous, fragile and flat-rate fees. Additional functionality (and reduced code duplication) is incorporated through aspects: logging the shipment of an item, error handling for incorrect input, and holiday discounts for free fragile-item shipping and reduced cost ground shipping. See Figure 4 for the feature and aspect model of this system.

3.2.1 Feature Development

The core program and functionality was developed using AHEAD through the FeatureIDE Tool suite in the Eclipse development studio. This was a straightforward FOP development. Though we intended to also have the FOP portion automatically generate the appropriate input prompts (note input generator feature in Figure 4) as user selected in the feature model, in the interest of time we chose to omit this. This omission did not affect our results as we were interested in a proof of concept using off the shelf tools, not the generation of a fully functioning software product line.

As far as implementation details are concerned, each of the individual features was coded in a .jak file and automatically composed via the AHEAD tool suit. These composed files were compilable java source files that could be called via another client program, and thus were also the files that were referenced by the aspects discussed in the following section.

3.2.2 Aspect Development

The next step in the development of our shipping calculator was to add various aspects for cross-cutting features. The first cross-cut implemented was shipped-package logging. Aspects generally lend themselves quite well to logging, and this was not an exception. After a user has input a package to the system, we logged that package and it's shipping type (ground, air or flat-rate) by creating a pointcut anytime the methods *groundFee*, *airFee* or *flatrate* finished execution (see Figure 5 for a code example).

```

/*Log the completion of a shipped package via flat-rate, ground or air*/
pointcut itemShipped() : call(* FlatFees.FlatShip(..))||
    call(* ShipType.*Fee(..));

after() : itemShipped(){
    System.out.println("A Package has been shipped via: ");
    if(thisJoinPoint.getClass()==FlatFees.class)
        System.out.println("Flat-Rate Shipping.");
    String temp = thisJoinPoint.getSignature().toShortString();
    if(temp.contains("FlatShip")){
        System.out.println("Flat-Rate Shipping.");
    }
    else if (temp.contains("Air"))
        System.out.println("Air Shipping.");
    else
        System.out.println("Ground Shipping.");
}
}

```

Figure 5. Package logging aspect for shipping calculator.

The second cross-cut in our model was error-handling. No input should be collected in the 'DistanceFees', 'FlatFees' or 'ShipType' features (and all corresponding subfeatures) which is a negative number. We assume that this advice should not be generic enough to execute outside of the specified classes for future maintainability purposes. To do this we simply add the *within(Class)* modifier to our pointcut.

The previous two aspects were implemented using standard public advice. For the last feature we utilized a privileged aspect, which can see and modify any object within a class. Currently this ability is seen as too over-reaching but for now remains in AspectJ implementations. Our privileged aspect is a holiday bonus modification, which will offer no fee for fragile items, 33% off air shipping and 45% off ground shipping.

4. Evaluation

Here we present an evaluation of our experiences with FOP, AOP, and combining the two paradigms.

4.1 Feature Oriented Programming Analysis

4.1.1 Strengths

FOP and feature diagramming allows for a logical design structure. We have a hierarchical decomposition of a program into its constituent features. This approach tends to mimic, in our opinion, the actual thought process behind the design of a product for use in software product lines. A product of this approach has the ability to generate a wide variety of programs without designing each one from the ground up. Additionally, given a well-defined set of constraints, it becomes possible for an unskilled user to detail and generate a program which fits their needs precisely, rather than a system containing extraneous functionality.

In a depth first search (DFS) program we built, we were able to see these previously mentioned strengths of feature oriented programming. One was the ability to generate many similar, but functionally different programs from the same set of features simply through selection of the desired features. For example changing between a directed or undirected graph, or choosing whether or not the program would classify the edges. Additionally, with carefully designed models and equations it becomes impossible for a user to create an invalid program. As another example, in our DFS program the connected component identifier only worked with undirected graphs, but with properly implemented constraints, it was impossible for a user to select this invalid combination of features. This feature selection illustrates the aforementioned ability for an unskilled user to select the desired program functionality with no knowledge of the underlying code.

4.1.2 Weaknesses

Beyond the feature and code duplication problems mentioned earlier, perhaps the biggest pitfall of both this design paradigm and the related variability modeling is scalability. Both feature diagrams and variability models grow at staggering rates when the programs become even moderately complex. Compounding this problem is the nature of interactions between features. The selection of one feature or asset can necessarily preclude or enforce the selections of others. This is a difficult problem in that it becomes hard to create and present, in a human-understandable fashion, the complex interactions between feature choices and constraint propagation. Even in the small DFS program we implemented it was necessary to hand code several constraints that occasionally interacted in surprising ways.

Another problem which is prevalent in almost any new programming paradigm is that of effective debugging and

FOP is no exception. With the complex interactions between many features and constraints it can become very difficult to track down the source of a bug, even more so than in more conventional purely object oriented systems due to the scattered nature of feature selection, propagation, and interaction.

Yet another issue is that there can be a disconnect between the feature diagram and the actual implementation. This is particularly evident in the stepwise refinement found in AHEAD. For example, let us examine an abstract data type as it would exist in a feature diagram. We have our access methods, manipulation methods, calculations, etc. each as individual features. While it makes sense from a design perspective to decompose these into atomic features beneath the ADT, it becomes quite messy as an implementation, and unnecessary if all those features are mandatory across any selection of other features. Unless care is taken we can have a needless proliferation of features as well as a confusing spread of code.

All of the above are in essence scalability problems. They are easy enough to deal with in a small program but quickly balloon to unmanageable sizes in even moderately sized implementations. This highlights the need for a fundamental change in how we approach the design and creation of software product lines under this programming paradigm. There are many possible small tweaks that could ease these issues, such as intelligent abstraction of parts of diagrams, and compressing mandatory features into singular entities. However, the primary issue still remains of complicated interactions between features, and explosion of diagram size and model size as product lines increase in complexity. We hope that our modest revision of feature diagramming and combined FOP-AOP programming will alleviate some of the above scalability concerns.

4.2 Aspect Oriented Programming Analysis

4.2.1 Strengths

In general, aspects are described to be useful mostly for logging purposes. While this is true, there are other features to aspect oriented programming which have a positive impact when mixing with feature oriented programming. As shown in section 3.1, we can use aspects to eliminate the need for recurring features, such as error handling. The error handling in our shipping calculator, shown in Section 3.2, could have been made more generic to ensure a positive number for all input in the system. This would have resulted in one aspect checking user input across sixteen different method calls in our software system, which is a generous gain!

Just as aspects are capable of being extremely general, they are also well-equipped to handle very fine-grained areas of a software system. A join-point can be defined to entail a spe-

cific class, a method call within a class, another join-point, or even another aspect. The ability to determine when an event is occurring within a software system, and not need to modify any code creates a large benefit in the maintainability and integrity of the system.

4.2.2 Weaknesses

Relevant to AspectJ, there are a few key drawbacks. First, the capabilities of privileged aspects are far-reaching. One can literally modify private member variables of *any* object within *any* class. This seems like a recipe for disaster in that the capability to change various values of any object, regardless of declaration, explicitly removes data hiding functionality within a software system.

Debugging an aspect can be rather tedious as well. Aspects are notoriously buggy during development because of a lack of useful error messages and warnings. During runtime, advice will be ignored if a pointcut is improperly defined, but that is the only information the programmer receives. Any additional information relating to why the advice was ignored could help prevent the need for endless searching for errors within a code fragment.

Another potential problem with advice is generality. While this can be a positive, it is quite possible to implement overly-general advice. In the original implementation of our shipping calculator, an error was thrown in the driver class anytime a user entered zero, which often was valid input. It is easy to imagine an aspect having the capability for overreaching its mandate on a large system if too general a join-point is defined.

4.3 Analysis of Combined FOP and AOP program

The perceived, meaning hoped for, benefits of incorporating FOP and AOP together are fairly obvious. The elimination of redundancy in feature models, a smaller code base, increased maintainability and decreased code complexity. All of these benefits were, in our eyes, ultimately achieved and demonstrated in our example implementation.

To model our domain for the shipping calculator, we initially utilized the FOP model with aspects represented in the traditional dashed line format (An example in UML is shown in Figure 6) (12). Each aspects lines would connect to various features in our feature model. It became quickly apparant that such a method would be, even in a moderately sized feature model, difficult to understand and elegantly draw. Our proposed coloring mechanism makes a feature model with aspects scale much easier for larger project sizes. Additionally, we were able to immediately understand the context and influence of aspects based on the clear and concise visual representation. The implementation of our model involved a layering of aspects onto the completed feature oriented shipping calculator. In a large-scale system, it would be possible

for separate programmers to build aspects and features concurrently based on a detailed feature model. Though more time-consuming, our approach was strictly linear.

Another noticeable gain in our system was a reduction in code-complexity. Aspects, generally amounting to no more than ten lines of code, were able to properly affect sixteen places within our small system. That is a huge amount of savings in terms of coding time. By not having any need for copy/paste coding, we are able to maximize our efforts in other areas of the system and its development.

FeatureIDE allows for generation of multiple variations of a system, based on feature selection and interaction. We thought this could be a potential problem but aspects, as implemented in AspectJ, are able to be ignored if a FeatureIDE generation does not involve a specific aspect. This is a useful trait because there is no need for removal or modification of aspects if an aspect is no longer pertinent to a particular variation of a generated FOP program. This can be considered positive from a maintainability standpoint. If we had to specify which features were acting on various subsystems our modeling and implementation would be significantly more complex.

There are still some issues to be addressed with combined FOP and AOP architectures in addition to their individual problems. First off, generality can still be a concern with aspects in this architecture. If the scope of an aspect is not limited, it is quite possible to still have an aspect affect more features in a system than intended. This is also partly due to FeatureIDE generating quite a bit of extra code. The best method for preventing such an issue is to explicitly state the context of an aspect, or what classes an aspect should offer advice in.

Alternatively, the generality of aspects can also be enticing for certain situations. Easily defined general aspects can add advice, in a menial amount of code, to a huge number of features operating in a system. This solves issues such as constraint propagation in FeatureIDE of recurring features. Therefore, because it is up to the programmer to decide on the generality/specificity of an aspect, it can be thought of as both a pro and a con depending on the situation.

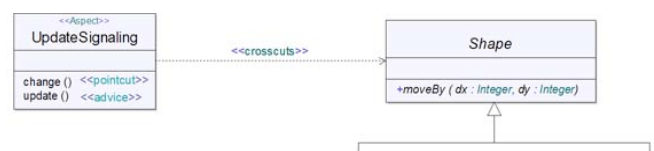


Figure 6. An aspect pointing to an object with a dashed line.

5. Conclusion

Overall, we found that FeatureIDE and AspectJ surprisingly work very well together. There was no need to modify the java source output from FeatureIDE's composed .jak files and no need for basing advice on quirky semantics of a generated system. Our initial expectation was that some modification of FeatureIDE's generated code would be needed in order for AspectJ to recognize various FeatureIDE peculiarities. This was not the case; no retrofitting of FeatureIDE or AspectJ was needed which was very encouraging at the onset of this project. After concluding our work with developing a system using these tools we were able to form various conclusions relating to gains in various areas of the development lifecycle.

It is clear to us that AOP is very complementary towards FOP systems. If an architecture based on features is needed, aspects can add valuable capabilities which make up for FOP shortcomings. This is especially the case in functionality crossing multiple features such as error handling. Because of this the addition of an AOP language can simplify the construction of a given FOP designed program. Some worthwhile future work would be to investigate the quantitative gains made in development time of two systems—one with features only and another which incorporates aspects.

In building this system, we were able to cultivate a new modeling mechanism for combining these two programming paradigms. We believe this addresses some scalability issues of FOP modeling. Our modeling mechanism, which reduces or eliminates recurring features also allows for greater ease in using FeatureIDE because there are less constraint propagations to work through during implementation when similar features occurring in multiple areas of the FOP model are withdrawn and defined as aspects.

In the end, we have found that the combination of Aspects into an FOP designed program was surprisingly easy to implement. Even with a mashed together implementation of two unrelated off the shelf tools, we demonstrated the viability and usefulness of a new combined design and implementation methodology.

The FeatureIDE Eclipse project and AspectJ may be downloaded at this link www.soe.ucsc.edu/iadams/Proj.zip

If the reader wishes to view the other code mentioned in this write up, please contact the authors.

References

- [1] Krzysztof Czarnecki, Eisenecker, U.W. Generative Programming, Addison-Wesley, 2000.
- [2] Don Batory. Feature-Oriented Programming and the AHEAD Tool Suite. In Proceedings of the International Conference on Software Engineering 2004, 2004.
- [3] Don Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In Generative and Transformational Techniques in Software Engineering, International Summer School (GTTSE), 2006.
- [4] T. Thuem, D. Batory, and C. Kaestner. Reasoning about Edits to Feature Models, International Conference on Software Engineering (ICSE), May 2009.
- [5] D. Batory, M. Azanza, and J. Saraiva. The Objects and Arrows of Computational Design, Model Driven Engineering Languages and Systems (MODELS), October 2008.
- [6] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. Testing Software Product Lines Using Incremental Test Generation, Int. Symposium on Software Reliability Engineering (ISSRE), November 2008.
- [7] Deepak Dhungana, P. Grnbacher, and R. Rabiser. DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling. The First International Workshop on Variability Modelling of Software-intensive Systems. Limerick, Ireland, 2007, pp. 119–128.
- [8] Deepak Dhungana, R. Rabiser, P. Grnbacher. Decision-Oriented Modeling of Product Line Architectures, Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture, p.22, January 06-09, 2007.
- [9] Rick Rabiser, P. Grunbacher, D. Dhungana. Supporting Product Derivation by Adapting and Augmenting Variability Models. Proceedings of the 11th International Software Product Line Conference, p.141-150, September 10-14, 2007.
- [10] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature Modeling Plug-in for Eclipse. In OOPSLA '04 Eclipse Technology eXchange (ETX) Workshop, 2004.
- [11] Don Batory, J. N. Sarvela, A. Rauschmayer. Scaling Step-Wise Refinement, Proceedings of the 25th International Conference on Software Engineering, May 03-10, 2003, Portland, Oregon.
- [12] AspectJ: <http://aspectj.org>.
- [13] Joseph D. Gradecki, N. Lesiecki. Mastering AspectJ: Aspect-Oriented Programming in Java, John Wiley and Sons, Inc., New York, NY, 2003.
- [14] Adrian Colyer, A. Clement, G. Harley, M. Webster. Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ development tools. Addison-Wesley, 2004.
- [15] Sven Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In Proceedings of the International Conference on Software Engineering 2006, May 2006.
- [16] Kwanwoo Lee, Kyo C. Kang, Minseong Kim, Sooyong Park, "Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development," Software Product Line Conference, International, vol. 0, no. 0, pp. 103-112, 10th International Software Product Line Conference (SPLC'06), 2006.
- [17] Apel, S. and Batory, D. 2006. When to use features and aspects?: a case study. In Proceedings of the 5th international Conference on Generative Programming and Component Engineering (Portland, Oregon, USA, October 22 -

- 26, 2006). GPCE '06. ACM, New York, NY, 59-68. DOI=<http://doi.acm.org/10.1145/1173706.1173716>
- [18] Cacho, N., Sant'Anna, C., Figueiredo, E., Garcia, A., Batista, T., and Lucena, C. 2006. Composing design patterns: a scalability study of aspect-oriented programming. In Proceedings of the 5th international Conference on Aspect-Oriented Software Development (Bonn, Germany, March 20 - 24, 2006). AOSD '06. ACM, New York, NY, 109-121. DOI=<http://doi.acm.org/10.1145/1119655.1119672>
- [19] Uira Kulesza, Cl?udio Sant?Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, Carlos Lucena, "Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study," Software Maintenance, IEEE International Conference on, vol. 0, no. 0, pp. 223-233, 22nd IEEE International Conference on Software Maintenance (ICSM'06), 2006.
- [20] Guadalupe Ortiz, Frank Leymann, "Combining WS-Policy and Aspect-Oriented Programming," Advanced International Conference on Telecommunications / Internet and Web Applications and Services, International Conference on, vol. 0, no. 0, pp. 143, Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT-ICIW'06), 2006.