AOVis: A Model-Driven Multiple-Graph Approach to Program Fact Extraction for AspectJ/Java Source Code

Jeffrey Koch[1], Kendra Cooper[2]

[1]*Department of Computer Science and Information Technology*
Tarrant County College - Northeast Campus
828 West Harwood Road, Hurst, Texas 76054 USA
jeffrey.koch@tccd.edu

[2]*Department of Computer Science*
The University of Texas at Dallas, 800 West Campbell Road, Richardson, Texas 75080 USA
kcooper@utdallas.edu

Abstract- **AspectJ reverse engineering and visualization remains a challenge at the architectural and design levels, with fewer tools available for reverse engineers compared to other languages such as Java. Prior work on AspectJ modeling focused on forward engineering or detailed-design reverse engineering, or required special instrumentation to identify cross-cutting relationships. Effective visualization is also a challenge since cross-cutting relationships can be difficult to depict cleanly and effectively.**

AOVis (Aspect-Oriented Visualization) is a reverse engineering framework to extract UML-based detailed design, architecture, and analysis level models of Java/AspectJ projects. This framework employs automated on-demand transformation of implementation models to formal design models using AI based pareto optimal ranking techniques and established quality metrics. It also expresses and stores its results as UML data through the use of standard UML extensions, allowing the resulting models to be used and analyzed by existing UML-compliant tools. Finally, it provides separation of cross-cutting concerns from the base model in accordance with aspect-oriented principles.

This paper focuses on a three-step model-driven multiple-graph approach to automated program fact extraction that leverages existing reverse engineering technologies to support the demands of AspectJ reverse engineering. The validation includes a prototype tool which we believe to be the first to leverage information from production AspectJ compilers, as well as the use of the AJHotDraw, AJHSQLDB, Contract4J, and *J-Pool open source AspectJ projects.

Keywords - **Aspect J, eclipse, plugis reverse engineering and visualization.**

1. Introduction

The aspect-oriented paradigm addresses concerns that cut across other concerns, including error and failure handling, performance optimization [28], security requirements [3, Introduction to AspectJ], contract enforcement, tracing, logging [3, Development Aspects], monitoring [27], and other non-functional properties [11, 12]. In object-oriented systems, implementation of these cross-cutting concerns are usually scattered across multiple classes and tangled with other concerns. This introduces the risk of incomplete or inconsistent implementation of the cross-cutting concern as well as interference between concerns. Aspects eliminate scattering by centralizing the implementation and specification of join points–places where aspect behavior should be inserted. They also eliminate tangling by leaving other concerns unaffected [28]. One popular aspect-oriented implementation is AspectJ which extends Java by adding aspects [3, 4].

Although AspectJ and aspect-oriented forward engineering is the subject of ongoing research [10, 16, 22, 1], AspectJ reverse engineering has received less attention. Aspect-oriented software development poses several challenges in reverse engineering and the visualization of reverse-engineered models. These challenges may be considered in terms of the fundamental activities of reverse engineering: input or extraction, transformation or abstraction, and output or presentation [9, 33].

The first step in reverse engineering aspect-oriented code is the extraction of program facts–the system's elements and the relationships between them–in a form that can be used by the rest of the reverse engineering system. In AspectJ, program facts include aspects, pointcuts, advice, inter-type elements, and cross-cutting relationships in addition to object-oriented elements and relationships. Some compilers make program fact information programmatically available through abstract syntax trees, light-weight models, or both [35, 49]. The system must usually be successfully compiled in order to obtain such information; however, these models provide access to the same program fact information as the compilers. Parsers are also commonly used to extract program facts from other languages; however, AspectJ parsing is complicated by pointcut semantics [50]. As Avgustinov et al. [5] notes, "in AspectJ itself, the pointcut language is very complex, allowing the programmer to capture intricate properties related to the static structure of the program."

We identified several design model reverse engineering issues in [9], and the long term vision of this research is to address these issues though an automated reverse-engineering visualization approach with several important characteristics including:

Program fact extraction including Java (object-oriented) program facts and AspectJ-specific program facts including aspects, inter-type declarations, pointcuts, advice, and the cross-cutting relationships created by these features. As Section 2 discusses, several tools are available for Java program fact extraction; however, AspectJ tools and techniques are not readily available.

Representation of program facts and models that allows internal reuse of existing tools and external reuse of extracted program facts and abstracted models by external tools. This includes the representation of AspectJ and Java program facts and models, the representation of details appropriate for reverse engineering, and a standards-compliant representation format.

High-level architecture and early visualization ideas were introduced in [31]; an overview of the fact extraction approach was presented in [32].

We have developed a three-step model-driven multiple-graph approach to automated program fact extraction that leverages existing reverse engineering technologies to support the demands of AspectJ reverse engineering. This approach answers the need for AspectJ program fact extraction techniques and tools as well as the need for reusable models and tools.

The rest of this article expands upon the research described above. Section 2 discusses recent work and common approaches in extraction and aspect-oriented reverse engineering; this is an expansion of the discussion from [32]. Section 3 provides an overview of the proposed AOVis approach. This approach was described in [31] and [32], and is repeated here to make the article self-contained. Section 4 substantially expands upon the AOVis program fact extraction framework introduced in [32] by describing the models, graphs, and profiles involved in the framework, introducing the algorithms used to extract Java and AspectJ program facts, and presenting examples. This section also describes tool support and presents validation results. Finally, Section 5 presents the conclusions and future work for this project.

2. Related work

2.1 Program Fact Extraction

Current reverse engineering research and commercial tools approach program fact extraction through compiler-generated models, models generated by extended versions of standard compilers, and external tools such as byte code analyzers and parsers. This section compares existing approaches to the program fact extraction requirements outlined in Section 1: extraction of Java program facts, extraction of AspectJ program facts, and standards-compliant representation of extracted program facts.

2.1.1 Compiler Model

Some compilers create a model of the system's structure that can be used by reverse engineering systems in lieu of the actual source code. For example, Eclipse's Java Development Tools (JDT) generate abstract syntax trees (AST) that map every token in the source code into a tree of nodes [34], as well as light-weight Java Models that maps the source code down to types, fields and methods rather than individual tokens [35, 49]. These "compiler models" are then used by other JDT tools to generate Eclipse views of the file structure, system structure, and relationships. JDT makes these models available programmatically, and other tools and plug-ins have used the AST and Java Model to reverse-engineer systems by generating other models using the Unified Modeling Language (UML) or other modeling languages [47, 25]. These tools are able to extract Java program facts, but not AspectJ program facts.

AspectJ support in Eclipse includes the AspectJ Development Tools (AJDT) which provide a two-part compiler model: a JDT-compatible Java Model and the AJProjectModelFacade, a lightweight model of the AspectJ system for access to aspect-related program facts. Like JDT, other AJDT tools use this program fact information to generate aspect-specific Eclipse views. The facade is also available programmatically; however, our survey of the literature did not identify any projects that used the production version of AJDT to extract program fact information from AspectJ projects.

2.1.2 Extended Compiler Model

Coelho [8] developed an AspectJ reverse engineering framework with similar goals; however, the standard

AJDT AJProjectModelFacade could not provide the level of program fact detail required by the project. This issue was resolved by extending the AspectJ compiler to generate a more detailed AJProjectModelFacade that could be used to create UML models with extensions for cross-cutting relationships. This work successfully extracted Java and AspectJ program facts, although modification of the existing AJDT compiler was required. It represented the program facts as UML models; however, it relied upon non-standard UML extensions to represent AspectJ-specific features such as inter-type elements and pointcuts.

2.1.3 External Tools

Other reverse-engineering tools extract program facts from the code artifacts, either as a design decision or because program facts aren't externally available from the compiler suite. The Byte Code Engineering Library, or BCEL, allows users to work with the symbolic information within binary Java class files [2]. This information has been used to extract program facts for reverse engineering purposes [53] as well as other applications such as memory profiling, test coverage, and software protection [48]. BCEL is able to extract Java program facts, but not AspectJ program facts.

Other tools such as Rigi [29] and Bauhaus [41] extract program facts without depending on compiler output, either through internal tools or through the addition of external third-party tools such as parsers. As noted earlier, the semantics of pointcuts, particularly dynamic pointcuts, are particularly challenging [50, 5], and attempts to faithfully capture those semantics through parsers would be risky, especially since existing open-source tools such as AJDT already make pointcut information readily available without those risks.

2.1.4 Summary

As Table 1 illustrates, current approaches for extracting program facts do not completely satisfy the extraction-specific characteristics outlined in Section 1. The Extended Compiler Model does successfully extract AspectJ program facts, but requires modifications to the standard AspectJ compilers.

TABLE 1
CURRENT EXTRACTION APPROACHES

	Java Program Fact Extraction	AspectJ Program Fact Extraction	Standards-Compliant Representation
Compiler Model	X		
Extended Compiler Model	X	M	P
BCEL	X		
Parsers	X		

X = Supported
M = Supported with compiler modifications
P = Partially supported

2.2 Model Representation

As noted in Section 1, UML is the de-facto standard used by object-oriented modeling tools. XML Metadata Interchange (XMI) is the standard developed by the Object Management Group for exchanging UML and other meta-model data via Extensible Markup Language (XML) [40]. XMI allows UML models to be saved and read as-is by any UML tool that understands XMI, making it an excellent candidate for representing program facts and models in object-oriented environments, although UML doesn't provide built-in support for aspect-oriented concepts.

A common reverse engineering technique is to design a standard notation or intermediate language to represent the extracted program facts, allowing extraction to be decoupled from the rest of the reverse engineering process. Examples include Rigi Standard Format from Rigi [29], InterMediate Language (IML) from Bauhaus [41], and the Annotated Syntax Graph (ASG) used by SolidFX [46]. These notations are not as widely supported as XMI; however, they do allow program facts to be exported in a format that can be used by other tools. They also do not explicitly model aspect-specific details, although tool users can create models that depict such details manually.

Since the aspect-oriented paradigm is an extension of the object-oriented paradigm, research to extend UML to represent aspect-oriented elements is ongoing [8]. For example, Hachani [21] used subclasses to represent AspectJ features, while Stein, Hanenberg and Unland [44] used stereotyped operations and dependencies. Some of these earlier projects used non-standard mechanisms to extend UML, preventing existing tools from supporting these extensions, while other projects failed to take full advantage of the UML meta-model.

Evermann [16] defined an AspectJ meta-model as a UML profile, allowing it to be supported by UML-compliant tools as-is. It also modeled base-model and crosscutting concerns separately. Since this work concentrated on forward engineering, the focus was on a complete meta-model of AspectJ that fully reflect its nuances, including 17 non-abstract pointcut types, relationships between each pointcut type and the affected code, and relationships between advice and the referenced pointcut. This level of complexity is necessary when transforming UML models into source code; however, it doesn't easily lend itself to reverse engineering where the resulting models are intended to provide clarity for human readers. This work was later extended to accommodate other aspect-oriented languages and automatic code generation [1].

Mosconi et al. [39] took a similar approach, but defined pointcuts in terms of pointcut expressions (PCEs) specified in a meta-model based expression language rather than String-based PCEs or direct references to model elements.

These papers concentrated on forward engineering, and did not consider the impact of these technologies on reverse engineering. For example, AJDT doesn't identify different pointcut types, and it identifies advice-to-advised-code relationships rather than pointcut-to-affected-code or pointcut-to-advice relationships.

Table 2 summarizes current approaches for representing program facts and models, relative to the representation-specific characteristics outlined in Section 1. This table is divided between representation formats (XMI and intermediate languages) and AspectJ-specific UML extensions. As Table 2 illustrates, none of these approaches completely meet all of our goals–the representation of Java and AspectJ program facts at a level appropriate for reverse-engineering using standards commonly used by other systems.

TABLE 2

CURRENT MODEL REPRESENTATION APPROACHES

	Java and AspectJ Representation	RE-appropriate Level of Details	Standards-Compliant Representation
XMI	J	M	X
Intermediate Notation (RSF, IML, ASG)	M	M	P
Early AspectJ UML Extensions	X	M	V
Evermann UML Extension	X		X
X = Supported			
J = Supported for Java			
M = Manually implemented by user			
V = Support varies			
P = Partially supported			

3. AOVis OVERVIEW

AOVis (Aspect-Oriented Visualization) is a framework for the reverse engineering and visualization of AspectJ and Java source code to the detailed design and architectural levels. As Figure 1 illustrates, program facts are extracted from AspectJ/Java source code and saved as XMI files. Abstraction engines are then used to translate these program facts into models at the desired level of abstraction. Finally, the models are visualized as extended UML diagrams [31].

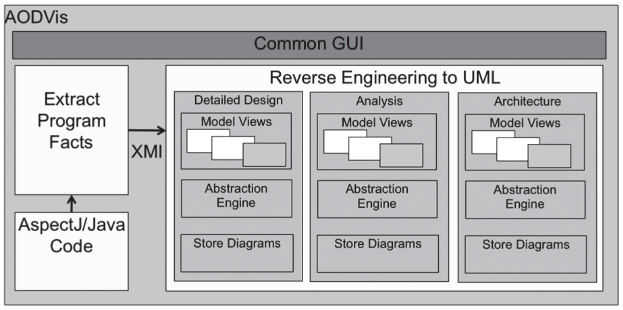


FIGURE 1: PROPOSED AOVIS ARCHITECTURE.

4. PROGRAM FACT EXTRACTION

4.1 Proposed Approach

This paper proposes a three-step model-driven graph-based approach for the extraction of program facts involving the generation of the compiler model, the extraction of program facts from the compiler model, and the translation of those facts into XMI. The proposed extraction approach, which reflects several concepts described by Bennett et al. as part of a graph-based approach to aspect-oriented forward engineering [6], is illustrated in Figure 2. Section 4.1.1 discusses the generation of the compiler model, Section 4.1.2 outlines the program fact extraction process, and Section 4.1.3 covers the XMI translation.

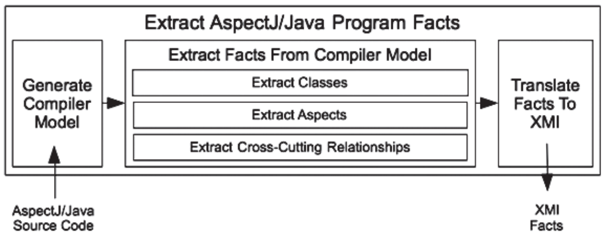


FIGURE 2: PROPOSED APPROACH TO EXTRACTING PROGRAM FACTS FROM ASPECTJ/JAVA PROJECTS TO XMI FILES.

4.1.1 Generate Compiler Model

The Eclipse AspectJ Development Tools plugin (AJDT) generates two distinct models of an Eclipse AspectJ project while building the project. These models are described below.

Java Model As noted in Section 2.1.1, the Java Model was introduced in JDT to map Java source code down to a hierarchical structure that identifies types, fields, and methods [34, 49]. The Java Model may be represented as a directed graph $JM = (V_{JM}, E_{JM})$ where V_{JM} is the set of nodes in the graph and E_{JM} is the set of edges in the graph.

V_{JM} may be divided into the following node types:

JavaProject$_{JM}$: The root node of a project's Java Model.
Packages$_{JM}$: Packages and subpackages within the *JavaProject*$_{JM}$ node.
CompilationUnits$_{JM}$: Java or AspectJ source files within a *Packages*$_{JM}$ node.
Types$_{JM}$: Classes and interfaces within a *CompilationUnits*$_{JM}$ node.
Methods$_{JM}$: Methods of a *Types*$_{JM}$ node.
Fields$_{JM}$: Fields of a *Types*$_{JM}$ node.

Attributes relevant to each node type are described through a tuple associated with that type.

E_{JM} contains directed edges $(v_{s1},v_{s2}) | v_{s1},v_{s2} \in V_{JM}$ denoting containment; if $v_{s1} \in Types_{JM}$ and $v_{s2} \in Fields_{JM}$, then $(v_{s1},v_{s2}) \in E_{JM}$ indicates that v_{s2} is a field of v_{s1}.

For backward compatibility with JDT, the AJDT Java Model does not include any aspect-specific information; instead, aspect-specific nodes are represented as their Java analogues, including classes for aspects.

AspectJ Facade AJDT provides aspect-specific structural and relationship information through the AJProjectModelFacade ("AspectJ Facade" or "Facade") which contains the following:

- A hierarchy that maps the source code down to types, fields, methods, aspects, and aspect-specific information. Like the Java Model, this AspectJ Facade hierarchy may be represented as a graph $AFH = (V_{AFH},E_{AFH})$ where V_{AFH} is the set of nodes in the graph and E_{AFH} is the set of edges in the graph. V_{AFH} contains node types corresponding to AspectJ-specific elements as well as elements common to AspectJ and Java; Table 3 lists the node types in V_{AFH}. Like E_{JM}, E_{AFH} edges denote containment.

TABLE 3
V_{AFH} NODE TYPES

Aspects$_{AFH}$
InterTypeFields$_{AFH}$
InterTypeMethods$_{AFH}$
InterTypeConstructors$_{AFH}$
InterTypeParents$_{AFH}$
Pointcuts$_{AFH}$
Advices$_{AFH}$
Projects$_{AFH}$
Packages$_{AFH}$
AspectjFiles$_{AFH}$
Classes$_{AFH}$
Methods$_{AFH}$
Fields$_{AFH}$

- Information that identifies the AspectJ Facade node that corresponds to a given Java Model node, and vice versa [15, 14]. This AspectJ-to-Java information may be represented as a graph $AJJ = (V_{AJJ},E_{AJJ})$ where $V_{AJJ} = V_{JM} \cup V_{AFH}$ and E_{AJJ} consists of edges $(vJM,vAFH)$ where $v_{JM} \in V_{JM}, v_{AFH} \in V_{AFH}$, and $(v_{JM},v_{AFH}) \leftrightarrow v_{JM}$ and v_{AFH} represent the same element in their respective models.

- Information on cross-cutting relationships created by aspect-oriented elements such as advice or inter-type fields and methods. Each IRelationship identifies the relationship type (such as "advises" or "declared on"), the source V_{AFH} node, and the destination V_{AFH} nodes [15]. This cross-cutting information may be represented as a graph $CC = (V_{CC},E_{CC})$ where $V_{CC} = V_{AFH}$ and $E_{CC} = Advises_{CC} \cup DeclaredOn_{CC}$. $Advises_{CC}$, which identifies cross-cutting advice relationships, contains edges (v_1,v_2) where $v_1 \in Advices_{AFH}, v_2 \in V_{CC}$ and $(v_1,v_2) \leftrightarrow v_1$ advises v_2. $DeclaredOn_{CC}$, which identifies cross-cutting relationships such as inter-type relationships, contains edges (v_1,v_2) where $v_1,v_2 \in V_{CC}$ and $(v_1,v_2) \leftrightarrow v_1$ declares an inter-type element affecting v_2.

Figure 6 includes partial depictions of *JM*, *AFH*, *AJJ*, and *CC* that illustrate how the four graphs are related.

4.1.2 Extract Facts From Compiler Model

Once the project is built, our approach extracts the program facts of interest from the Java Model and AspectJ Facade. As illustrated in Figure 2, extraction is a three-stage process.

Extract Classes In the first stage, projects, packages, and classes are extracted from the Java Model. Figure 3 outlines the algorithm used to traverse the *JM* graph and extract the desired program facts. This algorithm is derived from the Topcased reverse-engineering plugin [47]. Although it initially appears to be a standard graph-traversal algorithm, it introduces one significant contribution: the ability to use multiple graphs to separate object-oriented fact extraction from aspect-oriented fact extraction. As noted above, a "class" within the *JM* graph could be either a class or an aspect. (Within this discussion and the algorithms, a *JM* "class" will be referred to as a JavaModelClass.) An aspect isn't identified as an aspect within a V_{JM} node; therefore, it is necessary to use graph *AJJ* to associate the V_{JM} node with its corresponding V_{AFH} node which can identify a JavaModelClass as either an aspect or a true Java class. If the V_{AFH} node identifies the JavaModelClass as a true Java class, *JM* graph traversal proceeds normally.

Although the same information could be extracted from *AFH* alone, the combined *JM-AJJ-AFH* approach maximizes reuse of existing reverse-engineering frameworks and systems that are designed to work with *JM*. This exploits the low coupling between base object-oriented elements and aspect-oriented elements, allows these elements to be transformed separately, and enables us to focus on the aspect-specific details of the solution [6].

The extractFacts() function referenced by Figure 3 extracts the information listed in Table 4 from the tuple associated with a V_{JM} node and returns it as a program fact.

TABLE 4

INFORMATION EXTRACTED BY EXTRACTFACTS()

Node Type	Information Extracted
Project	Name
Package	Name
Enumeration	Name, modifier flags
Interface	Name, template parameters, modifier flags, generalizations
Class including aspects	Name, template parameters, modifier flags, generalizations
Property including pointcuts and inter- type fields and constructors	Name, types and associations, modifier flags
Operation including inter-type methods and advice	Name, modifier flags, parameters, return type

Extract Aspects Aspects are extracted from *JM, AFH,* and *AJJ* in this stage. Figure 4 outlines the algorithm used to transform the aspect-associated elements of *JM, AFH,* and *AJJ* into the program facts contained in *F*. Although this algorithm could have been designed to traverse *AFH* exclusively, it parallels Figure 3 by traversing *JM* in search of JavaModelClasses, then checking *AJJ* and *AFH* for aspects. This approach maximizes reuse of existing *JM*-centered Java reverse-engineering systems and permits the two stages to be implemented as a single hybrid class/aspect stage.

If the V_{AFH} node reports that the element is an aspect, this approach extracts inter-type elements, pointcuts, and advice from the V_{AFH} node. The extractFacts() function extracts the same program fact information for V_{JM} and V_{AFH} nodes; see Table 4 for the extracted information. V_{AFH} nodes does not contain cross-cutting relationship information; that information is extracted in the next stage.

Extract Cross-Cutting Relationships Once the object-oriented and aspect-oriented elements are extracted, cross-cutting relationships are extracted from the *CC* graph. As noted above, each edge in E_{CC} connects a source V_{AFM} node to a target V_{AFM} node. The *F* node corresponding to the source V_{AFM} node already has aspect-related stereotypes from the previous stage (Advice or StaticCrossCuttingFeature), so the relationship is extracted by adding the target element(s) as a property to the stereotype. The stereotypes and properties associated with each relationship are described in Table 5.

TABLE 5
STEREOTYPE PROPERTIES ADDED FOR EACH
CROSS-CUTTING RELATIONSHIP

E_{CC} Edge Type	Source Stereotype	Target Added To Source Stereotype As Property
$Advises_{CC}$	Advice	advised Element
$DeclaredOn_{CC}$	Static Cross Cutting Feature	on Type

```
input  : A set $P$ of $JavaProject_{JM}$ nodes
output: A set $F$ of program facts extracted from $P$
$F \longleftarrow \emptyset$;
foreach $project_{JM} \in JavaProject_{JM}$ do
    $F \longleftarrow F+$ {extractFacts($project_{JM}$)};
    foreach $package_{JM} \in Packages_{JM} | (project_{JM}, package_{JM}) \in V_{JM}$
    do
        $F \longleftarrow F+${extractFacts($package_{JM}$)};
        foreach $compUnit_{JM} \in$
        $CompilationUnits_{JM} | (package_{JM}, compUnit_{JM}) \in V_{JM}$ do
            foreach $type_{JM} \in Types_{JM} | (compUnit_{JM}, type_{JM}) \in V_{JM}$
            do
                if $type_{JM}$ is an enumeration then
                    $F \longleftarrow F+${extractFacts($type_{JM}$)};
                else if $type_{JM}$ is an interface then
                    $F \longleftarrow F+${extractFacts($type_{JM}$)};
                    foreach
                    $field_{JM} \in Fields_{JM} | (type_{JM}, field_{JM}) \in V_{JM}$ do
                        $F \longleftarrow F+${extractFacts($field_{JM}$)};
                    end
                    foreach
                    $method_{JM} \in Methods_{JM} | (type_{JM}, method_{JM}) \in V_{JM}$
                    do
                        $F \longleftarrow F+${extractFacts($method_{JM}$)};
                    end
                else
                    /*$type_{JM}$ is a JavaModelClass, which could be
                      either a class or an aspect                   */
                    $type_{AFH} \leftarrow V_{AFH}$ node such that
                    $(type_{AFH}, type_{JM}) \in E_{AJJ}$;
                    if $type_{AFH}$ is an Java class then
                        $F \longleftarrow F+${extractFacts($type_{JM}$)};
                        foreach
                        $field_{JM} \in Fields_{JM} | (type_{JM}, field_{JM}) \in V_{JM}$ do
                            $F \longleftarrow F+${extractFacts($field_{JM}$)};
                        end
                        foreach $method_{JM} \in$
                        $Methods_{JM} | (type_{JM}, method_{JM}) \in V_{JM}$ do
                            $F \longleftarrow F+${extractFacts($method_{JM}$)};
                        end
                    end
                    /*When $type_{JM}$ is an aspect, see Figure 4    */
                end
            end
        end
    end
end
```

FIGURE 3: CLASS EXTRACTION ALGORITHM

```
input  : A set $P$ of $JavaProject_{JM}$ nodes
output: A set $F$ of program facts extracted from $P$
$F \longleftarrow \emptyset$;
foreach $project_{JM} \in JavaProject_{JM}$ do
    foreach $package_{JM} \in Packages_{JM} | (project_{JM}, package_{JM}) \in V_{JM}$
    do
        foreach $compUnit_{JM} \in$
        $CompilationUnits_{JM} | (package_{JM}, compUnit_{JM}) \in V_{JM}$ do
            foreach $type_{JM} \in Types_{JM} | (compUnit_{JM}, type_{JM}) \in V_{JM}$
            do
                if $type_{JM}$ is an class then
                    /*$type_{JM}$ is a JavaModelClass, which could be
                      either a class or an aspect                    */
                    $type_{AFH} \leftarrow V_{AFH}$ node such that
                    $(type_{AFH}, type_{JM}) \in E_{AJJ}$;
                    if $type_{AFH}$ is an aspect then
                        $aspectFact$ ←extractFacts($type_{AFH}$);
                        apply Aspect stereotype to $aspectFact$;
                        $F \longleftarrow F + \{aspectFact\}$;
                        foreach $child_{AFH} \in V_{AFH}$ such that
                        $(type_{AFH}, child_{AFH}) \in E_{AFH}$ do
                            if $child_{AFH}$ is an inter-type field or inter-type
                            constructor then
                                $property$ ←extractFacts($child_{AFH}$)
                                  /*Property                                 */
                                apply StaticCrossCuttingFeature stereotype
                                to $property$;
                                $F \longleftarrow F + \{property\}$;
                            else if $child_{AFH}$ is an inter-type method then
                                $method$ ←extractFacts($child_{AFH}$)
                                  /*Operation                                */
                                apply StaticCrossCuttingFeature stereotype
                                to $method$;
                                $F \longleftarrow F + \{method\}$;
                            else if $child_{AFH}$ is a pointcut then
                                $property$ ←extractFacts($child_{AFH}$)
                                  /*Property                                 */
                                apply PointCut stereotype to $property$;
                                $F \longleftarrow F + \{property\}$;
                            else if $child_{AFH}$ is an advice then
                                $method$ ←extractFacts($child_{AFH}$)
                                  /*Operation                                */
                                apply Advice stereotype to $method$;
                                $F \longleftarrow F + \{method\}$;
                            end
                        end
                    end
                end
            end
        end
    end
end
```

FIGURE 4: ASPECT EXTRACTION ALGORITHM

4.1.3 Translate Facts to XMI

Once the facts have been extracted from the Compiler Model, the facts are translated on a one-to-one basis to an XMI file which allows us to represent program facts at a suitable level for AspectJ/Java reverse engineering. As Section 2 explains, UML is already used by other tools to structure and represent reverse-engineered program facts for human use. By using XMI to encode UML as XML, we can represent our reverse-engineered program facts in a format suitable for human and automated use. Although UML (and XMI by extension) does not provide built-in support for AspectJ and aspect-oriented concepts, we have developed a profile based on Evermann [16] that uses standard UML extension mechanisms to represent these concepts. This profile, which is illustrated by Figure 5, allows the XMI file to represent both object-oriented and aspect-oriented program facts at a suitable level for reverse engineering. Other projects have used the Evermann profile "because of its maturity to implement aspects" [26]. We defined the AspectJ extensions via standard UML extension mechanisms so other XMI-compliant tools can read our XMI output with AspectJ information without modifications.

Furthermore, XMI is an Object Management Group standard [40], thereby satisfying our requirement for standards-based representation. The XMI file with standard UML extension mechanisms can be used by any existing tool that can read XMI, thereby making AspectJ program fact information more readily available to other researchers.

Figure 6 is a simplified representation of the transformation from AspectJ source code to the four graphs (***JM,AFH,AJJ,*** and ***CC***) , extracted facts, and XMI. Sample code is from UndoableCommand.aj in AJHotDraw, an aspect-oriented refactoring of the JHotDraw graphical user interface framework [36, 37]. As noted earlier, aspects are correctly designated in the Facade, but appear in the Java Model as classes.

4.2 Tool Support

We have developed an open-source Eclipse plugin that uses this approach to extract program facts for Java 1.6 and AspectJ 1.6.7 [30, 32]. This plugin is available at http://www.utdallas.edu/~jwk043000. Our plugin leverages existing open-source technologies and solutions in a number of ways:

- The Eclipse plugin structure allows other tools such as AOVis to extend the existing Eclipse environment [7].
- TheAJDT2.0.2JavaModelandAJProjectModelFacade are used as the source of program facts for the project being analyzed.
- Our plugin extended the reverse-engineering plugin from Topcased (version 2.5) [47] to provide Java reverse-engineering, allowing us to focus on AspectJ-specific problem details.

The current plugin adds a pop-up menu command that allows users to create an XMI file for an Eclipse AspectJ project or package by right-clicking it within Package Explorer. Internally, part of the command code is an extension of the Topcased Java2UMLConverter class, thereby maximizing usage of the Topcased Java reverse-engineering solution. This extension combines the first two stages described in Section 4.1.2 to extract and store the facts using the UML2 classes available from the Eclipse UML2 plugin [24, 43, 38]. The plugin then uses the UML2 plugin to transform the UML2 facts into XMI.

4.3 Validation

We selected several open-source AspectJ projects to validate this approach. For each selected project, we started an Eclipse instance with our plugin installed, loaded the selected project, and executed the plugin's command via the project's right-click menu in Package Explorer. This generated an XMI file which could be loaded into the project and viewed with any XMI-compliant tool such as the UML editor from the Eclipse UML2Tools plugin [17]. The XMI file was then validated against expected results generated by manual analysis or AJDT tools such as the AspectJ editor and the Cross-Reference view.

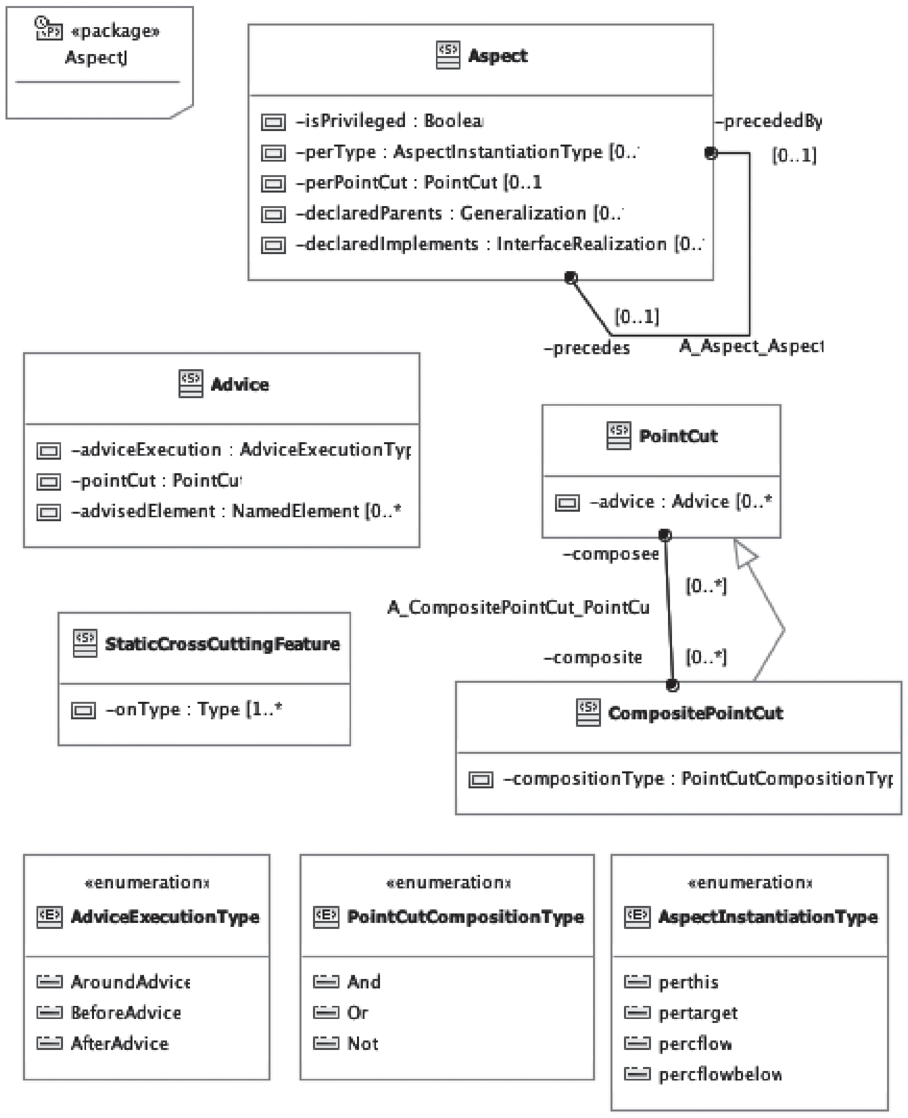
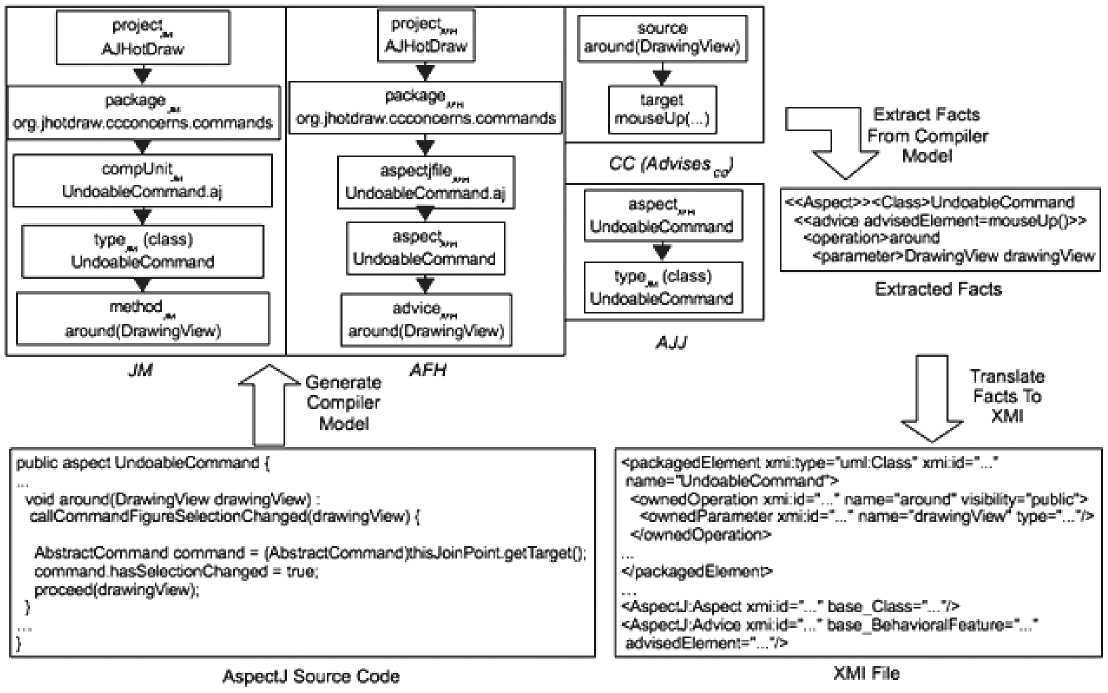


FIGURE 5: ASPECTJ UML PROFILE


project JM AJHotDraw
package JM org.jhotdraw.ccconcerns.commands
compUnit JM UndoableCommand.aj
type JM (class) UndoableCommand
method JM around(DrawingView)
JM
project AFH AJHotDraw
package AFH org.jhotdraw.ccconcerns.commands
aspectjfile AFH UndoableCommand.aj
aspect AFH UndoableCommand
advice AFH around(DrawingView)
AFH
source around(DrawingView)
target mouseUp(...)
CC (Advises CC)
aspect AFH UndoableCommand
type JM (class) UndoableCommand
AJJ
Extract Facts From Compiler Model
Extracted Facts
Generate Compiler Model
Translate Facts To XMI
AspectJ Source Code
XMI File

```
<<Aspect>><Class>UndoableCommand
  <<advice advisedElement=mouseUp()>>
   <operation>around
     <parameter>DrawingView drawingView
```

```
public aspect UndoableCommand {
...
 void around(DrawingView drawingView) :
  callCommandFigureSelectionChanged(drawingView) {

   AbstractCommand command = (AbstractCommand)thisJoinPoint.getTarget();
   command.hasSelectionChanged = true;
   proceed(drawingView);
 }
...
}
```

```
<packagedElement xmi:type="uml:Class" xmi:id="..."
 name="UndoableCommand">
  <ownedOperation xmi:id="..." name="around" visibility="public">
   <ownedParameter xmi:id="..." name="drawingView" type="..."/>
  </ownedOperation>
...
</packagedElement>
...
<AspectJ:Aspect xmi:id="..." base_Class="..."/>
<AspectJ:Advice xmi:id="..." base_BehavioralFeature="..."
 advisedElement="..."/>
```

FIGURE 6: TRANSFORMATION OF AJHOTDRAW ASPECTJ CODE TO XMI.

The following projects were used to validate the proposed approach:

AJHotDraw: an aspect-oriented refactoring of the JHotDraw graphical user interface framework [36, 37]. An example of AJHotDraw validation results can be seen in Figure 6.

AJHSQLDB: an aspect-oriented refactoring of HSQLDB (HyperSQL Database), a Java-based database management system [45].

Contract4J: Design by Contract for Java [51].

***J-Pool:** an aspect-oriented enhancement to the *J Java analysis framework, developed by the Sable Research Group and the Programming Tools Group at Oxford University as part of the AspectJ Benchmark Suite [42, 13].

Project metrics are shown in Table 6, including project sizes and the amount of extracted data in the form of XMI file sizes.

TABLE 6

VALIDATION PROJECT SIZES

Project	LOC	Classes	Aspects	XMI File Size (MB)
AJHotDraw	22,104	294	31	2.1
AJHSQLDB	75,556	250	31	4.1
Contract4J	10,722	209	15	0.7
*J-Pool	38,217	512	1	2.7

5 Conclusions and Future Work

We have developed the AOVis framework which provides the following unique contributions to the field of aspect-oriented reverse engineering:

- It provides an automated solution for abstracting reverse-engineered AspectJ systems to multiple levels of design models, including the depiction of cross-cutting concerns as well as base concerns.
- It leverages the production version of standard AspectJ tools to provide UML visualization of AspectJ systems. Earlier visualization approaches required modification of the standard tools.

As noted above, a number of extraction sources are available including off-the-shelf compiler models, models from extended and instrumented compilers, BCEL, and external parsers. We chose to leverage the standard AJDT compiler model (Java Model and AspectJ Facade) rather than other approaches; based on our survey of the literature, we believe that this project is the first to use production AJDT information to reverse-engineer AspectJ systems. We have successfully verified that our approach can extract aspect-oriented and object-oriented program facts from AspectJ and Java source code.

Although the AJDT compiler model is available for other developers, the APIs are still evolving. The documentation warns that it is out of date [15]; furthermore, documentation on the key IProgramElement interface is incomplete, requiring researchers to consult the AJDT source code when using the API. We believe that our approach and plug-in will be extremely useful to other researchers with a need for AspectJ program facts.

We have already been able to validate our use of XMI files and AspectJ profile as the intermediate language by leveraging existing XMI-compliant tools as part of tool support:

- UML2Tools Eclipse plug-in [17] to read and validate plug-in output.
- SWI Prolog SGML/XML parser [52] to import XMI files into the Prolog environment.

We expect that this approach will be useful in projects such as design visualization, traceability relationships, automatic test case generation, code metrics, and product line variation refactoring.

Ongoing and future work on the project includes the following activities:

- As noted in Section 1, the second fundamental activity of reverse engineering is transformation or abstraction: abstracting the system from raw program facts into suitable models [20]. Through this process, reverse engineers try to discover the system's architecture as part of understanding the system [18]. Aspect-oriented elements of the system should also be identified as part of the architecture; Hannemann and Kiczales [23] notes that some object-oriented elements, such as design patterns, are implemented more cleanly as aspects. We are currently developing abstraction engines to identify the underlying architecture of the system and transform the program-fact XMI files into architectural-model XMI files.

- The final step in reverse engineering is to visualize the models developed in the previous step. Visualization "reduces the burden on the human brain" [19] and aids system understanding. We are currently developing extended UML visualization of the architectural and detailed design models, and are investigating the feasibility and merits of 3D visualization as a means of separating aspect-oriented and object-oriented concerns.

- Although we have already been able to validate our decision to use XMI by leveraging existing XMI-compliant tools as part of tool support, we wish to confirm our hypothesis that outside tools will be able to leverage our own tool. Future work will include verification that other tools will be able to use AOVis output.

In addition, we have identified the following issues which will also be studied in future work:

- Since this is a rule-based approach, speed and scalability are both major concerns.

- The current plug-in collects only a subset of the AspectJ language details identified by Evermann [16], primarily because these details aren't present in the AspectJ Facade. The absence of these details, such as the connections between pointcuts and advice, led to the use of the Extended Compiler Model in [8].

- Some language features such as inner private static aspects are not currently supported by the plug-in. We expect to discover other unusual language features throughout validation, and future work will include enhancements to handle these features.

- Although AJDT correctly compiles and weaves aspects located inside ".java" files, it omits some of the aspect information from the compiler model. This omission is visible in Eclipse views as well as the compiler model data. We are studying this issue as it applies to our work.

ACKNOWLEDGEMENT

We would like to thank the AJDT, Topcased, and UML2 teams for making these tools available to the community. We would also like to thank Andrew Eisenberg for his quick response to our AJDT questions.

REFERENCES

[1] F. E. Alam, J. Evermann, and A. Fiech, "Modeling for dynamic aspect-oriented development," in *C3S2E '09: Proceedings of the 2009 C3S2E conference*, pages 143–147, New York, NY, USA, 2009.

[2] Apache Software Foundation, BCEL. http://jakarta.apache.org/bcel/, Jun 2006.

[3] AspectJ, The AspectJ™ Programming guide. http://www.eclipse.org/aspectj/doc/released/progguide/index.html, 2003.

[4] AspectJ, Frequently asked questions about AspectJ. http://www.eclipse.org/aspectj/doc/released/faq.html, November 2006.

[5] P. Avgustinov, E. Hajiyev, N. Ongkingco, O. de Moor, D. Sereni, J. Tibble, and M. Verbaere, "Datalog semantics of static pointcuts in AspectJ 1.2.1," Technical Report abc-2006-2, AspectBench Compiler Project, July 2006.

[6] J. Bennett, K. Cooper, and L. Dai, "Aspect-oriented model-driven skeleton code generation: A graph-based transformation approach," *Science of Computer Programming*, vol. 75, no. 8:689 – 725, 2010. Designing high quality system/software architectures.

[7] A. Bolour, "Notes on the Eclipse plug-in architecture," http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, July 2003.

[8] W. Coelho, "Presenting crosscutting structure with active models," Master's *thesis, University of British Columbia*, August 2005.

[9] K. M. Cooper and J. Koch, "Reverse engineering visual design models from code: What are the issues?," In *Distributed Multimedia Systems, 2010. 16th International Conference*, 297–299, October 2010.

[10] D. Dahiya and R. K. Sachdeva, "Approaches to aspect oriented design: a study," *ACM SIGSOFT Softw. Eng. Notes*, vol. 31, no. 5:1–4, 2006.

[11] L. Dai, "Formal Design Analysis Framework: An Aspect-Oriented Architectural Framework," *PhD thesis, The University of Texas at Dallas*, December 2005.

[12] L. Dai and K. Cooper, "Using fdaf to bridge the gap between enterprise and software architectures for security," *Science of Computer Programming*,. Special Issue on the 5th International Workshop on System/Software Architectures (IWSSA'06), vol. 66, no. 1,:87 – 102, April 2007.

[13] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge, "Measuring the dynamic behaviour of AspectJ programs," in *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 150–169, New York, NY, USA, 2004.

[14] A. Eisenberg. Re: [ajdt-dev] aj handle diffs - ijavaelement.gethandleidentifier() vs. irelationship.getsourcehandle(). http://dev.eclipse.org/mhonarc/lists/ajdt-dev/msg01487.html, Apr 2010.

[15] A. Eisenberg, K. Hassani, H. Hawkins, and M. Chapman, "Developer's guide to building tools on top of AJDT and AspectJ," http://wiki.eclipse.org/index.php/Developer%27s_guide_to_building_tools_on_top_of_AJDT_and_AspectJ, March 2010.

[16] J. Evermann, "A meta-level specification and profile for AspectJ," in UML. In *AOM '07: Proceedings of the 10th international workshop on Aspect-oriented modeling*, pages 21–27, New York, NY, USA, 2007. ACM.

[17] T. Fesenko and M. Golubev, MDT-UML2Tools. http://wiki.eclipse.org/MDT-UML2Tools, February 2010.

[18] R. Fiutem, E. Merlo, G. Antoniol, and P. Tonella, "Understanding the architecture of software systems," in *Program Comprehension, 1996, Proceedings., Fourth Workshop on*, pages 187–196, Mar 1996.

[19] Y. Ghanam and S. Carpendale, "A survey paper on software architecture visualization," Technical Report 2008-906-19, University of Calgary, 06, 2008.

[20] J. Girard and R. Koschke, "Finding components in a hierarchy of modules: a step towards architectural understanding," in *Proceedings International Conference on Software Maintenance*, pages 58–65. IEEE, IEEE Computer Society Press, 1997.

[21] O. Hachani, "Extending UML meta-model for AspectJ: AspectJ/UML," Technical Report 1.0, LSR-IMAG, Sigma Team, June 2003.

[22] S. Hanenberg, D. Stein, and R. Unland, "From aspect-oriented design to aspect-oriented programs: tool-supported translation of JPDDs into code," in *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 49–62, New York, NY, USA, 2007.

[23] J. Hannemann and G. Kiczales, "Design pattern implementation in Java and AspectJ" in *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM.

[24] K. Hussey and J. Bruck, Getting started with UML2. http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Getting_Started_with_UML2/article.html, July 2008.

[25] IBM. Rational Application Developer for WebSphere Software. http://www-01.ibm.com/software/awdtools/developer/application/, July 2010.

[26] J. U. Júnior, V. V. Camargo, and C. V. F. Chavez, "Uml-aof: a profile for modeling aspect-oriented frameworks," in *AOM '09: Proceedings of the 13th workshop on Aspect-oriented modeling*, pages 1–6, New York, NY, USA, 2009. ACM.

[27] R. Khaled, J. Noble, and R. Biddle, "Inspectj: program monitoring for visualisation using aspectj," in *ACSC '03: Proceedings of the 26th Australasian computer science conference*, pages 359–368, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in M. Aksit and S. Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0053381.

[29] H. M. Kienle and H. A. Müller, "Rigi–an environment for software reverse engineering, exploration, visualization, and redocumentation," *Science of Computer Programming*, 75(4):247 – 263, 2010. Experimental Software and Toolkits (EST 3): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008).

[30] J. Koch. Jeffrey Koch, http://www.utdallas.edu/~jwk043000, August 2010.

[31] J. Koch, V. Aare, S. Pagade, K. Cooper, J. Cangussu, and K. Hamlen, "3D visualization of aspect-oriented source code," in *Proceedings of the 2010 Symposium on Information Systems and Computing,* Raytheon, April 2010.

[32] J. Koch, S. Bohra, R. Goel, S. Pagade, and K. Cooper, "AODVis: Leveraging Eclipse plugins to reverse engineer and visualize AspectJ/Java source code," in *TOPI 2011 (ICSE): Proceedings of the 1st Workshop on Developing Tools as Plug-ins*, May 2011. Accepted.

[33] R. Krikhaar, "Reverse architecting approach for complex systems," in *Software Maintenance, 1997. Proceedings., International Conference on*, pages 4–11, Oct 1997.

[34] T. Kuhn and O. Thomann, Abstract syntax tree, http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html, November 2006.

[35] C. Laffra and N. Veys, FAQ What is the Java model? http://wiki.eclipse.org/FAQ_What_is_the_Java_model%3F, June 2006.

[36] M. Marin., AJHotDraw, http://swerl.tudelft.nl/bin/view/AMR/AJHotDraw, July 2007.

[37] M. Marin, L. Moonen, and A. van Deursen, "An integrated crosscutting concern migration strategy and its application to JHOTDRAW," in *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pages 101 –110, Sept. 2007.

[38] M. Merdes and D. Dorsch, "Experiences with the development of a reverse engineering tool for uml sequence diagrams: a case study in modern java development," in *PPPJ '06: Proceedings of the 4th international symposium on Principles and practice of programming in Java*, pages 125–134, New York, NY, USA, 2006, ACM.

[39] M. Mosconi, A. Charfi, J. Svacina, and J. Wloka, "Applying and evaluating aom for platform independent behavioral uml models," in *AOM '08: Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling*, pages 19–24, New York, NY, USA, 2008, ACM.

[40] Object Management Group, MOF 2.0/XMI mapping, version 2.1.1. http://www.omg.org/spec/XMI/2.1/PDF, December 2007.

[41] A. Raza, G. Vogel, and E. Plödereder. Bauhaus, "A tool suite for program analysis and reverse engineering," in L. M. Pinho and M. G. Harbour, editors, *Ada-Europe*, volume 4006 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 2006.

[42] Sable Research Group, Sable benchmarks. http://www.sable.mcgill.ca/benchmarks/, October 2004.

[43] N. Skrypuch, UML2. http://www.eclipse.org/modeling/mdt/?project=uml2, 2010.

[44] D. Stein, S. Hanenberg, and R. Unland, "A UML-based aspect-oriented design notation for AspectJ," in *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 106–112, New York, NY, USA, 2002, ACM.

[45] M. Störzer. AJHSQLDB, http://sourceforge.net/projects/ajhsqldb/, August 2006.

[46] A. Telea and L. Voinea, "An interactive reverse engineering environment for large-scale C++ code," in *SoftVis '08: Proceedings of the 4th ACM symposium on Software visualization*, pages 67–76, New York, NY, USA, 2008, ACM.

[47] Topcased. Topcased, home. http://www.topcased.org/, Sep 2009.

[48] J. van Zyl, BCEL projects. http://jakarta.apache.org/bcel/projects.html, Jun 2006.

[49] L. Vogel. "Eclipse JDT, abstract syntax tree (AST) and the Java model - tutorial.," http://www.vogella.de/articles/EclipseJDT/article.html, May 2009.

[50] D. Walker, S. Zdancewic, and J. Ligatti, "A theory of aspects," in *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, New York, NY, USA, 2003, ACM.

[51] D. Wampler, Contract4J: Design by Contract®;for Java. http://www.contract4j.org, March 2010.

[52] J. Wielemaker, SWI-Prolog SGML/XML parser. http://www.swi-prolog.org/pldoc/package/sgml.html, 2010. Accessed 06-Dec-2010.

[53] X. Zhang, M. Young, and J. H. E. F. Lasseter, "Refining code-design mapping with flow analysis," in *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 231–240, New York, NY, USA, 2004, ACM.

ABOUT THE AUTHORS

Mr. Jeffrey Koch: Mr. Koch is an Assistant Professor in the Department of Computer Science and Information Technology, Tarrant County College (TCC), Hurst, Texas, USA. Prior to joining the TCC faculty in 2008, his industry experience included Honeywell Commercial Flight Systems, the United States Department of State, and Texas Instruments. Mr. Koch is also a Ph.D. software engineering candidate in the Department of Computer Science, The University of Texas at Dallas (UTD), Richardson, Texas, USA. Mr. Koch earned his bachelor's degree in computer science at Arizona State University, Tempe, Arizona, USA, and his master's degree in software engineering at UTD. He is a professional member of ACM.

Dr. Kendra Cooper: Dr. Cooper is an Associate Professor in the Department of Computer Science in the Erik Jonsson School of Engineering and Computer Science. She received her Ph.D. in Electrical and Computer Engineering from The University of British Columbia and has over 90 peer reviewed publications in journals, conferences, symposia, and workshops. She serves on numerous journal editorial boards and programme committees in her field. Dr. Cooper has worked extensively in the early phases of the software and systems development in industrial and academic settings. Her research focuses on the component-based architecture of complex, large-scale systems and engineering education.